Hans-Wolfgang Loidl
Ricardo Peña (Eds.)

LNCS 7829

Trends in Functional Programming

13th International Symposium, TFP 2012
St. Andrews, UK, June 2012
Revised Selected Papers

Springer

Hans-Wolfgang Loidl   Ricardo Peña (Eds.)

Trends in Functional Programming

13th International Symposium, TFP 2012
St. Andrews, UK, June 12-14, 2012
Revised Selected Papers

Springer

Volume Editors

Hans-Wolfgang Loidl
Heriot-Watt University
School of Mathematics and Computer Sciences
Edinburgh, EH14 4AS, UK
E-mail: h.w.loidl@hw.ac.uk

Ricardo Peña
Universidad Complutense de Madrid
Facultad de Informática
c/. Profesor José García Santesmases s/n
28040 Madrid, Spain
E-mail: ricardo@sip.ucm.es

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40446-7 e-ISBN 978-3-642-40447-4
DOI 10.1007/978-3-642-40447-4
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013945452

CR Subject Classification (1998): D.1.1, D.1.3, D.3, F.3, F.4, E.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

With the 13th Symposium on Trends in Functional Programming (TFP 2012), held in St. Andrews, Scotland, the TFP series returned to its origins in Scotland. TFP is the heir of the successful series of Scottish Functional Programming Workshops, which ran during 1999–2002, organized at several Scottish universities. Reflecting the increasingly international audience of these events, from 2003 onwards these became the Symposium on Trends in Functional Programming, and were held in Edinburgh (2003), in Munich (2004), in Tallinn (2005), in Nottingham (2006), in New York (2007), in Nijmegen (2008), in Komarno (2009), in Oklahoma (2010), and in Madrid (2011).

In June 2012 TFP was the main event of a week of functional programming extravaganza at the University of St. Andrews, attended by more than 80 registered participants. Co-located events were the International Workshop on Trends in Functional Programming in Education (TFPIE 2012), the Workshop on 70 Years of Lambda Calculus, an Erlang Factory Lite, and a technical workshop on Patterns for Multicores (ParaPhrase/RELEASE projects).

In total, TFP 2012 received 49 submissions for the draft proceedings. After a screening process, 41 of these papers were accepted for the draft proceedings and for presentation at the symposium. The screening process was performed by a subset of the Program Committee and aimed to ensure that all contributions are in scope and contain relevant information for the TFP audience. After TFP 2012, all authors of presentations were invited to submit full papers to a formal refereeing process, the result of which is presented in these proceedings. The papers in these proceedings have been judged by members of the Program Committee on their contribution to the research area with appropriate criteria applied to each category of paper. For each paper at least three referee reports were produced. Based on these reports, the international Program Committee selected the papers presented in these proceedings. In total, 36 papers were submitted for the formal refereeing process and 18 papers were accepted.

Additionally to these refereed papers, these proceedings feature a paper by a high-profile member of the research community, David A. Turner, Professor Emeritus at Middlesex University and at the University of Kent. Prof. Turner is the inventor of such influential functional languages as Miranda, KRC and SASL. Prof. Turner presented an invited talk on "The History of Lazy Functional Programming Languages" at the symposium and graciously agreed to summarize his insights into the development of functional languages in the first paper in these proceedings. We would like to thank Prof. Turner for his contribution to TFP, which was most appreciated by the attendees.

TFP traditionally pays special attention to research students, acknowledging that students are almost by definition part of new subject trends. A student paper is one for which the authors state that the paper is mainly the work of students, the students are listed as first authors, and a student would present the paper. These papers also receive an extra round of feedback by the Program Committee before they are submitted to the standard review process for formal publication. In this way, students can improve their papers before they compete within a full formal refereeing process. In 2012, 23 of the 49 papers submitted for the draft proceedings were student papers. Acknowledging the contributions by student papers, the TFP Program Committee awards a prize for the best student paper each year. We are delighted to announce that for TFP 2012 the TFP prize for the *best student paper* was awarded to:

> *Luminous Fennell and Peter Thiemann* for the paper
> "The Blame Theorem for a Linear Lambda Calculus with Type Dynamic."

We are proud to announce that this year, for the first time, an EAPLS Best Paper Award elevates one of the papers in these proceedings as making an outstanding contribution to the field. Based on a short-list of papers provided by the TFP 2012 Program Committee, the EAPLS board made the final selection for this award. We are therefore proud to announce that the *EAPLS prize for the best paper of TFP 2012* was awarded to:

> *Josef Svenningsson and Emil Axelsson* for the paper
> "Combining Deep and Shallow Embedding for EDSL."

Finally, we are grateful for the sponsorship by Erlang Solutions Ltd. and the Scottish Informatics and Computer Science Alliance (SICSA). The TFP Best Student Paper award is funded by the TFP series. The EAPLS Best Paper Award is funded by the European Association for Programming Languages and Systems.

March 2013

Hans-Wolfgang Loidl
Ricardo Peña

Organization

Committees

Program Chair

Hans-Wolfgang Loidl	Heriot-Watt University, U.K.

Symposium Chair

Ricardo Pena	Universidad Complutense de Madrid, Spain

General Chair

Kevin Hammond	University of St. Andrews, U.K.

Program Committee

Peter Achten	Radboud University Nijmegen, The Netherlands
Jost Berthold	University of Copenhagen, Denmark
Edwin Brady	University of St. Andrews, U.K.
Matthias Blume	Google, U.S.A.
Clemens Grelck	University of Amsterdam, The Netherlands
Kevin Hammond	University of St. Andrews, U.K.
Graham Hutton	University of Nottingham, U.K.
Patricia Johann	University of Strathclyde, U.K.
Hans-Wolfgang Loidl (PC Chair)	Heriot-Watt University, U.K.
Jay McCarthy	Brigham Young University, Utah, U.S.A.
Rex Page	University of Oklahoma, U.S.A.
Ricardo Peña	Complutense University of Madrid, Spain
Kostis Sagonas	Uppsala University, Sweden
Manuel Serrano	INRIA Sophia Antipolis, France
Mary Sheeran	Chalmers, Sweden
Nikhil Swamy	Microsoft Research, Redmond, U.S.A.
Phil Trinder	Heriot-Watt University, U.K.
Wim A. Vanderbauwhede	University of Glasgow, U.K.
Marko van Eekelen	Radboud University Nijmegen, The Netherlands

David Van Horn	Northeastern University, U.S.A.
Malcolm Wallace	Standard Chartered, U.K.
Viktória Zsók	Eötvös Loránd University, Hungary

Local Organization

Kevin Hammond
Edwin Brady
Vladimir Janjic

Table of Contents

Some History of Functional Programming Languages

(Invited Talk)

D.A. Turner

University of Kent & Middlesex University

Abstract. We study a series of milestones leading to the emergence of lazy, higher order, polymorphically typed, purely functional programming languages. An invited lecture given at TFP12, St Andrews University, 12 June 2012.

Introduction

A comprehensive history of functional programming languages covering all the major streams of development would require a much longer treatment than falls within the scope of a talk at TFP, it would probably need to be book length. In what follows I have, firstly, focussed on the developments leading to lazy, higher order, polymorphically typed, purely functional programming languages of which *Haskell* is the best known current example. Secondly, rather than trying to include every important contribution within this stream I focus on a series of snapshots at significant stages.

We will examine a series of milestones:

1. Lambda Calculus (Church & Rosser 1936)
2. LISP (McCarthy 1960)
3. Algol 60 (Naur et al. 1963)
4. ISWIM (Landin 1966)
5. PAL (Evans 1968)
6. SASL (1973–83)
7. Edinburgh (1969–80) — NPL, early ML, HOPE
8. Miranda (1986)
9. Haskell (1992 ...)

1 The Lambda Calculus

The lambda calculus (Church & Rosser 1936; Church 1941) is a typeless theory of functions. In the brief account here we use lower case letters for variables: $a, b, c \cdots$ and upper case letters for terms: $A, B, C \cdots$.

A *term* of the calculus is a variable, e.g. x, or an application AB, or an abstraction $\lambda x.A$ for some variable x. In the last case $\lambda x.$ is a *binder* and free occurrences of x in A become *bound*. A term in which all variables are bound is said to be *closed* otherwise it is *open*. The motivating idea is that closed terms represent functions.

In writing terms we freely use parentheses to remove ambiguity.
The calculus has three rules

$$
\begin{array}{ll}
(\alpha) & \lambda x.A \rightarrow_\alpha \lambda y.[y/x]A \\
(\beta) & (\lambda x.A)B \rightarrow_\beta [B/x]A \\
(\eta) & \lambda x.Ax \rightarrow_\eta A \quad \text{if } x \text{ not free in } A
\end{array}
$$

Here $[B/x]A$ means substitute B for free occurrences of x in A[1]. Rule α permits change of bound variable. Terms which are the same up to α-conversion, e.g. $\lambda x.x$ and $\lambda y.y$, are not usually distinguished.

The smallest reflexive, symmetric, transitive, substitutive relation on terms including $\rightarrow_\alpha$, $\rightarrow_\beta$ and $\rightarrow_\eta$, written $\Leftrightarrow$, is Church's notion of $\lambda-$*conversion*. If we omit symmetry from the definition we get an oriented relation, written $\Rightarrow$, called *reduction*.

An instance of the left hand side of rule β or η is called a *redex*. A term containing no redexes is said to be in *normal form*. A term which is convertible to one in normal form is said to be *normalizing*. There are non-normalizing terms, for example $(\lambda x.xx)(\lambda x.xx)$ which β-reduces to itself.

The three most important technical results are

Church-Rosser Theorem. If $A \Rightarrow B$ and $A \Rightarrow B'$ there is a term C such that $B \Rightarrow C$ and $B' \Rightarrow C$. An immediate corollary is that the normal form of a normalizing term is unique (up to α-conversion).

Second Church-Rosser Theorem. The normal form of a normalizing term can be found by repeatedly reducing its *leftmost outermost redex*, a process called normal order reduction.

Böhm's theorem. If A, B have distinct normal forms there is a context $C[]$ with $C[A] \Rightarrow \lambda x.(\lambda y.x)$ and $C[B] \Rightarrow \lambda x.(\lambda y.y)$.
This tells us that α, β, η-conversion is the *strongest possible* equational theory on normalizing terms — if we add any equation between non-convertible normalizing terms the theory becomes *inconsistent*, that is all terms are now interconvertible, e.g. we will have $x \Leftrightarrow y$.

The lambda calculus originates from an endeavour by Church, Curry and others to define an alternative foundation for mathematics based on functions rather than sets. The attempt foundered in the late 1920's on paradoxes analogous to those which sank Cantor's untyped set theory. What remained after the propositional parts of the theory were removed is a consistent equational theory of functions. Notwithstanding that it was devised before computers in the modern sense existed, the lambda calculus makes a simple, powerful and elegant programming language.

[1] Substitution includes systematic change of bound variables where needed to avoid *variable capture* — for details see any modern textbook, e.g. Hindley & Seldin (2008).

In the pure[2] untyped lambda calculus everything is a function — a closed term in normal form can only be an abstraction, $\lambda x.A$. An *applied lambda calculus* adds constants representing various types of data and primitive functions on them, for example natural numbers with $plus, times$ etc. and appropriate additional reduction rules — Church (1941) calls these δ-rules — this can be done while ensuring that Church-Rosser and other technical properties of the calculus are preserved. A type discipline can be imposed to prevent the formation of meaningless terms. There is thus a richly structured family of applied lambda calculi, typed and untyped, which continues to grow new members.

However, the pure untyped lambda calculus is already computationally complete. There are functional representations of natural numbers, lists, and other data. One of several possibilities for the former are the Church numerals

$$\begin{aligned}\overline{0} &= \lambda a.\lambda b.b\\ \overline{1} &= \lambda a.\lambda b.ab\\ \overline{2} &= \lambda a.\lambda b.a(ab)\\ \overline{3} &= \lambda a.\lambda b.a(a(ab)) \text{ etc. } \cdots\end{aligned}$$

Conditional branching can be implemented by taking

$$\begin{aligned}\overline{True} &\equiv \lambda x.(\lambda y.x)\\ \overline{False} &\equiv \lambda x.(\lambda y.y)\end{aligned}$$

We then have

$$\begin{aligned}\overline{True}AB &\Rightarrow A\\ \overline{False}AB &\Rightarrow B\end{aligned}$$

Recursion can be encoded using $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ which has the property, for any term A

$$YA \Rightarrow A(YA)$$

With this apparatus we can code all the recursive functions of type $N \to N$ (using N for the set of natural numbers) but also those of type $N \to (N \to N)$, $(N \to N) \to N$, $(N \to N) \to (N \to N)$ and so on up.

It is the power to define functions of higher type, together with clean technical properties — Church-Rosser etc. — that make lambda calculus, either pure or applied, a natural choice as a basis for functional programming.

At first sight it seems a restriction that λ creates only functions of one argument, but in the presence of functions of higher type there is no loss of generality. It is a standard result of set theory that for any sets A, B, the function spaces $(A \times B) \to C$ and $A \to (B \to C)$ are isomorphic[3].

[2] By *pure* we here mean that variables are the only atomic symbols.
[3] Replacing the first by the second is called *Currying*, after H.B.Curry.

1.1 Normal Order Graph Reduction

At this point we temporarily break from the sequence of language milestones to trace an important implementation issue through to the present.

An implementation of λ-calculus on a sequential machine should use *normal order reduction*, otherwise it may fail to find the normal form of a normalizing term. Consider one reduction step, in applying rule β

$$(\lambda x.A)B \Rightarrow [B/x]A$$

we substitute B into the function body *unreduced*[4]. In general this will produce multiple copies of B, apparently requiring any redexes it contains to be reduced multiple times. For normal order reduction to be practical it is necessary to have an efficient way of handling this.

An alternative policy is to always reduce arguments before substituting into the function body — this is *applicative order reduction*, also known as parameter passing *by value*. Call-by-value is an unsafe reduction strategy for lambda calculus, at least if the measure of correctness is conformity with Church's theory of conversion, but efficient because the actual parameter is reduced only once.

All practical implementations of functional languages for nearly two decades from LISP in 1958 onwards used call-by-value.

The thesis of Wadsworth (1971, Ch 4) showed that the efficiency disadvantage of normal order reduction can be overcome by *normal graph reduction*. In Wadsworth's scheme the λ-term is a directed acyclic graph, and the result of β-reduction, which is performed by update-in-place of the application node, is that a *single copy* of the argument is retained, with pointers to it from each place in the function body where it is referred to. As a consequence any redexes in the argument are reduced at most once.

Turner (1979a) applied normal graph reduction to S, K combinators (Curry 1958) allowing a much simpler abstract machine. In Turner's scheme the graph may be cyclic, permitting a more compact representation of recursion. The combinator code is compiled from a high level functional language using a variant of Curry's abstraction algorithm (Turner 1979b). Initially this was SASL (Turner 1975) and in later incarnations of the system, Miranda (Turner 1986).

For an interpreter, a fixed set of combinators, S, K, C, B, I etc., each with a simple reduction rule, works well. But for compilation to native code on stock hardware it is better to use λ-abstractions derived from the program source as combinators with potentially bigger reduction steps. Extracting these requires a program transformation, *λ-lifting*, (Hughes 1984; Johnsson 1985). This method was used in a compiler for Lazy ML first implemented at Chalmers University in 1984 by Augustsson & Johnsson (1989). Their model for mapping combinator graph reduction onto von Neumann hardware, the G machine, has been refined by Simon Peyton Jones (1992) to the Spineless Tagless G-machine which underlies the Glasgow Haskell compiler, GHC.

[4] The critical case, which shows why normal order reduction is needed, is when B is non-normalizing but A contains no free occurrences of x.

Thus over an extended period of development normal order reduction has been implemented with increasing efficiency.

2 LISP

The first functional programming language and the second oldest programming language still in use (after FORTRAN), LISP began life in 1958 as a project led by John McCarthy at MIT. The aim was to create a system for programming computations over symbolic data, starting with an algorithm McCarthy had drafted for symbolic differentiation. The first published account of the language and theory of LISP is (McCarthy 1960).

The data on which LISP works is the **S-language**. This has a very simple structure, it consists of atoms, which are words like `X` or `TWO` and a pairing operation, written as a dot. Examples of S-expressions are

```
((X.Y).Z)
(ONE.(TWO.(THREE.NIL)))
```

S-expressions can represent lists, trees and so on — they are of variable size and can outlive the procedure that creates them. A far sighted decision by McCarthy was to refuse to clutter his algorithms with storage claim and release instructions. LISP therefore required, and had to invent, heap storage with a garbage collector.

The **M-language** defines computations on S-expressions. It has

(a) S-expressions
(b) function application, written $f[a;b;\ldots]$ with primitive functions *cons, car, cdr,* for creating and decomposing dotted pairs and *atom, eq,* which test for atoms and equality between atoms
(c) conditional expressions written [test$_1$ $\rightarrow$ result$_1$; test $\rightarrow$ result$_2$; . . .]
(d) the ability to define recursive functions — example, here is a function to extract the leftmost atom from an S-expression:

```
first[x] = [atom[x] -> x; T -> first[car[x]]]
```

Note the use of atoms `T, F` as truth values. Function definitions introduce variables, e.g. x, which are lower case to distinguish them from atoms. The values to which the variables become bound are S-expressions. Function names are also lower case but don't get confused with variables because in the M-language function names cannot appear as arguments.

This is computationally complete. McCarthy (1960, Sect. 6) showed that an arbitrary flowchart can be coded as mutually recursive functions.

The M-language of McCarthy (1960) is first order, as there is no provision to pass a function as argument or return a function as result[5].

[5] There has been much confusion about this because McCarthy (1960) uses λ-abstraction — but in a completely different context from (Church 1941).

However, McCarthy shows that M-language expressions and functions can be easily encoded as S-expressions and then defines in the M-language functions, *eval* and *apply*, that correctly interpret these S-expressions.

Thus LISP allows meta-programming, that is treating program as data and vice versa, by appropriate uses of *eval* and *quote*. The 1960 paper gives the impression, quite strongly, that McCarthy saw this as removing any limitation stemming from the M-Language itself being first order.

It was originally intended that people would write programs in the M-language, in an Algol-like notation. In practice LISP programmers wrote their code directly in the S-language form, and the M-language became a kind of ghost that hovered in the background — theoretically important but nobody used it.

In LISP 1.5 (McCarthy et al. 1962) atoms acquired *property lists*, which serve several puposes and numbers appeared, as another kind of atom, along with basic arithmetic functions. This was the first version of LISP to gain a large user community outside MIT and remained in use for many years[6].

Many other versions and dialects of LISP were to follow.

Some Myths About LISP

Something called "Pure LISP" never existed — McCarthy (1978) records that LISP had assignment and **goto** before it had conditional expressions and recursion — it started as a version of FORTRAN I to which these latter were added. LISP 1.5 programmers made frequent use of *setq* which updates a variable and *rplaca*, *rplacd* which update the fields of a CONS cell.

LISP was not based on the lambda calculus, despite using the word "LAMBDA" to denote functions. At the time he invented LISP, McCarthy was aware of (Church 1941) but had not studied it. The theoretical model behind LISP was Kleene's theory of first order recursive functions[7].

The M-language was first order, as already noted, but you could pass a function as a parameter by quotation, i.e. as the S-expression which encodes it. Unfortunately, this gives the wrong binding rules for free variables (dynamic instead of lexicographic).

To represent functions in closed form McCarthy uses $\lambda[[x_1;\ldots;x_n];e]$ and for recursive functions he uses ***label***[*identifier;function*].

However, these functional expressions can occur ONLY IN THE FUNCTION POSITION of an application $f[a;b;\ldots]$. This is clear in the formal syntax for the M-language in the LISP manual (McCarthy at al. 1962, p9).

That is, McCarthy's λ and *label* add no new functions to the M-language, which remains first order. They are introduced solely to allow M-functions to be written in closed form.

[6] When I arrived at St Andrews in 1972 the LISP running on the computer laboratory's IBM 360 was LISP 1.5.

[7] McCarthy made these statements, or very similar ones, in a contribution from the floor at the 1982 ACM symposium on LISP and functional programming in Pittsburgh. No written version of this exists, as far as I know.

If a function has a free variable, e.g y in

$$f = \lambda x.x + y$$

y should be bound to the value in scope for y where f is defined, not where f is called.

McCarthy (1978) reports that this problem (wrong binding for free variables) showed up very early in a program of James Slagle. At first McCarthy assumed it was a bug and expected it to be fixed, but it actually springs from something fundamental — that meta-programming is not the same as higher order programming. Various devices were invented to get round this FUNARG problem, as it became known[8].

Not until SCHEME (Sussman 1975) did versions of LISP with default static binding appear. Today all versions of LISP are lambda calculus based.

3 Algol 60

Algol 60 is not normally thought of as a functional language but its rules for procedures (the Algol equivalent of functions) and variable binding were closely related to those of λ-calculus.

The Revised Report on Algol 60 (Naur 1963) is a model of precise technical writing. It defines the effect of a procedure call by a copying rule with a requirement for systematic change of identifiers where needed to avoid variable capture — exactly like β-reduction.

Although formal parameters could be declared **value** the default parameter passing mode was *call by name*, which required the actual parameter to be copied unevaluated into the procedure body at every occurrence of the formal parameter. This amounts to normal order reduction (but not graph reduction, there is no sharing). The use of call by name allowed an ingenious programming technique: Jensen's Device. See `http://en.wikipedia.org/wiki/Jensen's_Device`

Algol 60 allowed textually nested procedures and passing procedures as parameters (but not returning procedures as results). The requirement in the copying rule for systematic change of identifiers has the effect of enforcing static (that is lexicographic) binding of free variables.

In their book "Algol 60 Implementation", Randell and Russell (1964, Sect. 2.2) handle this by two sets of links between stack frames. The dynamic chain links each stack frame, representing a procedure call, to the frame that called it. The static chain links each stack frame to that of the textually containing procedure, which might be much further away. Free variables are accessed **via the static chain**.

[8] When I started using LISP, at St Andrews in 1972–3, my programs failed in unexpected ways, because I expected λ-calculus like behaviour. Then I read the LISP 1.5 manual carefully — the penny dropped when I looked at the syntax of the M-language (McCarthy et al. 1962, p9) and saw it was first order. This was one of the main reasons for SASL coming into existence.

This mechanism works well for Algol 60 but in a language in which functions can be returned as results, a free variable might be held onto after the function call in which it was created has returned, and will no longer be present on the stack.

Landin (1964) solved this in his SECD machine. A function is represented by a **closure**, consisting of code for the function plus the environment for its free variables. The environment is a linked list of name-value pairs. Closures live in the heap.

4 ISWIM

In early 60's Peter Landin wrote a series of seminal papers on the relationship between programming languages and lambda calculus. This includes (Landin 1964), already noted above, which describes a general mechanism for call-by-value implementation of lambda calculus based languages.

In "The next 700 programming languages", Landin (1966) describes an idealised language family, ISWIM, "If you See What I Mean". The sets of constants and primitive functions and operators of the language are left unspecified. By choosing these you get a language specific to some particular domain. But they all share the same design, which is described in layers.

There is an applicative core, which Landin describes as "Church without lambda". He shows that the expressive power of λ-calculus can be captured by using *where. let, rec* and saying $f(x) = \epsilon$ instead of $f = \lambda\ x\ .\ \epsilon$ and so on. Higher order functions are defined and used without difficulty.

In place of Algol's **begin** ... **end** the offside rule is introduced to allow a more mathematical style of block structure by levels of indentation.

The imperative layer adds mutable variables and assignment.
In a related paper, Landin (1965) defines a control mechanism, the J operator, which allows a program to capture its own continuation, permitting a powerful generalization of labels and jumps. In short,

$$\textit{ISWIM} = \textit{sugared lambda} + \textit{assignment} + \textit{control}$$

The ISWIM paper also has the first appearance of algebraic type definitions used to define structures. This is done in words, but the sum-of-products idea is clearly there.

At end of paper there is an interesting discussion, in which Christopher Strachey introduces the idea of a DL, that is a purely declarative or descriptive language and wonders whether it would be possible to program exclusively in one.

5 PAL

ISWIM inspired PAL (Evans 1968) at MIT and GEDANKEN (Reynolds 1970) at Argonne National Laboratory. These were quite similar. I was given the PAL tape from MIT when I started my PhD studies at Oxford in 1969.

The development of PAL had been strongly influenced by Strachey who was visiting MIT when Art Evans was designing it. The language was intended as a vehicle for teaching programming linguistics, its aims were:

(i) completeness — all data to have the same rights,
(ii) to have a precisely defined semantics (denotational).

There were three concentric layers:

R-PAL: this was an applicative language with sugared λ (let, rec, where) and conditional expressions: $test \rightarrow E_1 \; ! \; E_2$.
One level of pattern matching, e.g. *let x, y, z = expr*
L-PAL: this had everything in R-PAL but adds mutable variables & assignment
J-PAL: adds first class labels and **goto**

PAL was call-by-value and typeless, that is, it had run time type checking. The basic data types were: integer & float numbers, truthvalues, strings — with the usual infixes: $+$ $-$ *etc.* and string handling functions: *stem, stern, conc.*

Data structures were built from tuples, e.g. (a, b, c) these were vectors, not linked lists.

Functions & labels had the same rights as basic data types: to be named, passed as parameters, returned as results, included as tuple elements.

First class labels are very powerful — they allow unusual control structures — coroutines, backtracking and were easier to use than Landin's J operator.

6 SASL

SASL stood for "St Andrews Static Language". I left Oxford in October 1972 for a lectureship at St Andrews and gave a course on programming linguistics in the Autumn term. During that course I introduced a simple DL based on the applicative subset of PAL. This was at first intended only as a blackboard notation but my colleague Tony Davie surprised me by implementing it in LISP over the weekend! So then we had to give it a name.

Later in the academic year I was scheduled to teach a functional programming course to second year undergraduates, hitherto given in LISP. In preparation I started learning LISP 1.5 and was struck by its lack of relationship to λ-calculus, unfriendly syntax and the complications of the *eval/quote* machinery. I decided to teach the course using SASL, as it was now called, instead.

The implementation of SASL in LISP wasn't really robust enough to use for teaching. So over the Easter vacation in 1973 I wrote a compiler from SASL to SECD machine code and an interpreter for the latter, all in BCPL. The code of the first version was just over 300 lines — SASL was not a large language. It ran under the RAX timesharing system on the department's IBM 360/44, so the students were able to run SASL programs interactively on monitors, which they liked.

The language had `let ...in ...` and `rec ...in ...` for non-recursive and recursive definitions. Defining and using factorial looked like this:

```
rec fac n = n < 0 -> 1;
            n * fac (n-1)
in fac 10
```

For mutual recursion you could say `rec` *`def1`* `and` *`def2`* `and ... in ....`

The data types were: integers, truthvalues, characters, lists and functions. All data had same rights — a value of any of the five types could be named, passed to a function as argument, returned as result, or made an element of a list. Following LISP, lists were implemented as linked lists. The elements of a list could be of mixed types, allowing the creation of trees of arbitrary shape.

SASL was implemented using call-by-value, with run time type checking. It had two significant innovations compared with applicative PAL:

(i) strings, "...", were not a separate type, but an abbreviation for lists of characters
(ii) I generalised PAL's pattern matching to allow multi-level patterns, e.g. `let (a,(b,c),d) =` *stuff* `in ...`

SASL was and remained purely applicative. The only method of iteration was recursion — the interpreter recognised tail recursion and implemented it efficiently. The compiler did constant folding — any expression that could be evaluated at compile time, say $(2 + 3)$, was replaced by its value. These two simple optimizations were enough to make the SASL system quite usable, at least for teaching.

As a medium for teaching functional programing, SASL worked better than LISP because:

(a) it was a simple sugared λ-calculus with no imperative features and no eval/quote complications
(b) following Church (1941), function application was denoted by juxtaposition and was left associative, making curried functions easy to define and use
(c) it had correct scope rules for free variables (static binding)
(d) multi-level pattern-matching on list structure made for a big gain in readability. For example the LISP expression[9]

```
cons(cons(car(car(x)),cons(car(car(cdr(x))),nil)),
cons(cons(car(cdr(car(x))),cons(car(cdr(car(cdr(x)))),
nil)),nil))
```

becomes in SASL

```
let ((a,b),(c,d)) = x in ((a,c),(b,d))
```

SASL proved popular with St Andrews students and, after I gave a talk at the summer meeting of IUCC, "Inter-Universities Computing Colloqium", in Swansea in 1975, other universities began to show interest.

[9] The example is slightly unfair in that LISP 1.5 had library functions for frequently occurring compositions of *car* and *cdr*, with names like $caar(x)$ for $car(car(x))$ and $cadr(x)$ for $car(cdr(x))$. With these conventions our example could be written $cons(cons(caar(x), cons(caadr(x), nil)), cons(cons(cadar(x), cons(cadadr(x), nil)), nil))$ However this is still less transparent than the pattern matching version.

6.1 Evolution of SASL 1974–84

The language continued in use at St Andrews for teaching fp and evolved as I experimented with the syntax. Early versions of SASL had an explicit λ, written `lambda`, so you could write e.g. $f = \lambda\ x\ .\ \mathit{stuff}$ as an alternative to $f\ x = \mathit{stuff}$. After a while I dropped λ, allowing only the sugared form. Another simplification was dropping `rec`, making `let` definitions recursive by default. SASL's list notation acquired right associative infixes, *":"* and *"++"*, for *cons* and *append*.

In 1976 SASL syntax underwent two major changes:

(i) a switch from `let` *defs* `in` *exp* to *exp* `where` *defs* with an offside rule to indicate nested scope by indentation.
(ii) allowing multi-equation function definitions, thus extending the use of pattern matching to case analysis. Examples:

```
length () = 0
length (a:x) = 1 + length x

ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

I got this idea from John Darlington[10] (see Section 7 below). The second example above is Ackermann's function.

At the same time SASL underwent a major change of semantics, becoming *lazy*. For the implemention at St Andrews in 1976 I used a lazy version of Landin's SECD machine, following (Burge 1975), who calls it a "procrastinating machine".

On moving to the University of Kent in January 1977 I had two terms with very little teaching which I used to try out an idea I had been mulling over for several years — to apply the normal graph reduction of Wadsworth (1971) to SK combinators. I reimplemented lazy SASL by translation to SK combinators and combinator graph reduction (Turner 1979a, 1979b).

SASL continued to evolve gently, acquiring floating point numbers and list comprehensions[11] in 1983. The latter were inspired by Darlington's "set expressions" (see Section 7 below), but applied to lazy lists.

Burroughs Corporation adopted SASL for a major functional programming project in Austin, Texas, running from 1979 to 1986, to which I was a consultant. The team designed a hardware combinator reduction machine, NORMA (Scheevel 1984), of which two were built in TTL logic. NORMA's software, including compiler, operating system and verification tools was written in SASL.

By the mid-1980's lazy SASL had spread to a significant number of sites, see Table 1. There were three implementations altogether — the version at

[10] I didn't follow Darlington in using $(n + k)$ patterns because SASL's number type was integer rather than natural. I did later (1986) put *(n+k)* patterns into Miranda because Richard Bird wanted them for a book he was writing with Phil Wadler.

[11] I initially called these *ZF expressions*, a reference to Zermelo-Frankel set theory — it was Phil Wadler who coined the better term *list comprehension*.

St Andrews, using a lazy SECD machine, which was rewritten and improved by Bill Cambell; my SK combinator version; and the implementation running on NORMA at Burroughs Austin Research Centre.

6.2 Advantages of Laziness

Two other projects independently developed lazy functional programming systems in the same year as SASL — Friedman & Wise (1976), Henderson & Morris (1976). Clearly laziness was an idea whose time had arrived.

My motives in changing to a lazy semantics in 1976 were

(i) on a sequential machine, consistency with the theory of (Church 1941) requires normal order reduction
(ii) a non-strict semantics is better for equational reasoning
(iii) allows interactive I/O via lazy lists — call-by-value SASL was limited to outputs that could be held in memory before printing.
(iv) I was coming round to the view that lazy data structures could replace exotic control structures, like those of J-PAL.
 (a) lazy lists replace coroutines (e.g. equal fringe problem)
 (b) the *list of successes*[12] method replaces backtracking

6.3 Dynamic Typing

Languages for computation over symbolic data, such as LISP, POP2 and SASL, worked on lists, trees and graphs. This leads to a need for structural polymorphism — a function which reverses a list, or traverses a tree, doesn't need to know the type of the elements. Before the polymorphic type system of Milner (1978), the only convenient way to avoid specifying the type of the elements of a structure was to delay type checking until run time. SASL was dynamically typed for this reason.

But languages with dynamic typing also have a flexibility which is hard to emulate with a static type system. This example comes from SASL's 1976 manual. Let f be a curried function of some unknown number of Boolean arguments — we want to test if f is a tautology (the predicate *logical* tests if its argument is a truthvalue):

```
taut f = logical f -> f;
         taut (f True) & taut (f False)
```

There are still active user communities of languages with run time typing including LISP, which is far from disappearing and, a rising newcomer, Erlang (Cesarini & Thompson 2009).

[12] An example of list of successes method — for the 8 queens problem — is in the 1976 SASL manual, but the method didn't have a name until (Wadler 1985).

Table 1. Lazy SASL sites, circa 1986

California Institute of Technology, Pasadena
City University, London
Clemson University, South Carolina
Iowa State U. of Science & Technology
St Andrews University, UK
Texas A & M University
Université de Montréal, Canada
University College London
University of Adelaide, Australia
University of British Columbia, Canada
University of Colorado at Denver
University of Edinburgh, UK
University of Essex, UK
University of Groningen, Netherlands
University of Kent, UK
University of Nijmegen, Netherlands
University of Oregon, Eugene
University of Puerto Rico
University of Texas at Austin
University of Ulster, Coleraine
University of Warwick, UK
University of Western Ontario, Canada
University of Wisconsin-Milwaukee
University of Wollongong, Australia
Burroughs Corporation, Austin, Texas
MCC Corporation, Austin, Texas
Systems Development Corporation, PA
(24 educational + 3 commercial)

7 Developments in Edinburgh, 1969–1980

Burstall (1969), in an important early paper on structural induction, extended ISWIM with algebraic type definitions — still defined in words — and *case expressions* to analyse data structure.

John Darlington's NPL, "New Programming Language", developed with Burstall in the period 1973-5, replaced case expressions with multi-equation function definitions over algebraic types, including natural numbers, e.g.

```
fib (0)   <=  1
fib (1)   <=  1
fib (n+2) <=  fib (n+1) + fib (n)
```

Darlington got this idea from Kleene's recursion equations.

NPL was implemented in POP2 by Burstall and used for Darlington's work on program transformation (Burstall & Darlington 1977). The language was first order, strongly (but not polymorphically) typed, purely functional, call-by-value. It also had "set expressions" e.g.

```
        setofeven (X)  <=  <:x : x in X & even(x):>
```

NPL evolved into HOPE (Burstall, MacQueen & Sannella, 1980), this was higher order, strongly typed with explicit types and polymorphic type variables, purely functional. It kept multi-equation pattern matching but dropped set expressions.

Also in Edinburgh during 1973-78 the programming language ML emerged as the meta-language of Edinburgh LCF (Gordon et al 1979) a programmable verification system for Scott's logic for computable functions, PPLAMBDA.

This early version of ML had

λ, *let* and *letrec*
references and assignment
types built using $+$, $\times$ and type recursion.
type abstraction
polymorphic strong typing with *type inference* (NB!)
used `* ** ***` *etc.* as an alphabet of type variables

The language was higher order, call-by-value and allowed assignment and mutable data. It lacked pattern matching — structures were analysed by conditionals, tests e.g. *isl, isr* and projection functions.

Standard ML (Milner et al. 1990), which appeared later, in 1986, is the confluence of the HOPE and ML streams, thus has both pattern matching and type inference, but is not pure — it has references and exceptions.

8 Miranda

Developed in 1983-86, Miranda is essentially SASL plus algebraic types and the polymorphic type discipline of Milner (1978). It retains SASL's policies of no explicit λ's and, optional, use of an offside rule to allow nested program structure by indentation (both ideas derived from Landin's ISWIM). The syntax chosen for algebraic type definitions resembles BNF:

```
        tree * ::= Leaf * | Node (tree *) (tree *)
```

The use of `* ** ***` ... as type variables followed the original ML (Gordon et al. 1979) — standard ML had not yet appeared when I was designing Miranda.

For type specifications I used "`::`" because following SASL ":" was retained as infix *cons*.

A lexical distinction between variables and constructors was introduced to distinguish pattern matching from function definition. The decision is made on the initial letter of an identifier — upper case for constructors, lower case for variables. Thus

```
        Node x y = stuff
```

is a pattern match, binding x, y to a, b if *stuff* $=$ *Node* a b whereas

```
node x y = stuff
```

defines a function, *node*, of two arguments.

Miranda is lazy, purely functional, has list comprehensions, polymorphic with type inference and optional type specifications — see Turner (1986) for fuller description — papers and downloads at `www.miranda.org.uk`

An important change from SASL — Miranda had, instead of conditional expressions, conditional equations with guards. Example:

```
sign x  = 1,    if x>0
        = -1,   if x<0
        = 0,    if x=0
```

Combining pattern matching with guards gives a significant gain in expressive power. Guards of this kind first appeared in KRC, "Kent Recursive Calculator" (Turner 1981, 1982), a miniaturised version of SASL which I designed in 1980–81 for teaching. Very simple, KRC had only top level equations (no *where*) with pattern matching and guards; and a built in line editor — a functional alternative to BASIC. KRC was also the testbed for list comprehensions, from where they made their way into SASL and then Miranda.

Putting *where* into a language with guards raised a puzzle about scope rules, forcing a rethink of part of the ISWIM tradition. The solution is that a *where*-clause now governs a whole rhs, including guards, rather than an expression. That is *where* becomes part of definition syntax, instead of being part of expression syntax (a decision that is retained in Haskell).

Miranda was a product of Research Software Ltd, with an initial release in 1985, and subsequent releases in 1987 and 1989. It was quite widely taken up with over 200 universities and 40 companies taking out licenses.

Miranda was by no means the only project combining Milner's polymorphic type system with a lazy, purely functional language in this period (mid 1980's).

Lazy ML, first implemented at Chalmers in 1984 was, as the name suggests, a pure, lazy version of ML, used by Lennart Augustsson and Thomas Johnsson as both source and implementation language for their work on compiled graph reduction which we referred to in Section 1.1, see (Augustsson 1984; Augustsson & Johnsson 1989).

At Oxford, Philip Wadler developed Orwell, a simple equational language for teaching functional programming, along much the same lines as Miranda. Orwell and Miranda were able to share a text book. Bird & Wadler (1988) used a mathematical notation for functional programming — e.g. Greek letters for type variables — that could be used with either Miranda or Orwell (or indeed other functional languages).

Clean, a lazy language based on graph reduction, with *uniqueness types* to handle I/O and mutable state. was developed at Nijmegen from 1987 by Rinus Plasmeijer and his colleagues (Plameijer & van Eekelen 1993).

9 Haskell

Designed by a committee which started work in 1987, version 1.2 of the Haskell Report was published in SIGPLAN Notices (Hudak et al. 1992). The language continued to evolve, reaching a declared standard for long term support in Haskell 98 (Peyton Jones 2003).

Similar in many ways to Miranda, being lazy, higher order, polymorphically typed with algebraic types, pattern matching and list comprehensions, the most noticeable syntactic differences are:

Switched guards to left hand side of equations

```
sign x  | x > 0 =  1
        | x < 0 = -1
        | x==0  =  0
```

Change of syntax for type declarations — Miranda

```
bool ::= True | False
string == [char]
```

becomes in Haskell

```
data Bool = False | True
type String = [Char]
```

Extension of Miranda's var/constructor distinction by initial letter to types, giving lower case type variables, upper case type consts — Miranda

```
map ::  (*->**)->[*]->[**]
filter :: (*->bool)->[*]->[*]
zip3 :: [*]->[**]->[***]->[(*,**,***)]
```

becomes in Haskell

```
map :: (a->b)->[a]->[b]
filter :: (a->Bool)->[a]->[a]
zip3 :: [a]->[b]->[c]->[(a,b,c)]
```

Haskell has a richer and more redundant syntax. e.g. it provides conditional expressions and guards, *let*-expressions and *where*-clauses, *case*-expressions and pattern matching by equations, λ-expressions and *function-form = rhs* etc. . . .

Almost everything in Miranda is also present in Haskell in one form or another, but not vice versa — Haskell includes a system of type classes, monadic I/O and a module system with two level names. Of these the first is particularly noteworthy and the most innovative feature of the language. An account of the historical relationship between Haskell and Miranda can be found in (Hudak et al. 2007, s2.3 and s3.8).

```
class Taut a where
  taut :: a->Bool

instance Taut Bool where
  taut b = b

instance Taut a => Taut (Bool->a) where    -- problem here
  taut f = taut (f True) && taut (f False)
```

Fig. 1. Class *Taut*

This is not the place for a detailed account of Haskell, for which many excellent books and tutorials exist, but I would like to close the section with a simple example of what can be done with type classes. Let us try to recover the variadic tautology checker of Section 6.3.

Figure 1 introduces a class $Taut$ with two instances to cover the two cases. Unfortunately the second instance declaration is illegal in both Haskell 98 and the current language standard, Haskell 2010. Instance types are required to be generic, that is of the form $(Ta_1 \ldots a_n)$. So $(a \to b)$ is allowed as an instance type but not $(Bool \to a)$.

```
class Boolean b where
  fromBool :: Bool->b

instance Boolean Bool where
  fromBool t = t

class Taut a where
  taut :: a->Bool

instance Taut Bool where
  taut b = b

instance (Boolean a, Taut b) => Taut (a->b) where
  taut f = taut (f (fromBool True))
           && taut (f (fromBool False))
```

Fig. 2. Variadic *taut* in Haskell 2010

GHC, the Glasgow Haskell compiler, supports numerous language extensions and will accept the code in Figure 1 if language extension $FlexibleInstances$ is enabled. To make the example work in standard Haskell is more trouble; we have to introduce an auxilliary type class, see Figure 2. One could not claim that this is as simple and transparent as the SASL code shown earlier.

When what was to become the Haskell committee had its first scheduled meeting in January 1988 a list of goals for the language, which did not yet have a name, was drawn up, see (Hudak et al. 2007, s2.4), including

It should be based on ideas which enjoy a wide consensus.

Type classes cannot be said to fall under that conservative principle: they are an experimental feature, and one that pervades the language. They are surprisingly powerful and have proved extremely fertile but also add greatly to the complexity of Haskell, especially the type system.

Acknowledgements. I am grateful to the programme committee of TFP 2012 for inviting me to give a lecture on the history of functional programming at TFP 2012 in St Andrews. This written version of the lecture has benefitted from comments and suggestions I received from other participants in response to the lecture. In particular I am indebted to Josef Svenningsson of Chalmers University for showing me how to code the variadic *taut* function in Haskell using type classes.

Section 1 draws on material from my essay on *Church's Thesis and Functional Programming* in (Olszewski et al. 2006, 518-544).

References

Augustsson, L.: A compiler for Lazy ML. In: Proceedings 1984 ACM Symposium on LISP and Functional Programming, pp. 218–227. ACM (1984)

Augustsson, L., Johnsson, T.: The Chalmers Lazy-ML Compiler. The Computer Journal 32(2), 127–141 (1989)

Bird, R.S., Wadler, P.: Introduction to Functional Programming, 293 pages. Prentice Hall (1988)

Burge, W.: Recursive Programming Techniques, 277 pages. Addison Wesley (1975)

Burstall, R.M.: Proving properties of programs by structural induction. Computer Journal 12(1), 41–48 (1969)

Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. JACM 24(1), 44–67 (1977); Revised and extended version of paper originally presented at Conference on Reliable Software, Los Angeles (1975)

Burstall, R.M., MacQueen, D., Sanella, D.T.: HOPE: An experimental applicative language. In: Proceedings 1980 LISP Conference, Stanford, California, pp. 136–143 (August 1980)

Cesarini, F., Thompson, S.: Erlang Programming, 498 pages. O'Reilly (June 2009)

Church, A., Rosser, J.B.: Some Properties of conversion. Transactions of the American Mathematical Society 39, 472–482 (1936)

Church, A.: The calculi of lambda conversion. Princeton University Press (1941)

Curry, H.B., Feys, R.: Combinatory Logic, vol. I. North-Holland, Amsterdam (1958)

Evans, A.: PAL - a language designed for teaching programming linguistics. In: Proceedings ACM National Conference (1968)

Friedman, D.P., Wise, D.S.: CONS should not evaluate its arguments. In: Proceedings 3rd Intl. Coll. on Automata Languages and Programming, pp. 256–284. Edinburgh University Press (1976)

Gordon, M., Wadsworth, C.P., Milner, R.: Edinburgh LCF. LNCS, vol. 78. Springer (1979)

Henderson, P., Morris, J.M.: A lazy evaluator. In: Proceedings 3rd POPL Conference, Atlanta, Georgia (1976)

Hindley, J.R., Seldin, J.P.: Lambda-Calculus and Combinators: An Introduction, 2nd edn., 360 pages. Cambridge University Press (August 2008)

Hudak, P., et al.: Report on the Programming Language Haskell. SIGPLAN Notices 27(5), 164 pages (1992)

Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A History of Haskell: Being Lazy with Class. In: Proceedings 3rd ACM SIGPLAN History of Programming Languages Conference, San Diego, California, pp. 1–55 (June 2007)

Hughes, J.: The Design and Implementation of Programming Languages, Oxford University D. Phil. Thesis (1983); (Published by Oxford University Computing Laboratory Programming Research Group, as Technical Monograph PRG 40 (September 1984)

Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, Springer, Heidelberg (1985)

Landin, P.J.: The Mechanical Evaluation of Expressions. Computer Journal 6(4), 308–320 (1964)

Landin, P.J.: A generalization of jumps and labels. Report, UNIVAC Systems Programming Research (August 1965); Reprinted in Higher Order and Symbolic Computation 11(2), 125–143 (1998)

Landin, P.J.: The Next 700 Programming Languages. CACM 9(3), 157–165 (1966)

McCarthy, J.: Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. CACM 3(4), 184–195 (1960)

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I.: LISP 1.5 Programmer's Manual, 106 pages. MIT Press (1962); 2nd edn. 15th printing (1985)

McCarthy, J.: History of LISP. In: Proceedings ACM Conference on History of Programming Languages I, pp. 173–185 (June 1978), `www-formal.stanford.edu/jmc/history/lisp/lisp.html`

Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17(3), 348–375 (1978)

Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML. MIT Press (1990) (revised 1997)

Naur, N.: Revised Report on the Algorithmic Language Algol 60. CACM 6(1), 1–17 (1963)

Olszewski, A., et al. (eds.): Church's Thesis after 70 years, 551 pages. Ontos Verlag, Berlin (2006)

Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. Journal of Functional Programming 2(2), 127–202 (1992)

Peyton Jones, S.L.: Haskell 98 language and libraries: the Revised Report. JFP 13(1) (January 2003)

Plasmeijer, R., Van Eekelen, M.: Functional Programming and Parallel Graph Rewriting, 571 pages. Addison-Wesley (1993)

Randell, B., Russell, L.J.: The Implementation of Algol 60. Academic Press (1964)

Reynolds, J.C.: GEDANKEN a simple typeless language based on the principle of completeness and the reference concept. CACM 13(5), 308–319 (1970)

Scheevel, M.: NORMA: a graph reduction processor. In: Proceedings ACM Conference on LISP and Functional Programming, pp. 212–219 (August 1986)

Sussman, G.J., Steele Jr., G.L.: Scheme: An interpreter for extended lambda calculus, MEMO 349, MIT AI LAB (1975)

Turner, D.A.: SASL Language Manual, St. Andrews University, Department of Computational Science Technical Report CS/75/1 (January 1975); revised December 1976; revised at University of Kent, August 1979, November 1983

Turner, D.A.: A New Implementation Technique for Applicative Languages. Software-Practice and Experience 9(1), 31–49 (1979a)

Turner, D.A.: Another Algorithm for Bracket Abstraction. Journal of Symbolic Logic 44(2), 267–270 (1979b)

Turner, D.A.: The Semantic Elegance of Applicative Languages. In: Proceedings MIT/ACM conference on Functional Languages and Architectures, Portsmouth, New Hampshire, pp. 85–92 (October 1981)

Turner, D.A.: Recursion Equations as a Programming Language. In: Darlington, Henderson, Turner (eds.) Functional Programming and its Applications, pp. 1–28. Cambridge University Press (January 1982)

Turner, D.A.: An Overview of Miranda. SIGPLAN Notices 21(12), 158–166 (1986)

Wadler, P.: Replacing a failure by a list of successes. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985)

Wadsworth, C.P.: The Semantics and Pragmatics of the Lambda Calculus. D.Phil. Thesis, Oxford University Programming Research Group (1971)

Combining Deep and Shallow Embedding for EDSL

Josef Svenningsson and Emil Axelsson

Chalmers University of Technology
{josefs,emax}@chalmers.se

Abstract. When compiling embedded languages it is natural to use an abstract syntax tree to represent programs. This is known as a *deep* embedding and it is a rather cumbersome technique compared to other forms of embedding, typically leading to more code and being harder to extend. In *shallow* embeddings, language constructs are mapped directly to their semantics which yields more flexible and succinct implementations. But shallow embeddings are not well-suited for compiling embedded languages. We present a technique to combine deep and shallow embedding in the context of compiling embedded languages in order to provide the benefits of both techniques. In particular it helps keeping the deep embedding small and it makes extending the embedded language much easier. Our technique also has some unexpected but welcome knock-on effects. It provides fusion of functions to remove intermediate results for free without any additional effort. It also helps to give the embedded language a more natural programming interface.

1 Introduction

When compiling an embedded language it is natural to use an algebraic data type to represent the abstract syntax tree (AST). This is known as a *deep* embedding. Deep embeddings can be cumbersome: the AST can grow quite large in order to represent all the language features, which can make it rather unwieldy to work with. It is also laborious to add new language constructs as it requires changes to the AST as well as all functions manipulating the AST.

In contrast, *shallow* embeddings don't require an abstract syntax tree and all the problems that come with it. Instead, language constructs are mapped directly to their semantics. But if we wish to compile our embedded language we have little choice but having some form of AST — in particular if we not only want to compile it, but first transform the representation, or if we have another type of backend, say, a verification framework.

In this paper we present a technique for combining deep and shallow embeddings in order to achieve many of the advantages of both styles. This combination turns out to provide knock-on effects which we also explore. In particular, our technique has the following advantages:

Simplicity: By moving functionality to shallow embeddings, our technique helps keep the AST small without sacrificing expressiveness.

Abstraction: The shallow embeddings are based on *abstract data types* leading to better programming interfaces (more like ordinary APIs than constructs of a language). This has important side-effects:

– The shallow interfaces can have properties not possessed by the deep embedding. For example, our vector interface (Section 4.5) guarantees removal of intermediate structures (see Section 5).
– The abstract types can sometimes be made instances of standard Haskell type classes, such as Functor and Monad, even when the deep embedding cannot (demonstrated in Section 4.4 and 4.5).

Extensibility: Our technique can be seen as a partial solution to the expression problem[1] as it makes it easier to extend the embedded language with new language constructs and functions.

Before giving an overview of our technique (Section 3) we will give an introduction to shallow and deep embeddings in Section 2, including a comparison of the two methods (Section 2.1).

Throughout this paper we will use Haskell [16] and some of the extensions provided by the Glasgow Haskell Compiler. While we will use many Haskell-specific functions and constructs the general technique and its advantages translates readily to other languages.

2 Shallow and Deep — Pros and Cons

To explain the meaning of "deep" and "shallow" we will use the following small embedded domain specific language (EDSL) from [5] as an illustrating example.

```
inRegion :: Point  → Region → Bool
circle   :: Radius → Region
outside  :: Region → Region
(∩)      :: Region → Region → Region
(∪)      :: Region → Region → Region
```

This piece of code defines a small language for regions, i.e. two-dimensional areas. It only shows the interface; we will give two implementations, one deep and one shallow.

The type Region defines the type of regions which is the domain we are concerned with in this example. We can interpret regions by using inRegion, which allows us to check whether a point is within a region or not. We will refer to functions such as inRegion which interpret values in our domain as *interpretation functions*. The function inRegion takes an argument of type Point and we will just assume there is such a type together with the expected operations on points.

Regions can be constructed using circle which creates a region with a given radius (again, we assume a type Radius without giving its definition). The functions outside, (∩) and (∪) take the complement, intersection and union of regions. As an example of how to use the language, we define the function annulus which can be used to construct donut-like regions given two radii:

[1] http://www.daimi.au.dk/~madst/tool/papers/expression.txt

```
annulus :: Radius → Radius → Region
annulus r1 r2 = outside (circle r1) ∩ (circle r2)
```

The first implementation of our small region EDSL will use a *shallow* embedding. The code is shown below.

```
type Region = Point → Bool

p ‘inRegion‘ r = r p
circle  r      = λp → magnitude p ≤ r
outside r      = λp → not (r p)
r1 ∩ r2        = λp → r1 p && r2 p
r1 ∪ r2        = λp → r1 p || r2 p
```

Our concrete implementation of the type Region is the type Point → Bool. We will refer to the type Point → Bool as the *semantic domain* of the shallow embedding. It is no coincidence that the semantic domain is similar to the type of the function inRegion. The essence of shallow embeddings is that the representation they use directly encode the operations that can be performed on them. In our case Region is represented exactly as a test whether a Point is within the region or not.

The implementation of the function inRegion becomes trivial; it simply uses the function used to represent regions. This is common for shallow embeddings; interpretation functions like inRegion, can make direct use of the operations used in the representation. All the other functions encode what it means for a point to be inside the respective region.

The characteristic of deep embeddings is that they use an abstract syntax tree to represent the domain. Below is how we would represent our example language using a deep embedding.

```
data Region = Circle Radius   | Intersect Region Region
            | Outside Region  | Union     Region Region

circle  r = Circle    r
outside r = Outside   r
r1 ∩ r2   = Intersect r1 r2
r1 ∪ r2   = Union     r1 r2

p ‘inRegion‘ (Circle  r)        = magnitude p ≤ r
p ‘inRegion‘ (Outside r)        = not (p ‘inRegion‘ r)
p ‘inRegion‘ (Intersect r1 r2)  = p ‘inRegion‘ r1 && p ‘inRegion‘ r2
p ‘inRegion‘ (Union     r1 r2)  = p ‘inRegion‘ r1 || p ‘inRegion‘ r2
```

The type Region is here represented as a data type with one constructor for each function that can be used to construct regions.

Writing the functions for constructing new regions becomes trivial. It is simply a matter of returning the right constructor. The hard work is instead done in the interpretation function inRegion which has to interpret the meaning of each constructor.

2.1 Brief Comparison

As the above example EDSL illustrates, a shallow embedding makes it easier to add new language constructs — as long as they can be represented in the semantic domain. For instance, it would be easy to add a function `rectangle` to our region example. On the other hand, since the semantic domain is fixed, adding a different form of interpretation, say, computing the area of a region, would not be possible without a complete reimplementation.

In the deep embedding, we can easily add new interpretations (just add a new function like `inRegion`), but this comes at the price of having a fixed set of language constructs. Adding a new construct to the deep implementation requires updating the `Region` type as well as all existing interpretation functions.

This comparison shows that shallow and deep embeddings are dual in the sense that the former is extensible with regards to adding language constructs while the latter is extensible with regards to adding interpretations. The holy grail of embedded language implementation is to be able to combine the advantages of shallow and deep in a single implementation. This is commonly referred to as the *expression problem*.

One way to work around the limitation of deep embeddings not being extensible is to use "derived constructs". An example of a derived construct is `annulus`, which we defined in terms of `outside`, `circle` and (∩). Derived constructs are shallow in the sense that they do not have a direct correspondence in the underlying embedding. Shallow derived constructs of a deep embedding are particularly interesting as they *inherit most advantages of both shallow and deep embeddings.* They can be added with the same ease as constructs in a fully shallow embedding. Yet, the interpretation functions only need to be aware of the deep constructs, which means that we retain the freedom of interpretation available in deep embeddings. There are, of course, limitations to how far these advantages can be stretched. We will return to this point in the concluding discussion (Section 6).

The use of shallow derived constructs is quite common in deeply embedded DSLs. The technique presented in this paper goes beyond "simple" derived constructs to extensions with new interface types leading to drastically different interfaces.

3 Overview of the Technique

We assume a setting where we want an EDSL which generates code. Code generation tends to require intensional analysis of the AST, which is not directly possible with a shallow implementation (but see Reference [4] for how to generate code in the final tagless style). Hence, we need a deep embedding as a basis. Our technique can be summarized in the following steps:

1. Implement a deeply embedded core language. The aim of the core language is *not* to act as a convenient user interface, but rather to support efficient generation of common code patterns in the target language. For this reason, the core language should be kept as simple as possible.
2. Implement user-friendly interfaces as shallow embeddings on top of the core language. Each interface is represented by a separate type and operations on this type.

3. Give each interface a precise meaning by giving a translation to and from a corresponding core language program. In other words, make the deep embedding the semantic domain of the shallow embedding. This is done by means of type class instantiation. If such a translation is not possible, or not efficient, extend the core language as necessary.

These ideas have been partly described in our paper on the implementation of the Feldspar EDSL [1]. However, we feel that the ideas are important enough to be presented as a general technique, not tied to a particular language implementation.

In the sections that follow we will demonstrate our technique through a series of examples. For the sake of concreteness we have made some superficial choices which are orthogonal to our technique. In particular, we use a typed embedded language and employ higher order abstract syntax to deal with binding constructs. Neither of these choices matter for the applicability of our technique.

4 Examples

To demonstrate our technique we will use a small embedded language called FunC as our running example. The data type describing the FunC abstract syntax tree can be seen below.

```
data FunC a where
  LitI     :: Int  → FunC Int
  LitB     :: Bool → FunC Bool
  If       :: FunC Bool → FunC a → FunC a → FunC a
  While    :: (FunC s → FunC Bool) → (FunC s → FunC s) → FunC s → FunC s
  Pair     :: FunC a → FunC b → FunC (a,b)
  Fst      :: FunC (a,b) → FunC a
  Snd      :: FunC (a,b) → FunC b
  Prim1    :: String → (a → b) → FunC a → FunC b
  Prim2    :: String → (a → b → c) → FunC a → FunC b → FunC c
  Value    :: a → FunC a
  Variable :: String → FunC a
```

FunC is a low level, pure functional language which has a straightforward translation to C. It is meant for embedding low level programs and is inspired by the Core language used in the language Feldspar [2]. We use a GADT to give precise types to the different constructors. We have also chosen Higher Order Abstract Syntax [17] to represent constructs with variable binding. In the above data type, the only higher-order construct is While. We will add another one in Section 4.5.

FunC has constructs for integer and boolean literals and an if-expression for testing booleans. The while expression models while loops. Since FunC is pure, the body of the loop cannot perform side-effects, so instead the while loop passes around a state. The third argument to the While constructor is the initial value of the state. The state is then updated each iteration of the loop by the second argument. In order to determine when to stop looping the first argument is used, which performs a test on the state. Furthermore, FunC has pairs which are constructed with the Pair constructor and eliminated using Fst and Snd. The constructs Prim1 and Prim2 are used to create primitive functions in FunC. The string argument is the name of the primitive function which is used when generating code from FunC and the function argument is used during evaluation. It is possible to simply have a single constructor for primitive functions of an arbitrary number of arguments but that would complicate the presentation unnecessarily for the purpose of this paper. The two last constructors, Value and Variable , are not part of the language. They are used internally for evaluation and printing respectively.

The exact semantics of the FunC language is given by the eval function.

```
eval :: FunC a → a
eval (LitI i)         = i
eval (LitB b)         = b
eval (While c b i)    = head $ dropWhile (eval ∘ c ∘ Value) $
                        iterate (eval ∘ b ∘ Value) $ eval i
eval (If c t e)       = if eval c then eval t else eval e
eval (Pair a b)       = (eval a, eval b)
eval (Fst p)          = fst (eval p)
eval (Snd p)          = snd (eval p)
eval (Prim1 _ f a)    = f (eval a)
eval (Prim2 _ f a b)  = f (eval a) (eval b)
eval (Value a)        = a
```

4.1 The Syntactic Class

So far our presentation of FunC has been a purely deep embedding. Our goal is to be able to add shallow embeddings on top of the deep embedding and in order to make that possible we will make our language extensible using a type class. This type class will encompass all the types that can be compiled into our FunC language. We call the type class Syntactic .

```
class Syntactic a where
  type Internal a
  toFunC   :: a → FunC (Internal a)
  fromFunC :: FunC (Internal a) → a
```

When making an instance of the class Syntactic for a type T one must specify how T will represented internally, in the already existing deep embedding of FunC. This is what the associated type Internal is for. The two functions toFunC and fromFunC translates back and forth between the type T and its internal representation. The fromFunC method is needed when defining user interfaces based on the Syntactic class. The first instance of Syntactic is simply FunC itself, and the instance is completely straightforward.

```
instance Syntactic (FunC a) where
    type Internal (FunC a) = a
    toFunC   ast = ast
    fromFunC ast = ast
```

4.2 User Interface

Now that we have the Syntactic class we can give FunC a nice extensible interface which we can present to the programmer using FunC. This interface will mirror the deep embedding and its constructor but will use the class Syntactic to overload the functions to make them compatible with any type that we choose to make an instance of Syntactic .

```
true, false :: FunC Bool
true         = LitB True
false        = LitB False

ifC :: Syntactic a ⇒ FunC Bool → a → a → a
ifC c t e = fromFunC (If c (toFunC t) (toFunC e))

c ? (t,e) = ifC c t e

while :: Syntactic s ⇒ (s → FunC Bool) → (s → s) → s → s
while c b i = fromFunC (While (c ∘ fromFunC)
                              (toFunC ∘ b ∘ fromFunC)
                              (toFunC i))
```

When specifying the types in our new interface we note that base types are not overloaded, they are still on the form FunC Bool. The big difference is when we have polymorphic functions. The function ifC works for any a as long as it is an instance of Syntactic . The advantage of the type Syntactic a ⇒ FunC Bool → a → a → a over FunC Bool → FunC a → FunC a → FunC a is two-fold: First, it is closer to the type that an ordinary Haskell function would have and so it gives the function a more native feel, like it is less of a library and more of a language. Secondly, it makes the language extensible. These functions can now be used with any type that is an instance of Syntactic . We are no longer tied to working solely on the abstract syntax tree FunC.

We have not shown any interface for integers. One way to implement that would be to provide a function equivalent to the LitI constructor. In Haskell there is a nicer way: provide an instance of the type class Num. By instantiating Num we get access to Haskell's overloaded syntax for numeric literals so that we don't have to use a function for lifting numbers into FunC. Additionally, Num contains arithmetic functions which we also gain access to. Similarly, we instantiate the Integral class to get an interface for integral operations. The primive functions of said type classes are implemented using the constructors Prim1 and Prim2. We refrain from presenting the code as it is rather Haskell-specific and unrelated to the main point of the paper.

We will also be using comparison operators in FunC. For tiresome reasons it is not possible to overload the methods of the corresponding type classes Eq and Ord: these methods return a Haskell Bool and there is no way we can change that to fit the types of FunC. Instead we will simply assume that the standard definitions of the comparison operators are hidden and we will use definitions specific to FunC.

4.3 Embedding Pairs

We have not yet given an interface for pairs. The reason for this is that they provide an excellent opportunity to demonstrate our technique. We simply instantiate the Syntactic class for Haskell pairs:

```
instance (Syntactic a, Syntactic b) ⇒ Syntactic (a,b) where
    type Internal (a,b) = (Internal a, Internal b)
    toFunC (a,b)        = Pair (toFunC a) (toFunC b)
    fromFunC p          = (fromFunC (Fst p), fromFunC (Snd p))
```

In this instance, toFunC constructs an embedded pair from a Haskell pair, and fromFunC eliminates an embedded pair by selecting the first and second component and returning these as a Haskell pair.[2]

The usefulness of pairs comes in when we need an existing function to operate on a compound value rather than a single value. For example, the state of the while loop is a single value. If we want the state to consist of, say, two integers, we use a pair. Since functions such as ifC and while are overloaded using Syntactic , there is no need for the user to construct compound values explicitly; this is done automatically by the overloaded interface.

As an example of this, here is a for loop defined using the while construct with a compound state:

```
forLoop :: Syntactic s ⇒ FunC Int → s → (FunC Int → s → s) → s
forLoop len init step = snd $ while (λ(i,s) → i<len)
                                    (λ(i,s) → (i+1, step i s))
                                    (0,init)
```

The first argument to forLoop is the number of iterations; the second argument is the initial state; the third argument is the step function which, given the current loop index and current state, computes the next state. We define forLoop using a while loop whose state is a pair of an integer and a smaller state.

Note that the above definition only uses ordinary Haskell pairs: The continue condition and step function of the while loop pattern match on the state using ordinary pair syntax, and the initial state is constructed as a standard Haskell pair.

4.4 Embedding Option

If we want to extend our language with optional values, one may be tempted to make a Syntactic instance for Maybe. Unfortunately, there is no way to make this work, because fromFunC would have to decide whether to return Just or Nothing when the Haskell program is evaluated, which is one stage earlier than when the FunC program is evaluated. Instead, we can use the following implementation:

[2] Note that the argument p is duplicated in the definition of fromFunC. If both components are later used in the program, this means that the syntax tree will contain two copies of p. For this reason, having tuples in the language usually requires some way of recovering sharing [7]. This issue is, however, orthogonal to the ideas presented in this paper.

```
data Option a = Option { isSome :: FunC Bool, fromSome :: a }

instance Syntactic a ⇒ Syntactic (Option a) where
    type Internal (Option a) = (Bool, Internal a)
    fromFunC m               = Option (Fst m) (Snd m)
    toFunC (Option b a)      = Pair b a
```

We have borrowed the name `Option` from ML to avoid clashing with the name of the Haskell type. The type `Option` is represented as a boolean and a value.[3] The boolean indicates whether the value is valid or whether it should simply be ignored, effectively interpreting it as not being there. The `Syntactic` instance converts to and from the representation in `FunC` which is a pair of a boolean and the value.

The definition of `Option` may seem very straight forward but when we try to implement functions for creating values of type `Option` we run into problems. Specifically it is hard to create an empty `Option` value, because we need some value to put into the second component of the pair. FunC is simply not expressive enough to encode this type as it stands. So we will have to extend FunC somehow to accommodate the `Option` type. There are several ways of doing this and we have chosen a very minimal extension. We add a notion of undefined values. To begin with we add an extra constructor to the `FunC` type.

```
    Undef :: FunC a
```

Next we give semantics to `Undef` (`undefined` is part of the Haskell standard) and provide an overloaded function `undef` for convenience.

```
eval Undef = undefined

undef :: Syntactic a ⇒ a
undef = fromFunC Undef
```

Armed with undefined values we can now easily provide functions for constructing optional values:

```
some :: a → Option a
some a = Option true a

none :: Syntactic a ⇒ Option a
none = Option false undef

option :: (Syntactic a, Syntactic b) ⇒ b → (a → b) → Option a → b
option noneCase someCase opt = ifC (isSome opt)
                                   (someCase (fromSome opt))
                                   noneCase
```

The `some` function creates an optional value which actually contains a value whereas `none` defines an empty value. It is the function `none` which uses the `undef` function we previously added to FunC. The function `option` acts as a case on optional values, allowing the programmer to test an `Option` value to see whether is contains something or not.

[3] Larger unions can be encoded using an integer instead of a boolean.

The function above provides a nice programmer interface but the real power of the shallow embedding of the Option type comes from the fact that we can make it an instance of standard Haskell classes. In particular we can make it an instance of Functor and Monad.

```
instance Functor Option where
    fmap f (Option b a) = Option b (f a)

instance Monad Option where
    return a  = some a
    opt >>= k = b { isSome = isSome opt ? (isSome b, false) }
      where b = k (fromSome opt)
```

Being able to reuse standard Haskell functions is a great advantage as it helps to decrease the cognitive load of the programmer when learning our new language. We can map any Haskell function on the element of an optional value because we chose to let the element of the Option type to be completely polymorphic, which is why these instances type check. The advantage of reusing Haskell's standard classes is particularly powerful in the case of the Monad class because it has syntactic support in Haskell which means that it can be reused for our embedded language. For example, suppose that we have a function divO :: FunC Int → FunC Int → Option (FunC Int) which returns nothing in the case the divisor is zero. Then we can write functions such as the following:

```
divTest :: FunC Int → FunC Int → FunC Int → Option (FunC Int)
divTest a b c = do r1 ← divO a b
                   r2 ← divO a c
                   return (r1+r2)
```

4.5 Embedding Vector

Our language FunC is intended to target low level programming. In this domain most programs deal with sequences of data, typically in the form of arrays. In this section we will see how we can extend FunC to provide a nice interface to array programming.

The first thing to note is that FunC doesn't have any support for arrays at the moment. We will therefore have to extend FunC to accommodate this. The addition we have chosen is one constructor which computes an array plus two constructors for accessing the length and indexing into the array respectively:

```
Arr    :: FunC Int → (FunC Int → FunC a) → FunC (Array Int a)
ArrLen :: FunC (Array Int a) → FunC Int
ArrIx  :: FunC (Array Int a) → FunC Int → FunC a
```

The first argument of the Arr constructor computes the length of the array. The second argument is a function which given an index computes the element at that index. By repeatedly calling the function for each index we can construct the whole array this way. The meaning of ArrLen and ArrIx should require little explanation. The exact semantics of these constructors is given by the corresponding clauses in the eval function.

```
eval (Arr l ixf) = listArray (0,lm1) [eval $ ixf $ value i | i ← [0..lm1]]
                     where lm1 = eval l − 1
eval (ArrLen a)  = (1 +) $ uncurry (flip (−)) $ bounds $ eval a
eval (ArrIx a i) = eval a ! eval i
```

We will use two convenience functions for dealing with length and indexing: len which computes the length of the array and the infix operator (<!>) which is used to index into the array. As usual we have overloaded (<!>) so that it can be used with any type in the Syntactic class.

```
len :: FunC (Array Int a) → FunC Int
len arr = ArrLen arr

(<!>) :: Syntactic a ⇒ FunC (Array Int (Internal a)) → FunC Int → a
arr <!> ix = fromFunC (ArrIx arr ix)
```

Having extended our deep embedding to support arrays we are now ready to provide the shallow embedding. In order to avoid confusion between the two embeddings we will refer to the shallow embedding as vector instead of array.

```
data Vector a where
  Indexed :: FunC Int → (FunC Int → a) → Vector a

instance Syntactic a ⇒ Syntactic (Vector a) where
    type Internal (Vector a) = Array Int (Internal a)
    toFunC (Indexed l ixf)   = Arr l (toFunC ∘ ixf)
    fromFunC arr             = Indexed (len arr) (λix → arr <!> ix)
```

The type Vector forms the shallow embedding and its constructor Indexed is strikingly similar to the Arr construct. The only difference is that Indexed is completely polymorphic in the element type. One of the advantages of a polymorphic element type is that we can have any type which is an instance of Syntactic in vectors, not only values which are deeply embedded. Indeed we can even have vectors of vectors which can be used as a simple (although not very efficient) representation of matrices.

The Syntactic instance converts vectors into arrays and back. It is mostly straightforward except that elements of vectors need not be deeply embedded so they must in turn be converted using toFunC.

```
zipWithVec :: (Syntactic a, Syntactic b) ⇒
              (a → b → c) → Vector a → Vector b → Vector c
zipWithVec f (Indexed l1 ixf1) (Indexed l2 ixf2)
  = Indexed (min l1 l2) (λix → f (ixf1 ix) (ixf2 ix))

sumVec :: (Syntactic a, Num a) ⇒ Vector a → a
sumVec (Indexed l ixf) = forLoop l 0 (λix s → s + ixf ix)

instance Functor Vector where
    fmap f (Indexed l ixf) = Indexed l (f ∘ ixf)
```

The above code listing shows some examples of primitive functions for vectors. The call zipWith f v1 v2 combines the two vectors v1 and v2 pointwise using the function f. The sumVec function computes the sum of all the elements of a vector using the for loop defined in Section 4.3. Finally, just as with the Option type in Section 4.4 we can define an instance of the class Functor.

Many more functions can be defined for our Vector type. In particular, any kind of function where each vector element can be computed independently will work particularly well with the representation we have chosen. However, functions that require sharing of previously computed results (e.g. Haskell's unfoldr) will yield poor code.

```
scalarProd :: (Syntax a, Num a) ⇒ Vector a → Vector a → a
scalarProd a b = sumVec (zipWithVec (∗) a b)
```

An example of using the functions presented above we define the function scalarProd which computes the scalar product of two vectors. It works by first multiplying the two vectors pointwise using zipWithVec. The resulting vector is then summed to yield the final answer.

5 Fusion

Choosing to implement vectors as a shallow embedding has a very powerful consequence: it provides a very lightweight implementation of fusion [12]. We will demonstrate this using the function scalarProd defined in the previous section. Upon a first glance it may seem as if this function computes an intermediate vector, the vector zipWithVec (∗) a b which is then consumed by the sumVec. This intermediate vector would be quite bad for performance and space reasons if we ever wanted to use the scalarProd function as defined.

Luckily the intermediate vector is never computed. To see why this is the case consider what happens when we generate code for the expression scalarProd v1 v2, where v1 and v2 are defined as Indexed l1 ixf1 and Indexed l2 ixf2 respectively. Before generating an abstract syntax tree the Haskell evaluation mechanism will reduce the expression as follows:

```
  scalarProd v1 v2
⇒ sumVec (zipWithVec (∗) v1 v2)
⇒ sumVec (zipWithVec (∗) (Indexed l1 ixf1) (Indexed l2 ixf2))
⇒ sumVec (Indexed (min l1 l2) (λix → ixf1 ix ∗ ixf2 ix)
⇒ forLoop (min l1 l2) 0 (λix s → s + ixf1 ix ∗ ixf2 ix)
```

The intermediate vector has disappeared and the only thing left is a for loop which computes the scalar product directly from the two argument vectors.

In the above example, fusion happened because although zipWithVec constructs a vector, it does not generate an array in the deep embedding. In fact, all standard vector operations (fmap, take, reverse, etc.) can be defined in a similar manner, without using internal storage. Whenever two such functions are composed, the intermediate vector is guaranteed to be eliminated. This guarantee by far exceeds guarantees given by conventional optimizing compilers.

So far, we have only seen one example of a vector producing function that uses internal storage: fromFunC. Thus intermediate vectors produced by fromFunC (for example as the result of ifC or while) will generally not be eliminated.

There are some situations when fusion is not beneficial, for instance in a function which access an element of a vector more than once. This will cause the elements to be recomputed. It is therefore important that the programmer has some way of backing out of using fusion and store the vector to memory. For this purpose we can provide the following function:

```
memorize  :: Syntactic a ⇒ Vector a → Vector a
memorize (Indexed l ixf) = Indexed l (λn → Arr l (toFunC ∘ ixf) <!> n)
```

The function memorize can be inserted between two functions to make sure that the intermediate vector is stored to memory. For example, if we wish to store the intermediate vector in our scalarProd function we can define it as follows:

```
scalarProd :: (Syntax a, Num a) ⇒ Vector a → Vector a → a
scalarProd a b = sumVec (memorize (zipWithVec (∗) a b))
```

Strong guarantees for fusion in combination with the function memorize gives the programmer a very simple interface which still provides powerful optimizations and fine grained control over memory usage.

The Vector type is not the only type which can benefit from fusion. In Feldspar there is a library for streams which captures the notion of an infinite sequence of values [2]. We use a shallow embedding of streams which also enjoys fusion in the same way as the vector library we have presented here. In fact, fusion is only one example of the kind of compile time transformations that can be achieved. We have a shallow embedding of monads which provides normalization of monadic expressions by applying the third monad law [15]. Common to all of these transformations is that they come with very strong guarantees that the transformations are actually performed.

6 Discussion

The technique described in this paper is a simple, yet powerful, method that gives a partial solution to the expression problem. By having a deep core language, we can add new interpretations without problem. And by means of the Syntactic class, we can add new language types and constructs with minimal effort. There is only one problem: Quite often, shallow language extensions are not efficiently (or not at all) expressible in the underlying deep implementation. When this happens, the deep implementation has to be extended. For example, when adding the Vector type (Section 4.5), the core language had to be extended with the constructors Arr, ArrLen and ArrIx. Note, however, that this is a quite modest extension compared to the wealth of operations available in the vector library. (This paper has only presented a small selection of the operations.)

Furthermore, it is probable that, once the deep implementation has reached a certain level of completeness, new shallow extensions will not require any new changes to the deep implementation. As an example, when adding the Option type (Section 4.4), the only deep extension needed was the Undef constructor. But this constructor does not have much to do with optional values; it could easily have been in the language already for other reasons.

In this paper we have shown a small but diverse selection of language extensions to demonstrate the idea of combining deep and shallow embeddings. The technique has been used with great success by the Feldspar team during the implementation of Feldspar. The extensions implemented include libraries for finite and infinite streams, fixed-point numbers and bit vectors. Recently, we have also used the technique to embed monadic computations in order to enrich Feldspar with controlled side-effects [15].

7 Related Work

A practical example of the design pattern we have outlined here is the embedded DSL Hydra which targets Functional Hybrid Modelling [13]. Hydra has a shallow embedding of signal relations on top of a deep embedding of equations. However, they do not have anything corresponding to our `Syntactic` class. Furthermore, they don't take advantage of any fusion-like properties of their embedding nor do they make any instances of standard Haskell classes.

Our focus in this paper has been on deep and shallow embeddings. But these are not the only techniques for embedding a language into a meta language. Another popular technique is the Finally Tagless technique [4]. The essence of Finally Tagless is to have an interface which abstracts over all interpretations of the language. In Haskell this is realized by a typeclass where each method corresponds to one language construct. Concrete interpretations are realized by creating a data type and making it an instance of the type class. For example, creating an abstract syntax tree would correspond to one interpretation and would have its own data type, evaluation would be another one. Since new interpretations and constructs can be added modularly (corresponds to adding new interpretation types and new interface classes respectively), Finally Tagless can be said to be a solution to the expression problem.

Our technique can be made to work with Finally Tagless as well. Creating a new embedding on top of an existing embedding simply amounts to creating a subclass of the type class capturing the existing embedding. However, care has to be taken if one would like to emulate a shallow embedding on top of a deep embedding and provide the kind of guarantees that we have shown in this paper. This will require an interpretation which maps some types to their abstract syntax tree representation and some types to their corresponding shallow embedding. Also, it is not possible to define general instances for standard Haskell classes for languages using the Finally Tagless technique. Instances can only be provided by particular interpretations.

The implementation of Kansas Lava [10] uses a combination of shallow and deep embedding. However, this implementation is quite different from what we are proposing. In our case, we use a *nested embedding*, where the deep embedding is used as the semantic domain of the shallow embedding. In Kansas Lava, the two embeddings exist in parallel — the shallow embedding is used for evaluation and the deep embedding for compilation. It appears that this design is not intended for extensibility: adding new interpretations is difficult due to the shallow embedding, and adding new constructs is difficult due to the deep embedding.

At the same time, Kansas Lava contains a type class `Pack` [11] that has some similarities to our `Syntactic` class. Indeed, using `Pack`, Kansas Lava implements support for optional values by mapping them to a pair of a boolean and a value. However, it is not clear from the publications to what extent `Pack` can be used to develop high-level language extensions and optimizations.

While our work has focused on making shallow extensions of deep embeddings, it is also possible to have the extensions as deep embeddings. This approach was used by Claessen and Pace [6] to implement a simple language for behavioral hardware description. The behavioral language is defined as a simple recursive data type whose meaning is given as a function mapping these descriptions into structural hardware descriptions in the EDSL Lava [3].

In their seminal work on compiling embedded languages, Elliott et al. use a type class `Syntactic` whose name gave inspiration to our type class [9]. However, their class is only used for overloading if expressions, and not as a general mechanism for extending the embedded language.

The way we provide fusion for vectors was first reported in [8] for the language Feldspar. The same technique was used in the language Obsidian [18] but it has never been documented that Obsidian actually supports fusion. The programming interface is very closely related to that provided by the Repa library [14], including the idea of guaranteeing fusion and providing programmer control for avoiding fusion. Although similar, the ideas were developed completely independently. It should also be noted that our implementation of fusion is vastly simpler than the one employed in Repa.

As in [9] we note that deeply embedded compilation relates strongly to partial evaluation. The shallow embeddings we describe can be seen as a compositional and predictable way to describe partial evaluation.

Acknowledgments. The authors would like to thank the Feldspar team for comments and discussions on the ideas presented in this paper. The first author would like to thank the participants of the DSL conference 2011 for the their constructive feedback, when part of the material in this paper was presented as an invited tutorial. This research is funded by Ericsson and the Swedish Foundation for Strategic Research. The Feldspar project is an initiative of and is partially funded by Ericsson Software Research and is a collaboration between Chalmers, Ericsson and ELTE University.

References

1. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of Feldspar – an embedded language for digital signal processing. In: Hage, J., Morazán, M.T. (eds.) IFL 2010. LNCS, vol. 6647, pp. 121–136. Springer, Heidelberg (2011)
2. Axelsson, E., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. In: Formal Methods and Models for Codesign, MemoCode. IEEE Computer Society (2010)

3. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware Design in Haskell. In: ICFP 1998: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, pp. 174–184. ACM (1998)
4. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming 19(5), 509–543 (2009)
5. Carlson, W., Hudak, P., Jones, M.: An experiment using Haskell to prototype "geometric region servers" for navy command and control. R. R. 1031 (1993)
6. Claessen, K., Pace, G.: An embedded language framework for hardware compilation. Designing Correct Circuits 2 (2002)
7. Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: Thiagarajan, P.S., Yap, R.H.C. (eds.) ASIAN 1999. LNCS, vol. 1742, pp. 62–73. Springer, Heidelberg (1999)
8. Dévai, G., Tejfel, M., Gera, Z., Páli, G., Nagy, G., Horváth, Z., Axelsson, E., Sheeran, M., Vajda, A., Lyckegård, B., Persson, A.: Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs. In: Proc. ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems (2010)
9. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming 13(3), 455–481 (2003)
10. Farmer, A., Kimmell, G., Gill, A.: What's the Matter with Kansas Lava? In: Page, R., Horváth, Z., Zsók, V. (eds.) TFP 2010. LNCS, vol. 6546, pp. 102–117. Springer, Heidelberg (2011)
11. Gill, A., Bull, T., Farmer, A., Kimmell, G., Komp, E.: Types and type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. In: Page, R., Horváth, Z., Zsók, V. (eds.) TFP 2010. LNCS, vol. 6546, pp. 118–133. Springer, Heidelberg (2011)
12. Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: Proc. Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA), pp. 223–232. ACM (1993)
13. Giorgidze, G., Nilsson, H.: Mixed-level embedding and JIT compilation for an iteratively staged DSL. In: Mariño, J. (ed.) WFLP 2010. LNCS, vol. 6559, pp. 48–65. Springer, Heidelberg (2011)
14. Keller, G., Chakravarty, M., Leshchinskiy, R., Jones, S.P., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in haskell. In: Proceedings of ICFP 2010, ACM SIGPLAN (2010)
15. Persson, A., Axelsson, E., Svenningsson, J.: Generic monadic constructs for embedded languages. In: Gill, A., Hage, J. (eds.) IFL 2011. LNCS, vol. 7257, pp. 85–99. Springer, Heidelberg (2012)
16. Peyton Jones, S.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
17. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI 1988, pp. 199–208. ACM (1988)
18. Svensson, J., Sheeran, M., Claessen, K.: Obsidian: A domain specific embedded language for parallel programming of graphics processors. In: Scholz, S.-B., Chitil, O. (eds.) IFL 2008. LNCS, vol. 5836, pp. 156–173. Springer, Heidelberg (2011)

The Blame Theorem for a Linear Lambda Calculus with Type Dynamic

Luminous Fennell and Peter Thiemann

University of Freiburg, Georges-Köhler-Allee 079, 79110 Freiburg, Germany
{fennell,thiemann}@informatik.uni-freiburg.de

Abstract. Scripting languages have renewed the interest in languages with dynamic types. For various reasons, realistic programs comprise dynamically typed components as well as statically typed ones. Safe and seamless interaction between these components is achieved by equipping the statically typed language with a type `Dynamic` and coercions that map between ordinary types and `Dynamic`. In such a gradual type system, coercions that map from `Dynamic` are checked at run time, throwing a blame exception on failure.

This paper enlightens a new facet of this interaction by considering a gradual type system for a linear lambda calculus with recursion and a simple kind of subtyping. Our main result is that linearity is orthogonal to gradual typing. The blame theorem, stating that the type coercions always blame the dynamically typed components, holds in a version analogous to the one proposed by Wadler and Findler, also the operational semantics of the calculus is given in a quite different way. The significance of our result comes from the observation that similar results for other calculi, e.g., affine lambda calculus, standard call-by-value and call-by-name lambda calculus, are straightforward to obtain from our results, either by simple modification of the proof for the affine case, or, for the latter two, by encoding them in the linear calculus.

Keywords: linear typing, gradual typing, subtyping.

1 Introduction

Many of today's computing systems contain multi-core processors or regularly access distributed services. The behavior of such a concurrent system depends on the interaction of its components, so it is important for its construction and maintenance to specify and enforce communication protocols and security policies.

Ideally, static analysis should guarantee the adherence of a system to protocols and security policies, but such a comprehensive analysis is not always possible. For example, there may be legacy components that were implemented before a specification was established, or there may be components implemented in a dynamic language that are impractical to analyze. These are situations, where it is desirable to have the ability to combine static and dynamic enforcement of specifications.

Gradual typing is a specification method that satisfies these requirements. Gradual type systems were developed to introduce typing into dynamic languages to improve the efficiency of their execution as well as their maintainability. They are also useful to mediate accesses between statically typed and dynamically typed program components [5,10,12,22,24]. A commonality of these approaches is that they extend the statically typed language with a type `Dynamic` along with type coercions that map into and out of the `Dynamic` type. The dynamically typed part is then modeled as a program where every result is coerced to `Dynamic` and where every elimination form is preceded by a coercion from `Dynamic`. Clearly, the mapping into the `Dynamic` type never fails whereas the mapping from the `Dynamic` type to a, say, function type may fail if the actual dynamic value happens to be an integer. The standard implementation of a dynamic value is a pair consisting of a value and an encoding of its run-time type. The coercion from `Dynamic` to another type checks the run-time type against the expected one and throws an exception if the types do not match.

As a first step towards gradual enforcement of communication policies as embodied by work on session types [18], this paper considers the interaction of the type `Dynamic` and blame assignment with linearity [9]. A linear calculus formalizes single-use resources in the sense that each introduced value can and must only be eliminated exactly once. Thus, linearity is an important property in all type systems that are related to managing stateful resources, in particular communication channels in a session calculus [7, 20].

A linear lambda calculus has two dimensions that the type `Dynamic` may address: it may hide and dynamically check the type structure or the linearity requirement (or both). In this paper, we let the type system enforce linearity statically and have the type `Dynamic` mostly hide the type structure. The resulting calculus extends Turner and Wadler's linear lambda calculus [21] with cast expressions inspired by Wadler and Findler's blame calculus [22], as demonstrated in the following example program. It first wraps a pair of a linear function and a replicable function (the exponential in $!\lambda b.\ b - 1$ introduces a replicable value) into a dynamic value. Then it unwraps the components and demonstrates their use as functions and replicable functions.

$$\begin{array}{l}
\texttt{let}\ p = \langle D \Leftarrow (int \multimap int) \otimes !(int \multimap int)\rangle((\lambda a.\ a + 2) \otimes (!\lambda b.\ b - 1))\ \texttt{in} \\
\texttt{let}\ f \otimes g_1 = \langle D \otimes D \Leftarrow D\rangle p\ \texttt{in} \\
\texttt{let}\ !g = \langle !(int \multimap int) \Leftarrow D\rangle g_1\ \texttt{in} \\
\quad g\ (g\ (\langle int \multimap int \Leftarrow D\rangle f\ 0))
\end{array}$$

The next example illustrates a possible interaction of typed and untyped code through a latently typed communication channel, which is modeled with a linear, dynamic type. Initially, the channel is represented by a linear function `submit` $: int \multimap D$: it expects an integer and its remaining behavior is `Dynamic`. To use it after sending the integer 5, we attempt to coerce the remaining behavior to a *string* $\multimap \star$ channel. An exception will be raised if it turns out that the run-time type of `submit'` does not have this type.

$$\texttt{let submit'} = \langle \textit{string} \multimap \star \Leftarrow D\rangle(\texttt{submit}\ 5)\ \texttt{in submit'}\ \textit{``Hello''}$$

Our calculus has a facility to hide part of a linearity requirement. To this end, we extend the compatibility relation to admit casts that shortcut the elimination of the exponential, effectively allowing dynamic replicable values to be treated as linear. With this extension, the above example also executes correctly if the run-time type of `submit'` has a form like $!(!\textit{string} \multimap \star)$. In both cases, we show that linearity and gradual typing are orthogonal. We first define a suitable linear lambda calculus with a gradual type system and blame assignment in the manner of Taha and Siek [14, 16] and Wadler and Findler [22], respectively. We prove the soundness of this system and then follow Wadler and Findler in formulating subtyping relations and proving a blame theorem.

Overview

Section 2 defines the static and dynamic semantics of a typed linear lambda calculus with recursion.

Section 3 contains our **first contribution**, an extension of the calculus with gradual types, which adds a type `Dynamic` and cast expressions to map between ordinary types and `Dynamic`. We prove type soundness of this extended calculus. Our gradual extension is more modular than Wadler and Findler's blame calculus [22] thus making our calculus easier to extend and making our approach easier to apply to other calculi. Section 4 recalls the subtyping relation of the blame calculus and adapts it to the linearly typed setting.

Section 5 contains our **second contribution**. We state and prove the blame theorem for the linear blame calculus.

Section 6 contains our **third contribution**. We show how to exploit the modularity by integrating shortcut casts in the system that enable direct casting from a replicable value wrapped in the dynamic type to the underlying type. Type soundness and the blame theorem still hold for this extension.

All proofs may be found in a companion technical report.[1]

2 Linear Lambda Calculus

Turner and Wadler's linear lambda calculus [21] is the basis of our investigation. Figure 1 contains the syntax of their original calculus extended to deal with recursive exponentials. Types are unit types, pairs, linear functions, and exponentials. The term syntax is based on two kinds of variables, linear ones a and non-linear ones x. There are introduction and elimination forms for unit values $\star$, linear pairs $e \otimes e$, and lambda abstractions. By default all expressions are linear. Following the work on Lily [6] and in this respect generalizing Turner and Wadler, replicable and recursive expressions can be constructed with the

[1] `http://proglang.informatik.uni-freiburg.de/projects/gradual/`

$$\begin{array}{rrl} \text{linear variables } a, b ::= & \ldots & \text{non-linear variables } x, y ::= \ldots \\ \text{types} & t ::= & \star \mid t \otimes t \mid t \multimap t \mid !t \\ \text{expressions} & e, f ::= & \star \mid \texttt{let}\ \star = e\ \ \texttt{in}\ e \mid e \otimes e \mid \texttt{let}\ a \otimes a = e\ \ \texttt{in}\ e \mid \\ & & \lambda a.\ e \mid e\ e \mid a \mid !(x = e) \mid \texttt{let}\ !x = e\ \ \texttt{in}\ e \mid x \\ \text{storable values} & s ::= & \star \mid a \otimes a \mid \lambda a.\ e \mid !x \end{array}$$

Fig. 1. Original type and expression syntax

exponential $!(x = e)$, where the non-linear variable x may appear in e. The exponential can be seen as a suspended computation which, when forced, may return some value. We may also write $!e$ for $!(x = e)$ when x does not occur in e. The elimination forms, except function application, are expressed using suitable `let`-forms. Our presentation is based on Turner and Wadler's second, heap-based semantics. For that reason, our evaluation rules always return linear variables (heap pointers) that refer to storable values s. The latter only consist of introduction forms applied to suitable variables. In particular, the storable value for the exponential refers to a non-linear variable that keeps the real payload that is repeatedly evaluated.

The corresponding original operational semantics is given in Figure 2. The reduction rules are defined in terms of heap-expression configurations $\{H\}e$. The heap contains two kinds of bindings. Linear variables are bound to linear values, whereas standard variables are bound to expressions. The intention is that a linear variable is used exactly once and removed after use, whereas a standard variable can be used many times and its binding remains in the heap. In the rules we use the notation a^* to denote that a should be a fresh variable. Figure 3 recalls the inference rules of the corresponding type system as presented by Turner and Wadler [21]. There are two kinds of type environments, a linear one, Γ, and a non-linear one, Θ. Linearity is enforced by splitting the linear type environment in the inference rules in the usual way and by requiring the linear environment to be empty in the usual places (constants, non-linear variables, and the exponential). In addition to the expression typing, there is a heap typing judgment of the form $\vdash H : \Theta; \Gamma$ that relates a heap H to environments Θ and Γ that provide bindings for non-linear and linear variables, respectively. The judgment is defined in Figure 4 and is a minor modification of the corresponding judgment of Turner and Wadler with the *Closure* rule adapted to accomodate recursive bindings. It is needed for the type soundness proof.

The type of a heap is a pair of a linear and a non-linear type environment reflecting the respective variable bindings in the heap. Linear variables are defined by linear values, non-linear variables are defined by expressions that have no free linear variables. Note that the type system for expressions is syntax-directed and that the heap typing rules are invertible as in Turner and Wadler's original work.

$$
\begin{array}{rl}
\text{heaps} & H ::= \ \cdot \mid H, x = e \mid H, a = s \\
\text{evaluation contexts} & E ::= [\,] \otimes e \mid a \otimes [\,] \mid [\,]\ e \mid a\ [\,] \mid \\
 & \quad \texttt{let}\ a \otimes b = [\,]\ \texttt{in}\ e \mid \texttt{let}\ !x = [\,]\ \texttt{in}\ e \mid \texttt{let}\ \star = [\,]\ \texttt{in}\ e
\end{array}
$$

$\boxed{\{H\}e \to \{H\}e'}$

$$
\begin{array}{rlcl}
\star\text{-}I & \{H\}\star & \to & \{H, a^* = \star\}a^* \\
\star\text{-}E & \{H, a = \star\}\texttt{let}\ \star = a\ \texttt{in}\ e & \to & \{H\}e \\
\otimes\text{-}I & \{H\}a_1 \otimes a_2 & \to & \{H, b^* = a_1 \otimes a_2\}b^* \\
\otimes\text{-}E & \{H, a'' = a' \otimes b'\}\texttt{let}\ a \otimes b = a''\ \texttt{in}\ e & \to & \{H\}e[a/a', b/b'] \\
\multimap\text{-}I & \{H\}\lambda a.\ e & \to & \{H, b^* = \lambda a.\ e\}b^* \\
\multimap\text{-}E & \{H, b = \lambda a.\ e\}b\ a' & \to & \{H\}e[a/a'] \\
!\text{-}I & \{H\}!(x = e) & \to & \{H, b^* = !y^*, y^* = e[x/y^*]\}b^* \\
!\text{-}E & \{H, b = !y\}\texttt{let}\ !x = b\ \texttt{in}\ e & \to & \{H\}e[x/y] \\
\mathit{Var} & \{H, x = e\}x & \to & \{H, x = e\}e
\end{array}
$$

$$
\frac{\{H\}e \to \{H'\}e'}{\{H\}E[e] \to \{H'\}E[e']}\ \mathit{Context}
$$

Fig. 2. Original operational semantics

3 Linear Lambda Calculus with Type Dynamic

To introduce dynamic and gradual typing into the calculus, we extend the Turner-Wadler calculus with new constructs adapted from Wadler and Findler's work [22] as shown in Figure 5. There is a new kind of wrapper value $D_c(a)$, which denotes a dynamic value pointing to a variable a with type constructor c, a new type D of dynamic values, and a cast expression $\langle t' \Leftarrow t\rangle^p\ e$. The cast is supposed to transform the type of e from t to t'. Casts are annotated with a blame label p to later identify the place that caused a run-time error resulting from the cast. A plain blame label p indicates *positive blame* whereas an inverted label $\bar{p}$ indicates *negative blame*. Inversion is involutory so that $\bar{\bar{p}} = p$. The wrapper values are needed by the casts to perform run-time type checking. Unlike the wrapper values of Wadler and Findler's blame calculus, they are only defined for linear variables and occur exclusively as heap values.

The extensions to the type system are also inspired by the blame calculus. The additional rules are given in Figure 6. The *Cast* rule changes the type t_2 of an underlying expression to a compatible type t_1. The compatibility relation $t \sim t$ is also defined in Figure 6. It is analogous to the definition in the blame calculus. However, as subset types are not considered here, the only possible casts are either trivial or cast from or to D. The four value rules are only used inside of heap typings.

Linear type environments Γ,Δ and non linear type environments Θ

$$\Gamma, \Delta ::= \ \text{-} \mid \Gamma, a : t \qquad \Theta ::= \ \text{-} \mid \Theta, x : t$$

$\boxed{\Theta; \Gamma \vdash e : t}$

$$\frac{}{\Theta, x : t; \ \text{-} \vdash x : t}\ \mathit{Var} \qquad \frac{}{\Theta; a : t \vdash a : t}\ \mathit{LVar}$$

$$\frac{}{\Theta; \ \text{-} \vdash \star : \star}\ \star{-}I \qquad \frac{\Theta; \Gamma \vdash e : \star \quad \Theta; \Delta \vdash f : t}{\Theta; \Gamma, \Delta \vdash \texttt{let}\ \star = e\ \ \texttt{in}\ f : t}\ \star{-}E$$

$$\frac{\Theta; \Gamma \vdash e : t_e \quad \Theta; \Delta \vdash f : t_f}{\Theta; \Gamma, \Delta \vdash e \otimes f : t_e \otimes t_f}\ \otimes{-}I \qquad \frac{\Theta; \Gamma \vdash e : t_a \otimes t_b \quad \Theta; \Delta, a : t_a, b : t_b \vdash f : t_f}{\Theta; \Gamma, \Delta \vdash \texttt{let}\ a \otimes b = e\ \ \texttt{in}\ f : t_f}\ \otimes{-}E$$

$$\frac{\Theta; \Gamma, a : t_a \vdash e : t_e}{\Theta; \Gamma \vdash \lambda a.\ e : t_a \multimap t_e}\ \multimap{-}I \qquad \frac{\Theta; \Gamma \vdash e : t_f \multimap t_e \quad \Theta; \Delta \vdash f : t_f}{\Theta; \Gamma, \Delta \vdash e\ f : t_e}\ \multimap{-}E$$

$$\frac{\Theta, x : t; \ \text{-} \vdash e : t}{\Theta; \ \text{-} \vdash !(x = e) : !t}\ !{-}I \qquad \frac{\Theta; \Gamma \vdash e : !t_e \quad \Theta, x : t_e; \Delta \vdash f : t_f}{\Theta; \Gamma, \Delta \vdash \texttt{let}\ !x = e\ \ \texttt{in}\ f : t_f}\ !{-}E$$

Fig. 3. Original linear type system

$\boxed{\vdash H : \Theta; \Gamma}$

$$\frac{}{\vdash \cdot : \ \text{-}\ ; \ \text{-}}\ \mathit{Empty}$$

$$\frac{\vdash H : \Theta; \Gamma \quad \Theta, x : t; \ \text{-} \vdash e : t}{\vdash H, x = e : \Theta, x : t; \Gamma}\ \mathit{Closure} \qquad \frac{\vdash H : \Theta; \Gamma, \Delta \quad \Theta; \Delta \vdash s : t}{\vdash H, a = s : \Theta; \Gamma, a : t}\ \mathit{Value}$$

Fig. 4. Heap typing

Figure 7 shows the reduction rules that perform run-time type checking by manipulating casts and wrapper values. These rules are original to our system and they implement casting in a different way than Wadler and Findler. The first set of rules concerns casting into the dynamic type. It distinguishes casts by checking the source type for groundness (a type constructor directly applied to D). A cast from a ground type to D corresponds to a wrapper application (second part of first group) whereas any other cast is decomposed into a wrapper cast *preceded* by a cast into the ground type. It is not possible to merge these two parts because a wrapper cannot be applied before its argument is properly cast and evaluated.

The second group of rules specifies the functorial casts that descent structurally in the type. For the types dynamic and unit, the casts do nothing. For pairs, the casts are distributed to the components. For functions, the cast is turned into a wrapper function, which casts the argument with types reversed and the blame label "inverted" and which casts the result with the types in the

types	$t ::=$	$\ldots \mid D$	expressions	$e ::=$	$\ldots \mid \langle t \Leftarrow t \rangle^p \; e$
blame labels	$p, q ::=$	$\ldots$	storable values	$s ::=$	$\ldots \mid D_c(a)$
constructors	$c ::=$	$\star \mid \otimes \mid \multimap \mid !$			

Fig. 5. Syntactic extensions

$\boxed{\Theta; \Gamma \vdash e : t}$, $\boxed{\Theta; \Gamma \vdash s : t}$

$$\frac{\Theta; \Gamma \vdash e : t_2 \quad t_1 \sim t_2}{\Theta; \Gamma \vdash \langle t_1 \Leftarrow t_2 \rangle^p e : t_1} \; \mathit{Cast}$$

$$\frac{}{\Theta; a : \star \vdash D_\star(a) : D} \; \mathit{DynVal} \text{ - } \star \qquad \frac{}{\Theta; a : D \otimes D \vdash D_\otimes(a) : D} \; \mathit{DynVal} \text{ - } \otimes$$

$$\frac{}{\Theta; a : D \multimap D \vdash D_\multimap(a) : D} \; \mathit{DynVal} \text{ - } \multimap \qquad \frac{}{\Theta; a : !D \vdash D_!(a) : D} \; \mathit{DynVal} \text{ - } !$$

Compatibility $\boxed{t \sim t'}$

$$\frac{}{\star \sim \star} \qquad \frac{}{t \sim D} \qquad \frac{}{D \sim t} \qquad \frac{t_1 \sim t_1' \quad t_2 \sim t_2'}{t_1 \otimes t_2 \sim t_1' \otimes t_2'} \qquad \frac{t_1 \sim t_1' \quad t_2 \sim t_2'}{t_1 \multimap t_2 \sim t_1' \multimap t_2'} \qquad \frac{t \sim t'}{!t \sim !t'}$$

Fig. 6. Type system extensions and compatibility

original order. A cast between exponential types unfolds the exponential and performs the cast on the underlying linear value.

The third group of rules concerns casting from type D. This group has a simpler structure than the first group because top-level unwrapping can be done in any case. The "remaining" cast is then applied to the unwrapped value of ground type. It is put to work by the functorial rules.

The last group defines the failure rules. They apply to unwrapping casts that are applied to dynamic values with the wrong top type constructor. These are the only rules that generate blame.

The resulting linear blame calculus is still type safe. We can prove preservation and progress lemmas extending the ones given by Turner and Wadler.

Lemma 1 (Type preservation). *If* $\{H\}e \rightarrow \{H'\}e'$ *and* $\vdash H : \Theta; \Gamma$ *and* $\Theta; \Gamma \vdash e : t$ *then there exist* Θ' *and* Γ' *such that* $\vdash H' : \Theta, \Theta'; \Gamma'$ *and* $\Theta, \Theta'; \Gamma' \vdash e' : t$.

Lemma 2 (Progress). *If* $\vdash H : \Theta; \Gamma$ *and* $\Theta; \Gamma \vdash e : t$ *then one of the following alternatives holds: (i) there exist* H' *and* e' *such that* $\{H\}e \rightarrow \{H'\}e'$, *(ii) there exists* p *such that* $\{H\}e \rightarrow \Uparrow p$, *or (iii)* e *is a linear variable.*

The last case may be surprising, as the standard formulation of a progress lemma states the outcome of a value at this point. However, in the present setting, all

$$\begin{array}{rrl} \text{results} & r ::= & \{H\}e \mid \ \Uparrow p \\ \text{eval. contexts} & E ::= & \dots \mid \langle t_1 \Leftarrow t_2 \rangle^p [\,] \end{array}$$

$\boxed{\{H\}e \to r}$

$$\frac{\{H\}e \to \Uparrow p}{\{H\}E[e] \to \Uparrow p}\ \textit{ContextFail}$$

Casting to D

$$\begin{array}{rll}
\textit{CastDyn} \text{ - } \otimes & \{H\}\langle D \Leftarrow t_1 \otimes t_2 \rangle^p a & \to \{H\}\langle D \Leftarrow D \otimes D \rangle^p \langle D \otimes D \Leftarrow t_1 \otimes t_2 \rangle^p a \\
& & \quad \text{if } t_1 \neq D \vee t_2 \neq D \\
\textit{CastDyn} \text{ - } \multimap & \{H\}\langle D \Leftarrow t_1 \multimap t_2 \rangle^p a & \to \{H\}\langle D \Leftarrow D \multimap D \rangle^p \langle D \multimap D \Leftarrow t_1 \multimap t_2 \rangle^p a \\
& & \quad \text{if } t_1 \neq D \vee t_2 \neq D \\
\textit{CastDyn} \text{ - } ! & \{H\}\langle D \Leftarrow !t \rangle^p a & \to \{H\}\langle D \Leftarrow !D \rangle^p \langle !D \Leftarrow !t \rangle^p a \\
& & \quad \text{if } t \neq D
\end{array}$$

$$\begin{array}{rll}
\textit{WrapDyn} \text{ - } \star & \{H\}\langle D \Leftarrow \star \rangle^p a & \to \{H, b^* = D_\star(a)\}b^* \\
\textit{WrapDyn} \text{ - } \otimes & \{H\}\langle D \Leftarrow D \otimes D \rangle^p a & \to \{H, b^* = D_\otimes(a)\}b^* \\
\textit{WrapDyn} \text{ - } \multimap & \{H\}\langle D \Leftarrow D \multimap D \rangle^p a & \to \{H, b^* = D_\multimap(a)\}b^* \\
\textit{WrapDyn} \text{ - } ! & \{H\}\langle D \Leftarrow !D \rangle^p a & \to \{H, b^* = D_!(a)\}b^*
\end{array}$$

Functorial casts

$$\begin{array}{rll}
\textit{Cast} \text{ - } D & \{H\}\langle D \Leftarrow D \rangle^p a & \to \{H\}a \\
\textit{Cast} \text{ - } \star & \{H\}\langle \star \Leftarrow \star \rangle^p a & \to \{H\}a \\
\textit{Cast} \text{ - } \otimes & \{H\}\langle t_1' \otimes t_2' \Leftarrow t_1 \otimes t_2 \rangle^p a & \to \{H\}\texttt{let } {a_1}^* \otimes {a_2}^* = a \texttt{ in} \\
& & \qquad \langle t_1' \Leftarrow t_1 \rangle^p {a_1}^* \otimes \langle t_2' \Leftarrow t_2 \rangle^p {a_2}^* \\
\textit{Cast} \text{ - } \multimap & \{H\}\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2 \rangle^p a & \to \{H\}\lambda b^*.\ \langle t_2' \Leftarrow t_2 \rangle^p (a\ (\langle t_1 \Leftarrow t_1' \rangle^{\bar{p}} b^*)) \\
\textit{Cast} \text{ - } ! & \{H\}\langle !t' \Leftarrow !t \rangle^p a & \to \{H\}\texttt{let } !x^* = a \texttt{ in } !(y^* = \langle t' \Leftarrow t \rangle^p x^*)
\end{array}$$

Casting from D

$$\begin{array}{rll}
\textit{FromDyn} \text{ - } \star & \{H, a = D_\star(a')\}\langle \star \Leftarrow D \rangle^p a & \to \{H\}a' \\
\textit{FromDyn} \text{ - } \otimes & \{H, a = D_\otimes(a')\}\langle t_1 \otimes t_2 \Leftarrow D \rangle^p a & \to \{H\}\langle t_1 \otimes t_2 \Leftarrow D \otimes D \rangle^p a' \\
\textit{FromDyn} \text{ - } \multimap & \{H, a = D_\multimap(a')\}\langle t_1 \multimap t_2 \Leftarrow D \rangle^p a & \to \{H\}\langle t_1 \multimap t_2 \Leftarrow D \multimap D \rangle^p a' \\
\textit{FromDyn} \text{ - } ! & \{H, a = D_!(a')\}\langle !t \Leftarrow D \rangle^p a & \to \{H\}\langle !t \Leftarrow !D \rangle^p a'
\end{array}$$

Failing casts from D

$$\begin{array}{rlcll}
\textit{CastFail} \text{ - } \star & \{H, a = D_\star(a')\}\langle t \Leftarrow D \rangle^p a & \to & \Uparrow p & \text{if } t \neq \star \text{ and } t \neq D \\
\textit{CastFail} \text{ - } \otimes & \{H, a = D_\otimes(a')\}\langle t \Leftarrow D \rangle^p a & \to & \Uparrow p & \text{if } t \neq t_1 \otimes t_2 \text{ and } t \neq D \\
\textit{CastFail} \text{ - } \multimap & \{H, a = D_\multimap(a')\}\langle t \Leftarrow D \rangle^p a & \to & \Uparrow p & \text{if } t \neq t_1 \multimap t_2 \text{ and } t \neq D \\
\textit{CastFail} \text{ - } ! & \{H, a = D_!(a')\}\langle t \Leftarrow D \rangle^p a & \to & \Uparrow p & \text{if } t \neq !t' \text{ and } t \neq D
\end{array}$$

Fig. 7. Reduction rule extensions

Ground types

$$g ::= \star \mid D \otimes D \mid D \multimap D \mid !D$$

Subtyping $\boxed{t <: t'}$

$$\frac{}{\star <: \star} \quad \frac{}{D <: D} \quad \frac{t <: g}{t <: D} \quad \frac{t_1 <: t_1' \quad t_2 <: t_2'}{t_1 \otimes t_2 <: t_1' \otimes t_2'} \quad \frac{t_1' <: t_1 \quad t_2 <: t_2'}{t_1 \multimap t_2 <: t_1' \multimap t_2'} \quad \frac{t <: t'}{!t <: !t'}$$

Positive subtyping $\boxed{t <:^+ t'}$

$$\frac{}{\star <:^+ \star} \quad \frac{}{t <:^+ D} \quad \frac{t_1 <:^+ t_1' \quad t_2 <:^+ t_2'}{t_1 \otimes t_2 <:^+ t_1' \otimes t_2'} \quad \frac{t_1' <:^- t_1 \quad t_2 <:^+ t_2'}{t_1 \multimap t_2 <:^+ t_1' \multimap t_2'} \quad \frac{t <:^+ t'}{!t <:^+ !t'}$$

Negative subtyping $\boxed{t <:^- t'}$

$$\frac{}{\star <:^- \star} \quad \frac{}{D <:^- t} \quad \frac{t <:^- g}{t <:^- t'} \quad \frac{t_1 <:^- t_1' \quad t_2 <:^- t_2'}{t_1 \otimes t_2 <:^- t_1' \otimes t_2'} \quad \frac{t_1' <:^+ t_1 \quad t_2 <:^- t_2'}{t_1 \multimap t_2 <:^- t_1' \multimap t_2'} \quad \frac{t <:^- t'}{!t <:^- !t'}$$

Fig. 8. Subtyping rules

values are represented by (linear) pointers to the heap. Thus, the presence of a linear variable indicates that the evaluation returns the linear value pointed to by the variable.

From these two lemmas, we can prove type soundness in the usual way [23].

Theorem 1 (Type soundness). *If* $\vdash H : \Theta; \Gamma$ *and* $\Theta; \Gamma \vdash e : t$ *then either (i)* $\{H\}e$ *diverges, (ii)* $\{H\}e \rightarrow^* \Uparrow p$*, for some* p*, or (iii)* $\{H\}e \rightarrow^* \{H'\}a$*, for some* H' *and* a*.*

Wadler and Findler also define a transformation to embed untyped programs into the blame calculus. For our system, an analogous transformation can be defined for a version of the untyped linear lambda calculus [3]. For details we refer to the technical report above mentioned in Section 1.

4 Subtyping

The subtyping relation, defined in Figure 8, holds between types that can be cast without raising blame. Following the subtyping relation for the blame calculus, it is split into three relations. Positive subtyping characterizes casts that never raise positive blame, as we show in Section 5. Negative subtyping characterizes casts that never raise negative blame and the plain subtyping relation characterizes casts that never raise blame at all.

Ground types, also defined in Figure 8, play a special role in the definition of subtyping because it turns out that casting into a ground type never raises blame (viz. the definition of negative subtyping). Ground types increase the scope of the subtyping relation based on this observation.

$\boxed{e \text{ sf } p}$, $\boxed{s \text{ sf } p}$

$$\frac{t_2 <:^{+} t_1 \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2\rangle^{p} e \text{ sf } p} \quad \frac{t_2 <:^{-} t_1 \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2\rangle^{\bar{p}} e \text{ sf } p} \quad \frac{p \neq q \quad p \neq \bar{q} \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2\rangle^{q} e \text{ sf } p}$$

$$\frac{}{\star \text{ sf } p} \quad \frac{}{a \text{ sf } p} \quad \cdots \quad \frac{e \text{ sf } p}{\lambda a.\, e \text{ sf } p} \quad \cdots$$

Fig. 9. Blame-safe expressions

The subtyping rules involving D, the base type $\star$, and the linear function type are identical to Wadler and Findler's rules. The rules for pairs and exponentials are new but analogous to the existing rules: both are covariant and immutable. They do not give rise to new problems and similar results hold.

Lemma 3 (Factoring subtyping).
$t_1 <: t_2$ if and only if $t_1 <:^{+} t_2$ and $t_1 <:^{-} t_2$.

The inference rules for $t' <:^{+} t$ are syntax-directed which enables the proof of the following lemma.

Lemma 4. *If $D <:^{+} t$ then $t = D$.*

We have not considered naive subtyping in this paper as it is not needed for proving the blame theorem.

5 The Blame Theorem

The slogan of the blame theorem [22] is that "well typed programs can't be blamed". In other words, if a program raises blame, a dynamically typed expression is responsible.

To prove this theorem, Wadler and Findler define the notion of *safe expressions*. A safe expression for a specific blame label will never raise that blame label. This notion of blame safety can only be violated through a cast expression $\langle t' \Leftarrow t\rangle^{p} e$ and it can be derived from the particular subtyping relation which holds between the types t' and t. Therefore, given preservation and progress of the reduction rules with respect to blame safety, the potential for raising blame is defined by the types used in the individual cast expressions of a program.

We replicate this reasoning for the linear blame calculus by defining blame safety and showing its preservation and progress for typed configurations. It is not sufficient to define blame safety just for expressions like in the blame calculus, because our operational semantics is defined in terms of heap-expression configurations $\{H\}e$. As variables can refer to expressions in the heap, they are only safe if the referred expressions are safe. A sufficient condition to ensure safety is therefore to require the entire heap to contain safe expressions for a particular blame label. The precise definition may be found in the technical report.

We also explored the idea of making variables safe depending on the expression that they refer to in the heap. However, this idea turned out to require a lot more formalism (e.g., formalizing reachability in the heap) because of recursive bindings in the heap.

Some of the adapted rules for blame safety of expressions are shown in Figure 9. The complete definition is given in the technical report. The cast expressions are the only places, where blame safety may be violated. The rules for cast expressions are analogous to those of the blame calculus: a cast is safe for p if it has label p and its types are related by positive subtyping, if it has label $\bar{p}$ and its types are related by negative subtyping, or if it has a label unrelated to p. The remaining rules just propagate the safety requirement to their subexpressions. Variables, wrapped variables, and the unit value are always safe.

The preservation lemma for blame safety establishes the following fact: The reduction of a configuration which is safe for a blame label p results in a configuration which is again safe for p.

Lemma 5 (Safe configurations: preservation). *Suppose that* $\vdash H : \Theta; \Gamma$ *and* $\Theta; \Gamma \vdash e : t$ *and* $\{H\}e \;\, \mathrm{sf} \;\, p$ *and* $\{H\}e \to \{H'\}e'$ *then* $\{H'\}e' \;\, \mathrm{sf} \;\, p$.

The proof is by induction on the reduction relation.

The next essential part to the blame theorem is the fact that blame-free progress of safe configurations is guaranteed:

Lemma 6 (Progress of safe configurations).
If $\{H\}e \;\, \mathrm{sf} \;\, p$ *then* $\{H\}e \not\to \Uparrow p$.

Proof. We actually prove the contraposition "If $\{H\}e \to \Uparrow p$ then *not* $e \;\, \mathrm{sf} \;\, p$" by induction on the reduction relation. This induction uses Lemma 4.

Like in Wadler and Findler's blame calculus the blame theorem is a corollary of Lemmas 5 and 6:

Corollary 1 (Well typed linear programs can't be blamed). *Let* $\{H\}e$ *be a well typed configuration and* $\langle t_1 \Leftarrow t_2 \rangle^p f$ *a subexpression of* e *or a subexpression in a binding in* H *containing the only occurrence of* p *in any expression reachable by* $\{H\}e$. *If* $t_2 <:^+ t_1$ *then* $\{H\}e \not\to^* \Uparrow p$. *If* $t_2 <:^- t_1$ *then* $\{H\}e \not\to^* \Uparrow \bar{p}$. *If* $t_2 <: t_1$ *then* $\{H\}e \not\to^* \Uparrow p$ *and* $\{H\}e \not\to^* \Uparrow \bar{p}$.

6 Shortcut Casts

Consider the following simple example:

$$\begin{gathered}\texttt{let}\ f = \langle D \Leftarrow !(int \multimap int)\rangle !\lambda a.\ a + 1\ \texttt{in} \\ \texttt{let}\ !f_1 = \langle !D \Leftarrow D\rangle f\ \texttt{in}\ \langle int \multimap int \Leftarrow D\rangle f_1\ 0\end{gathered}$$

The dynamically typed variable f is used exactly once, but an extra cast and exponential elimination has to be inserted in the second line to satisfy the type checker. The problem in this case is that the variable f "hides" a value of type

$!(int \multimap int)$, but the single use of the function requires the unreplicated type $(int \multimap int)$.

This mismatch can be addressed by extending our system with "subtyping" of the form $!t <: t$ where the conversion from the left side to the right side must be made explicit via casting. This notion of subtyping is based on the observation that each replicable value can serve as a linear value. In a first step, we only address casting from a dynamic type that wraps a replicated type to the underlying unreplicated type. To this end, we replace the *CastFail-!* rule by a rule that directly dereferences the exponential and then retries the cast on the unwrapped value (after evaluation of x):

$$\mathit{FromDyn}\text{ - }!t \quad \{H, a = D_!(a'), a' = !x\}\langle t \Leftarrow D\rangle^p a \quad \rightarrow \quad \{H\}\langle t \Leftarrow D\rangle^p x \quad \text{if } t \neq !t'$$

With this rule in place, the example above can be rewritten as follows without explicitly casting f from D to $!D$:

$$\begin{array}{l}\texttt{let } f = \langle D \Leftarrow !(int \multimap int)\rangle !\lambda a.\ a + 1 \texttt{ in} \\ \quad \langle int \multimap int \Leftarrow D\rangle f\ 0\end{array}$$

The linear f, which wraps a value of exponential type, is directly converted to a linear function. This use of f is more convenient than explicit unwrapping.

Our semantics also handles the unlikely case of a multiply wrapped type like $!!!\star$. Converting a value of this type into the dynamic type requires $\texttt{let}\, s = \langle D \Leftarrow !!!\star\rangle !!!\star \texttt{ in} \ldots$ whereas using it (once) just requires casting with $\langle \star \Leftarrow D\rangle s$.

All the results we have so far, type soundness, properties of subtyping, and the blame theorem, still hold without change.

A less satisfactory consequence is that some sequences of cast expressions cannot be simplified, anymore. As an example, consider the expression

$$\texttt{let } f : int \multimap int = \ldots \texttt{ in } f\ (\langle int \Leftarrow D\rangle(\langle D \Leftarrow !int\rangle !42))$$

This expression is legal in the type system of Section 3 and it executes without run-time errors in the extended semantics of the current section. In other systems, it is possible to optimize such a (non-failing) sequence of casts to just a single cast as in:

$$f\ (\langle int \Leftarrow !int\rangle !42)$$

However, our present system does not permit this reduced cast. In fact, our operational semantics cannot execute this cast because its left and right side types are not compatible. This restriction is also enforced by the type system.

This problem can be amended by adding evaluation rules for the evaluation of casts from an exponential type to a non-dynamic type. The strategy is the same as for the *FromDyn* - $!t$ rule: eliminate the exponential and retry the cast.

$$\begin{array}{rll}\mathit{Cast}{-}!-\star & \{H, a = !x\}\langle \star \Leftarrow !t\rangle^p a & \rightarrow\ \{H\}\langle \star \Leftarrow t\rangle^p x \\ \mathit{Cast}{-}!-\otimes & \{H, a = !x\}\langle t_1 \otimes t_2 \Leftarrow !t\rangle^p a & \rightarrow\ \{H\}\langle t_1 \otimes t_2 \Leftarrow t\rangle^p x \\ \mathit{Cast}{-}!-\multimap & \{H, a = !x\}\langle t_1 \multimap t_2 \Leftarrow !t\rangle^p a & \rightarrow\ \{H\}\langle t_1 \multimap t_2 \Leftarrow t\rangle^p x\end{array}$$

$\boxed{t \lesssim t'}$

$$\frac{}{\star \lesssim \star} \quad \frac{}{t \lesssim D} \quad \frac{}{D \lesssim t} \quad \frac{t_1 \lesssim t_1' \quad t_2 \lesssim t_2'}{t_1 \otimes t_2 \lesssim t_1' \otimes t_2'} \quad \frac{t_1' \lesssim t_1 \quad t_2 \lesssim t_2'}{t_1 \multimap t_2 \lesssim t_1' \multimap t_2'} \quad \frac{t \lesssim t'}{!t \lesssim !t'}$$

$$\frac{t \lesssim t_1 \multimap t_2}{!t \lesssim t_1 \multimap t_2} \quad \frac{t \lesssim t_1 \otimes t_2}{!t \lesssim t_1 \otimes t_2} \quad \frac{t \lesssim \star}{!t \lesssim \star}$$

Fig. 10. Compatibility with shortcut subtyping

For these casts to be admissible in a program, the compatibility relation needs to be amended to reflect $!t <: t$ subtyping. By including this subtyping relation, compatibility is no longer symmetric, but it turns into a reflexive, anti-symmetric relation. Also the rule for two compatible function types is now contravariant in the argument type. Compatibility is not transitive to rule out casts like $\langle \star \Leftarrow t_1 \otimes t_2 \rangle$ that always fail. Compatibility ensures that casts only fail when trying to unwrap a dynamic type. Figure 10 shows the updated definition of the compatibility relation. It strictly encompasses the original notion of compatibility. The typing rule for casts must be updated accordingly.

$$\frac{\Theta; \Gamma \vdash e : t_2 \quad t_2 \lesssim t_1}{\Theta; \Gamma \vdash \langle t_1 \Leftarrow t_2 \rangle^p e : t_1}\ \mathit{Cast}'$$

With this extended framework in place, we can include a simplification rule for casts as follows:

$$\langle t_1 \Leftarrow t_2 \rangle^p \langle t_2 \Leftarrow t_3 \rangle^q e \Longrightarrow \langle t_1 \Leftarrow t_3 \rangle^{pq} e \qquad \text{if } t_3 \lesssim t_1$$

The blame labels for the two casts have to be combined to cater for a sequence of casts like

$$\langle \mathit{int} \otimes \mathit{int} \Leftarrow \mathit{int} \otimes D \rangle^p \langle \mathit{int} \otimes D \Leftarrow D \otimes D \rangle^q e$$

where both steps involve an unwrapping of a dynamic type, each of which may have to be attributed to a different part of the program, that is, either p or q.

Theorem 2. *Type soundness holds for the extended calculus with the Cast′ rule and the modified operational semantics.*

Proof. By extending the proofs of type preservation and progress with the cases for the four modifications of the reduction rules.

Working towards a blame theorem, the subtyping relations of Figure 8 extend in a similar way as the compatibility relation. It is sufficient to add the rules that drop the exponential as long as the target type is neither dynamic nor another exponential. This choice also keeps the subtyping relations deterministic. As these rules are identical for all three relations, $<:$, $<:^+$, and $<:^-$, we only show them for one relation.

$$\frac{t <:^+ \star}{!t <:^+ \star} \qquad \frac{t <:^+ t_1' \multimap t_2'}{!t <:^+ t_1' \multimap t_2'} \qquad \frac{t <:^+ t_1' \otimes t_2'}{!t <:^+ t_1' \otimes t_2'}$$

The lemmas in Section 4 still hold for the extended subtyping relations. Also the establishment of the blame theorem and its preliminaries carry over by extending the proofs with the four cases of the modified reduction rules.

7 Related Work

We based our work on Turner and Wadler's linear lambda calculus [21] and extended it with recursion as in Lily [6]. We considered other versions of linear lambda calculus as alternative starting points (for example [2–4]). However, we chose Turner and Wadler's because its formalization of linear values as heap references makes linearity very explicit.

Affine types are closely related to linear types. An affine value must be eliminated at most once, but it may also be discarded. The results of the paper extend readily to affine systems. Again we refer to the technical report for details.

In the introduction, we refer to a number of papers on statically typed languages with type `Dynamic`. Actually, there are different flavors of such languages. The first (earlier) flavor is the one treated by Abadi and coworkers [1]. It is not concerned with type casts, but rather has a specific expression to create a dynamic value by pairing a standard value with its static type. The corresponding, type-safe elimination form is a typecase construct that performs pattern matching on the type component and extracts the value in case of a match. There is a bulk of further work in this area, which we choose not to comment on, because we do not consider typecase in this work. We conjecture —based on our results— that linearity is also orthogonal to dynamics in a language with typecase.

The flavor that we are interested in starts with Henglein's investigation of dynamic typing [10], which pioneered the ideas of wrapped values and of safe and unsafe casts in the context of a simply-typed lambda calculus. However, Henglein does not introduce a subtyping relation that would have enabled him to prove a blame theorem.

Taha and Siek [14,16] have proposed the use of subtyping to characterize the potential failure of a cast expression. Their work has been extended by Wadler and Findler with blame assignment, the marking of cast expressions with unique labels to determine the program point that caused a cast failure. On this basis, they proved the blame theorem, which states that errors in a gradually typed program are always blamed on the untyped (dynamic) part of the program.

Henglein [10] introduces the idea of a coercion calculus in order to reduce the number of times a type is tested at run time. This idea has been picked up by a number of researches with different goals: eager reporting of cast errors [15], improving the efficiency of gradual typing [11], providing streamlined data structures and algorithms for representing and normalizing coercions [17]. We only touch on this subject briefly in Section 6 to indicate that our approach is compatible with coercion calculi.

We are not aware of any work that has explored the interaction of linearity and gradual typing.

8 Conclusion

We extended a linearly typed lambda calculus with recursion and type `Dynamic`, proved type soundness for it, and established a blame theorem a la Wadler and Findler. In comparison to Wadler and Findler's work, our calculus is more modular thus making it easy to extend with additional type constructions or with optimizations as shown in Section 6. As an extension, we integrated a notion of subtyping that allows single uses of dynamic replicated values not to require explicit unwrapping of the exponential, but rather just casting from the dynamic type to the linear target type. Thus, we have shown that linear typing and gradual typing with blame assignment are orthogonal aspects in a lambda calculus setting. We have also demonstrated the robustness of the concepts established with the blame theorem by extending the underlying calculus with a notion of subtyping. Although we have considered a core calculus, adding datatypes and conditionals would be straightforward.

There are a number of avenues for further work. It would be interesting to consider the interaction of linearity, polymorphism, and gradual typing to see if the orthogonality found in this work can be sustained. Because of the modularity of our approach, we expect the orthogonality to carry over to the context of session typing and probably also to more general process calculi. Last, instead of having the dynamic type forget the type structure, it could also forget about the linearity restriction. Similar ideas have been pursued by Pucella and Tov [19,20] and they could also be considered for session typing and process calculi.

Acknowledgment. This work was funded by the German Research Foundation, DFG, as part of the priority programme 1496 "Reliably Secure Software Systems RS^3".

References

[1] Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM TOPLAS 13(2), 237–268 (1991)

[2] Abramsky, S.: Computational interpretations of linear logic. Theor. Comput. Sci 111(1&2), 3–57 (1993)

[3] Alves, S., Fernández, M., Florido, M., Mackie, I.: Gödel's system T revisited. Theoretical Computer Science 411(11-13), 1484–1500 (2010)

[4] Benton, P.N., Bierman, G.M., de Paiva, V., Hyland, M.: A term calculus for intuitionistic linear logic. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 75–90. Springer, Heidelberg (1993)

[5] Bierman, G., Meijer, E., Torgersen, M.: Adding dynamic types to C#. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 76–100. Springer, Heidelberg (2010)

[6] Bierman, G.M., Pitts, A.M., Russo, C.V.: Operational properties of Lily, a polymorphic linear lambda calculus with recursion. Electr. Notes Theor. Comput. Sci. 41(3), 70–88 (2000)

[7] Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010)
[8] Castagna, G. (ed.): ESOP 2009. LNCS, vol. 5502. Springer, Heidelberg (2009)
[9] Girard, J.-Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
[10] Henglein, F.: Dynamic typing: Syntax and proof theory. Science of Computer Programming 22, 197–230 (1994)
[11] Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: Trends in Functional Programming (TFP) (2007)
[12] Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. ACM TOPLAS 31, 12:1–12:44 (2009)
[13] Palsberg, J. (ed.): Proc. 37th ACM Symp. POPL, Madrid, Spain. ACM Press (January 2010)
[14] Siek, J.G., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007)
[15] Siek, J.G., Garcia, R., Taha, W.: Exploring the design space of higher-order casts. In: Castagna [8], pp. 17–31
[16] Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (September 2006)
[17] Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Palsberg [13], pp. 365–376
[18] Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
[19] Tov, J.A., Pucella, R.: Stateful contracts for affine types. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 550–569. Springer, Heidelberg (2010)
[20] Tov, J.A., Pucella, R.: Practical affine types. In: Proc. 38th ACM Symp. POPL, Austin, TX, USA, pp. 447–458. ACM Press (January 2011)
[21] Turner, D.N., Wadler, P.: Operational interpretations of linear logic. Theoretical Computer Science 227(1-2), 231–248 (1999)
[22] Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Castagna [8], pp. 1–16
[23] Wright, A., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)
[24] Wrigstad, T., Nardelli, F.Z., Lebresne, S., Östlund, J., Vitek, J.: Integrating typed and untyped code in a scripting language. In: Palsberg [13], pp. 377–388

Higher-Order Size Checking without Subtyping

Attila Góbi[1,⋆], Olha Shkaravska[2], and Marko van Eekelen[2,3,⋆⋆]

[1] Faculty of Informatics, Eötvös Loránd University Budapest
[2] Institute for Computing and Information Sciences, Radboud University Nijmegen
[3] Open University of the Netherlands, Heerlen

Abstract. This work focuses on checking size annotations of higher-order polymorphic functional programs supporting nested lists. These annotations are in a lambda-calculus that formalize non-linear and non-monothonic polynomial relations between the sizes of arguments and those of the corresponding results of functions.

The presented sizing rules straightforwardly define verification condition generation. Since exact size dependencies are considered (i.e. without subtyping), checking of these conditions amounts to checking conditional equations between polynomials.

1 Introduction

Obtaining size information about program expressions is important in the case of resource analysis, where sized types are commonly used to build algorithms to predict resource consumption and termination. It is important to emphasize that the quality of size analysis determines the precision of the prediction.

This paper focuses on checking already given size-aware annotations of higher-order functional programs. To achieve this, a calculus is introduced that formalizes the relations between the sizes of the arguments and those of the results of functions in a higher-order polymorphic functional language. Informally, the calculus extends the lambda-calculus with arithmetic operations and two operators for finite maps: List, that defines (higher-order) finite maps, and Shift, which combines two finite maps.

In our approach, size checking consists of three phases. The first phase is checking whether the corresponding size expression is normalizable, the second one is the verification condition generation, and the third one is checking equations on normalizable lambda-terms. The verification conditions we obtain as the result of the syntax-directed stage of type-checking, are (conditional) equations in the combination of three theories: lambda-calculus, integer ring and finite maps.

In this paper we focus only on the inference of verification conditions. It is assumed that the VCs are solved by an external solver, and we do not reason here about their decidability.

⋆ The first author is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).

⋆⋆ The second and the third author are supported by the Artemis Joint Undertaking in the CHARTER project, grant-nr. 100039.

The ultimate goal of the paper is to continue the series of work on size analysis of first-order strict functional languages, in which annotation inference is based on polynomial interpolation [1]. Checking of interpolated sizes can be done by embedding into sized types. However, using sized types may lead to the need of polymorphism in size variables, which eventually forces the use of subtyping or higher-rank types and type functions. For example, consider the following function written in a Haskell-like language.

```
f :: ([a] -> b) -> (b,b)
f g = (g [], g [1])
```

By extending the original type declaration of `f` with size variables, the result would be $(\mathsf{L}_n(a) \to b) \to (b, b)$, where $\mathsf{L}_n(a)$ means a list of a with size n. The variables in the type are implicitly quantified at the outermost level, resulting in a typing error. The reason for this is that two different list types are applied to the function g (i.e. $\mathsf{L}_0(a)$ and $\mathsf{L}_1(a)$). A common solution is to use subtype polymorphism, where $\mathsf{L}_n(a) \leq \mathsf{L}_m(a)$ iff $n \leq m$. This approach solves the problem of type checking, but can lead to significant overestimation of the sizes.

Another solution would be to use parametric polymorphism, leading to higher-rank types, like $(\forall n.\mathsf{L}_n(a) \to b) \to (b, b)$. However, this type is still not generic enough. For example, the result of the expression `f` (`cons` 1) would be a pair of lists of different sizes, which is impossible, because the type of `f` suggested that the two elements of the pair must have the same size (i.e. the type of the result is (b, b)).

It is possible to generalize the idea further by using dependent types and by introducing a type function to handle the return types, but for a complicated function the type would be even more complicated. Such a complicated example is `t3` where the function takes two arguments and applies the first argument on the second one three times.

```
t3 :: (a -> a) -> a -> a
t3 f x = f (f (f x))
```

The problem here is that arguments of different types may lead to different kinds of polymorphism. The original type, for instance, can be used when the arguments are not sized. On the other hand, if the first argument is of type $\mathsf{L}_n(a) \to \mathsf{L}_{n+1}(a)$, the size of `t3` should be:

`t3` :: $(\forall n.\mathsf{L}_n(a) \to \mathsf{L}_{n+1}(a)) \to \mathsf{L}_n(a) \to \mathsf{L}_{n+3}(a)$

With the use of type functions, we can give correct typing for all different kinds of arguments, but it is troublesome to find a type which covers all the cases.

In this paper, size expressions make size checking fairly manageable and transparent, and can be seen as a light-weight version of the mechanisms discussed above.

This paper is a revised version of our work presented at TFP'11, and published as a non peer-reviewed technical report [2]. Since then, a proof-of-concept implementation[1] has been created, and most of the paper — including the size checking rules — have been rewritten to reflect the implementation.

The rest of the paper is organized as follows. Section 2 describes the syntax of our language, and introduces size expressions that are used to represent size dependencies of functions. Section 3 formalizes verification condition generation. Then, in Section 4, some examples of size checking are given. Section 5 reasons about the termination of verification condition generation. Some details about the current state of the implementation is given in Section 6. Section 7 summarizes related work, while Section 8 concludes, and points out future directions of work.

2 Language Syntax

Consider a higher-order functional language extended with size expressions. Its syntax is given on Fig. 1. In this language, a program consists of top-level bindings. All bindings are annotated with a type (τ) and a size expression (η). Size expressions are discussed in Section 2.1. The underlying type system is supposed to be a Hindley-Milner type system, and it is supposed to have two predefined types: `Int` for integers and $\mathtt{L}(\tau)$ for finite lists.

Program variables	$\in$	x, y, z, f, g, h
Integer literals	$\in$	m, n
Type variables	$\in$	$\alpha,\ \beta,\ \gamma$
Types	$\tau ::=$	$\mathtt{Int} \mid \mathtt{L}(\tau) \mid \alpha \mid \tau_1 \to \tau_2$
Data constructors	$K ::=$	$\mathtt{nil} \mid \mathtt{cons} \mid n$
Expressions	$e ::=$	$x \mid K \mid e_1\ e_2 \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$
	$\mid$	$\mathtt{match}\ e_1\ \mathtt{with}\ \mathtt{nil} \Rightarrow e_2$
		$\qquad\qquad\qquad \mathtt{cons\ hd\ tl} \Rightarrow e_3$
Programs	$prog ::=$	$\epsilon \mid f\ z_1 \ldots z_k :: \tau :: \eta = e;\ prog$

Fig. 1. Syntax

Our language does not allow lambda abstraction and recursive let-expressions, however, recursive functions can be defined as top-level bindings. It is easy to extend the language with recursive let, if we annotate let bindings with size expressions. Similarly, lambda abstraction can be enabled if it has an explicit size annotation. However, this restriction does not affect the expressiveness of the language, so they are omitted for simplicity.

In our examples, we do not use integer arithmetics, so operations of integers are omitted for the sake of simplicity, but it is straightforward to add them. Constructors `nil` and `cons` are, however, made explicit in the abstract syntax, because they play an important role in this paper.

[1] `http://kp.elte.hu/sizechecking`

2.1 Size Calculus

Size expressions represent size dependencies of functions. Their grammar is given on Fig. 2. Size expression is a lambda calculus extended with integer arithmetic and combinators to express the size dependencies of lists. Note that first order size expressions are polynomials that may be non-linear and non-monotonic.

Now a few simple examples are considered to give an idea behind the formalization. Let us begin with an integer literal, e.g. 42. We assume that it does not have a size, so the expression $\mathsf{Unsized}$ is assigned to it.

For arguments of functions, abstractions over size expressions are used. The following binding declares a function which maps everything to 42. Its size expression $(\lambda s_x.\,\mathsf{Unsized})$ mirrors the fact that the size of the result of the function is $\mathsf{Unsized}$ regardless of the size of the argument.

$$\texttt{const}\ x :: \alpha \to \texttt{Int} :: \lambda s_x.\,\mathsf{Unsized} = 42$$

The size of a list is expressed by the combinator List. For instance, the size of the list "[2]" is given by $\mathsf{List}\,1\,(\lambda i.\,\mathsf{Unsized})$. Here the first argument of List denotes the length of the list while the second is a lambda abstraction expressing the sizes of the elements of the list. As the only element of this list is 2, which is unsized, one can say that all the elements of this list are unsized: $\lambda i.\,\mathsf{Unsized}$.

The λ-bound variable i corresponds to the position of the element in a list. For instance, according to the expression $\mathsf{List}\,n\,\eta$, the expression $\eta\,(n-1)$ represents the size of the head, the expression $\eta\,(n-2)$ represents the size of the second element, while the expression $\eta\ 0$ represents the length of the last element[2]. It means that the expression η can be seen as a finite map defined on $0,\dots,n-1$. Some examples to clarify the List combinator:

$$\begin{array}{ll}
[2] & \mathsf{List}\ 1\,(\lambda i.\,\mathsf{Unsized}) \\
[[2]] & \mathsf{List}\ 1\,(\lambda i.\,\mathsf{List}\ 1\,(\lambda j.\,\mathsf{Unsized})) \\
[[2],[]] & \mathsf{List}\ 2\,(\lambda i.\,\mathsf{List}\ i\,(\lambda j.\,\mathsf{Unsized}))
\end{array}$$

Two lists are considered equal if their length are equal, and all of their elements have the same size expression.

$$\mathsf{List}\,s_1\,\eta_1 = \mathsf{List}\,s_2\,\eta_2 \Leftrightarrow s_1 = s_2 \wedge \forall i \in \{0,\dots,s_1-1\}:\ \eta_1\,i = \eta_2\,i$$

It also means that there are several ways to express the empty list, because $\mathsf{List}\,0\,(\lambda i.\,\mathsf{Unsized}) = \mathsf{List}\,0\,(\lambda i.\,\mathsf{List}\,42\,(\lambda j.\,\mathsf{Unsized}))$. In this paper, we usually use the symbol bottom for the size of an element of an empty list: $\mathsf{List}\,0\,(\lambda i.\bot)$.

To express a size expression of a function, in terms of its argument, the abstraction $\widehat{\lambda}^s_p$ is used. In the following example function addone takes its argument

[2] Note that this enumeration of list elements "opposes" the traditional one in functional languages, where the head element has number 0, etc. The enumeration we use is more convenient in our reasoning and, for instance, simplifies significantly the match-rule defined later.

$l : L(\texttt{Int})$ and returns the list ($\texttt{cons}\ 1\ l$). The size expression of the function tells us that the size of the list is incremented by one.

$$\mathsf{addone}\ l :: \mathsf{L}(\texttt{Int}) \to \mathsf{L}(\texttt{Int}) :: \widehat{\lambda}^{s}_{p}.\,\mathsf{List}\,(s+1)(\lambda i.\,\mathsf{Unsized}) = \texttt{cons}\ 1\ l$$

This abstraction is the dual of the List combinator as it can be seen on the reduction rules (Fig. 3). During reductions capture-avoiding substitutions are assumed, i.e. by alpha-renaming or other mechanism.

The combinator Shift is used to concatenate size functions. The expression $\mathsf{Shift}\, e_1\, s\, e_2$ means the size function of the list obtained by inserting the last s elements of e_1 before e_2. That is:

$$\begin{aligned} \eta_1 &= \{0 \mapsto \eta_1^0, 1 \mapsto \eta_1^1, \ldots, n \mapsto \eta_1^n\} \\ \eta_2 &= \{0 \mapsto \eta_2^0, 1 \mapsto \eta_2^1, \ldots, m \mapsto \eta_2^m\} \\ \mathsf{Shift}\,\eta_1\, s\, \eta_2 &= \{0 \mapsto \eta_1^0, 1 \mapsto \eta_1^1, \ldots, s-1 \mapsto \eta_1^{s-1}, s \mapsto \eta_2^0, s+1 \mapsto \eta_2^1, \ldots\} \end{aligned}$$

Size variables	$\in$	s, p
Integer literals	$\in$	n, m
Binary operators ξ	$::=$	$+ \mid - \mid *$
Size expressions η	$::=$	$\mathsf{List} \mid \mathsf{Unsized} \mid \mathsf{Shift} \mid \bot \mid s \mid n$
	$\mid$	$\lambda s.\eta \mid \widehat{\lambda}^{s}_{p}.\eta \mid \eta_1\eta_2 \mid \eta_1\,\xi\,\eta_2$

Fig. 2. Syntax of size expressions

$$\begin{aligned} (\lambda p.\eta_1)(\eta_2) &\to_\beta (\eta_1[p := \eta_2]) \\ (\widehat{\lambda}^{s}_{p}.e)(\mathsf{List}\,\eta_1\,\eta_2) &\to_\beta (e[s := \eta_1, p := \eta_2]) \\ \mathsf{Shift}\,\eta_1\, s\,\eta_2\, i &\to_\beta \begin{cases} \eta_1 i & \text{if } i < s \\ \eta_2(i-s) & \text{otherwise} \end{cases} \end{aligned}$$

Fig. 3. Reduction rules of the size calculus

Now, with the help of the constructs defined above, we can create the size expressions for the predefined functions (Fig. 4).

As a more complicated example, the definition of the function `concat` is shown. This function takes two lists as arguments and concatenates them.

$$\begin{aligned} \texttt{concat x y} &:: \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) \\ &:: \widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x) \\ &= \texttt{match}\ x\ \texttt{with}\ \texttt{nil} \Rightarrow y \\ &\qquad\qquad\qquad \texttt{cons hd tl} \Rightarrow \texttt{cons hd}\ (\texttt{concat tl}\ y) \end{aligned}$$

$$\texttt{nil} :: \mathsf{L}(\alpha) :: \mathsf{List}\ 0\ (\lambda i.\bot) \qquad 0 :: \texttt{Int} :: \mathsf{Unsized}$$
$$\texttt{cons} :: \alpha \to \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) :: \lambda s_x.\ \widehat{\lambda}^{s_l}_{p_l}.\ \mathsf{List}\ (s_l + 1)\ (\mathsf{Shift}\ p_l\, s_l\, (\lambda y.s_x))$$

Fig. 4. Size expressions of the predefined functions

3 Size Analysis

In the language defined, top level bindings are annotated with size expressions. In this section we formalize a way of checking the bodies of top level bindings against their annotations. The verification process consists of two steps. The first step is the verification condition generation, which is a syntax-driven procedure to generate verification conditions. The second step is checking the generated conditions. Checking the generated conditions is not covered in this paper, and it is assumed that an external solver is used.

First of all, an assumption set is defined in Fig. 5. The assumption set is a sequence of elements of the form $D \rightsquigarrow \eta$, where η is a size expression and D is a set of conditions. A $D \rightsquigarrow \eta$ means that the corresponding expression has size η if all of the conditions in D hold. Note that comparison of two size expressions is only possible if they can be reduced to integers.

Assumption set	$\mathcal{C} ::= \epsilon \mid \mathcal{C}, D \rightsquigarrow \eta$
Condition	$D ::= \epsilon \mid D, \eta = 0 \mid D, \eta_1 >= \eta_2 \mid D, \eta_1 < \eta_2$

Fig. 5. Syntax of the assumption set

Before introducing the size checking rules we need a function called MGS, which generates the most generic size expressions containing fresh variables for a given underlying type and a list of bound variables. The formal definition of the function can be found in Fig. 6, and informally, the following examples show what the function does.

$$\begin{aligned}
\textsc{mgs}\big(\mathsf{L}(\texttt{Int});\ []\big) &= \mathsf{List}\ \ s_1\ (\lambda i.\,\mathsf{Unsized})\\
\textsc{mgs}\big(\mathsf{L}(\alpha);\ []\big) &= \mathsf{List}\ \ s_1\ (\lambda i.\ s_2\ i)\\
\textsc{mgs}\big(\mathsf{L}(\mathsf{L}(\texttt{Int}));\ []\big) &= \mathsf{List}\ \ s_1\ \big(\lambda i.\,\mathsf{List}\ (s_2\ i)\ (\lambda j.\,\mathsf{Unsized})\big)\\
\textsc{mgs}\big(\mathsf{L}(\mathsf{L}(\alpha));\ []\big) &= \mathsf{List}\ \ s_1\ \big(\lambda i.\,\mathsf{List}\ (s_2\ i)\ (\lambda j.\ s_3\ i\ j)\big)
\end{aligned}$$

The resulting size expressions contain higher-order free variables, and all of the variables bound internally by lambda abstrations are applied to them. The second parameter is used to propagate an internal state (which contains the variables bound during the traversal of the size expression), hence an empty list is used when the function is called externally. All of the free variables are fresh.

$$\begin{aligned}
\textsc{mgs} &:: \tau \times [\eta] \to \eta \\
\textsc{mgs}(\mathtt{Int};\ \gamma) &= \mathsf{Unsized} \\
\textsc{mgs}(\tau_1 \to \tau_2;\ \gamma) &= \lambda\beta.\, \textsc{mgs}(\tau_2;\ \gamma, \beta) \\
\textsc{mgs}(\mathsf{L}(\tau);\ \gamma) &= \mathsf{List}\ \textsc{mgs}(\beta;\ \gamma)\ \big(\lambda\beta'.\, \textsc{mgs}(\tau;\ \gamma, \beta')\big) \\
\textsc{mgs}(\alpha;\ \eta, \gamma) &= \textsc{mgs}(\alpha;\ \gamma)\ \eta \\
\textsc{mgs}(\alpha;\ []) &= \beta
\end{aligned}$$

(Where β and β' are fresh size variables.)

Fig. 6. Most generic size expression for a given type

The derivation rules for top level bindings are shown in Fig. 7. The judgement $\Gamma \vdash e$ can be read as "in the size environment Γ the program e is well-sized". The size environment Γ is similar to a type environment, it is a map from program variables to size expressions, so it is a sequence of elements of the form $x : \eta$.

The rule Empty ensures that there are no free variables in the size expressions. Recall that it is assumed that function bindings are well-typed in the underlying type system. Then the rule Bind creates fresh size expressions for each argument, saves them in the environment to create an assumption set. In this rule the type τ' cannot be function type. This does not restrict the expressiveness, as a function can always be η converted to suit this. The generation of the assumption set uses the judgement $\Gamma \ \vdash e : \mathcal{C}$, and it is detailed later in this section. The judgement means "in the type environment Γ we assume that expression e has size $\mathcal{C}$". The assumed sizes are used to generate verification condition ($D \ \Vdash \eta_c = \eta'\bar{\eta}$), which is a set of conditional equations.

$$\frac{FV(\Gamma) = \emptyset}{\Gamma \vdash \epsilon}\ \textsc{Empty}$$

$$\frac{\begin{array}{c}\bar{\eta} = \textsc{mgs}(\bar{\tau}, \emptyset) \qquad \Gamma,\, \mathsf{f} : \{\emptyset \rightsquigarrow \eta'\},\ \bar{z} : \{\emptyset \rightsquigarrow \bar{\eta}\}\ \vdash e : \mathcal{C} \\ \forall D \rightsquigarrow \eta_c \in \mathcal{C} : D\ \Vdash \eta_c = \eta'\bar{\eta} \\ \Gamma,\, \mathsf{f} : \{\emptyset \rightsquigarrow \eta'\} \vdash \mathit{prog}\end{array}}{\Gamma \vdash \mathsf{f}\ \mathsf{z}_1 \ldots \mathsf{z}_k :: \tau_1 \to \ldots \to \tau_k \to \tau' :: \eta' = e, \mathit{prog}}\ \textsc{Bind}$$

Fig. 7. Verification condition generation for top level bindings

The rules to obtain the assumption set can be seen in Fig 8. Two of them, namely Var and Let, are familiar from the rules in other type systems. Rules Int, Nil and Cons should be clear as well, they are reformulations of Fig. 4.

The rule App uses the operation $\mathcal{C}_1 \ltimes \mathcal{C}_2$, which is defined as follows.

$$\mathcal{C}_1 \ltimes \mathcal{C}_2 = \{D_1 \cup D_2 \rightsquigarrow \eta_1\eta_2 \mid D_1 \rightsquigarrow \eta_1 \in \mathcal{C}_1,\ D_2 \rightsquigarrow \eta_2 \in \mathcal{C}_2\}$$

The operation combines two assumption sets. Each element of the result of the operation corresponds to one member of both of the two sets, applying the element from the right hand side to the element from the left hand side. The assumption set of the resulting element is the union of the assumption sets of the original elements.

The last and most interesting rule is Match. The first step is to obtain the assumption set for the pattern and for the nil branch. For each element of the assumption set inferred for the pattern we infer an assumption set for the cons branch, i.e. for the assumed size $D_i \rightsquigarrow \eta_i$ an assumed size $\mathcal{C}_i$ is obtained. The reason for this is that the assumed size $\mathcal{C}_i$ is valid only when η_i is not nil. This condition is expressed by the assumptions $(\widehat{\lambda}^s_p.s)\eta_i >= 1$. The result is the union of all cases.

$$\frac{}{\Gamma,\, a\!:\!\mathcal{C} \ \vdash a\!:\!\mathcal{C}}\ \textsc{Var} \qquad \frac{}{\Gamma\ \vdash m\!:\!\{\emptyset \rightsquigarrow \mathsf{Unsized}\}}\ \textsc{Int}$$

$$\frac{}{\Gamma\ \vdash \mathtt{nil}\!:\!\{\emptyset \rightsquigarrow \mathsf{List}\ \ 0\ (\lambda i.\bot)\}}\ \textsc{Nil}$$

$$\frac{}{\Gamma\ \vdash \mathtt{cons}\!:\!\lambda s_x.\ \widehat{\lambda}^{s_l}_{p_l}.\ \mathsf{List}\ (s_l+1)\ (\mathsf{Shift}\ p_l\, s_l\, (\lambda y.s_x))}\ \textsc{Cons}$$

$$\frac{\Gamma\ \vdash e_1\!:\!\mathcal{C}_1 \quad \Gamma\ \vdash e_2\!:\!\mathcal{C}_2}{\Gamma\ \vdash e_1 e_2\!:\!\mathcal{C}_1 \ltimes \mathcal{C}_2}\ \textsc{App} \qquad \frac{\begin{array}{c}\Gamma\ \vdash e_1\!:\!\mathcal{C}_1\\ \Gamma,\, x:\mathcal{C}_1\ \vdash e_2\!:\!\mathcal{C}_2\end{array}}{\Gamma\ \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\!:\!\mathcal{C}_2}\ \textsc{Let}$$

$$\frac{\begin{array}{l}\qquad\Gamma\ \vdash e_1\!:\!\{D_1 \rightsquigarrow \eta_1,\ldots,D_n \rightsquigarrow \eta_n\} \qquad \Gamma\ \vdash e_2\!:\!\mathcal{C}'\\ \forall i=1\ldots n:\ \Gamma,\ \mathtt{hd}\!:\!\{\emptyset \rightsquigarrow (\widehat{\lambda}^s_p.p\,(s-1))\eta_i\},\\ \qquad\qquad\qquad \mathtt{tl}\!:\!\{\emptyset \rightsquigarrow (\widehat{\lambda}^s_p.\,\mathsf{List}\ (s-1)\ p)\eta_i\}\ \vdash e_3\!:\!\mathcal{C}_i\\ \mathcal{C}'' = \cup_{i=1}^n \Big(\big(D_i, (\widehat{\lambda}^s_p.s)\eta_i >= 1 \rightsquigarrow \mathcal{C}_i\big) \cup \big(D_i, (\widehat{\lambda}^s_p.s)\eta_i = 0 \rightsquigarrow \mathcal{C}'\big)\Big)\end{array}}{\begin{array}{l}\Gamma\ \vdash \mathtt{match}\ e_1\ \mathtt{with}\ \ \mathtt{nil} \Rightarrow e_2 \qquad :\mathcal{C}''\\ \qquad\qquad\qquad\qquad\ \mathtt{cons\ hd\ tl} \Rightarrow e_3\end{array}}\ \textsc{Match}$$

Fig. 8. Rules for expressions

Separating the elements of the assumption set means we have to infer the cons branch more than one time. Furthermore the length of the resulting assumption set would be twice as much as the length of the assumption set of the pattern. This exponential behaviour occurs only when match expressions are embedded in patterns (like in the following example), which is fortunately an uncommon construct in practice.

```
let x = match e1 with  nil ⇒ e2
                       cons hd tl ⇒ e3
in match x with  nil ⇒ e2
                 cons hd tl ⇒ e3
```

4 Examples

concat

$$\begin{aligned}
&\mathsf{concat}\ \mathtt{x}\ \mathtt{y} :: \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)\\
&\quad :: \widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x)\\
&\quad = \mathtt{match}\ x\ \mathtt{with}\ \ \mathtt{nil} \Rightarrow y\\
&\qquad\qquad\qquad\quad \mathtt{cons}\ \mathtt{hd}\ \mathtt{tl} \Rightarrow \mathtt{cons}\ \mathtt{hd}\ (\mathsf{concat}\ \mathtt{tl}\ y)
\end{aligned}$$

Here x and y are of type $\mathsf{L}(\alpha)$. According to the Bind rule, we are creating size expressions for the arguments by using the function MGS. Assuming that the generated size expressions are $\mathsf{List}\ s_x\,(\lambda i.p_x\,i)$ and $\mathsf{List}\ s_y\,(\lambda j.p_y\,j)$ for x and y, respectively, we need to prove the following entailment:

$$\begin{aligned}
&\Gamma_1 = \{\\
&\qquad \mathsf{concat} : \{\emptyset \rightsquigarrow \widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x)\},\\
&\qquad \mathtt{x} : \{\emptyset \rightsquigarrow \mathsf{List}\ s_x\,(\lambda i.p_x\,i)\},\qquad \mathtt{y} : \{\emptyset \rightsquigarrow \mathsf{List}\ s_y\,(\lambda j.p_y\,j)\}\\
&\quad \}\\
&\Gamma_1 \vdash \mathtt{match}\ x\ \mathtt{with}\ \ \mathtt{nil} \Rightarrow y \qquad\qquad\qquad\qquad : \mathcal{C}''\\
&\qquad\qquad\qquad\quad \mathtt{cons}\ \mathtt{hd}\ \mathtt{tl} \Rightarrow \mathtt{cons}\ \mathtt{hd}\ (\mathsf{concat}\ \mathtt{tl}\ y)
\end{aligned}$$

To do this, we can calculate the assumed size of the nil branch and the condition using the Var rule:

$$\frac{}{\Gamma_1\ \vdash y : \{\emptyset \rightsquigarrow \mathsf{List}\ s_y\,(\lambda j.p_y\,j)\}} \qquad \frac{}{\Gamma_1\ \vdash x : \{\emptyset \rightsquigarrow \mathsf{List}\ s_x\,(\lambda i.p_x\,i)\}}$$

It is not necessary, but practical to β-reduce the size expressions during the derivation. To save space we will do it so. With the help of the assumed size of x, we can calculate the new elements of the size environment for the cons branch:

$$\mathtt{hd} : \{\emptyset \rightsquigarrow p_x\,(s_x - 1)\},\quad \mathtt{tl} : \{\emptyset \rightsquigarrow \mathsf{List}\,(s_x - 1)\,(\lambda i.p_x\,i)\}$$

Using the App and Var rules multiple times, we can obtain the following entailment for the cons branch.

$$\begin{aligned}
&\{\\
&\quad \mathsf{concat} : \{\emptyset \rightsquigarrow \widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x)\},\\
&\quad \mathtt{x} : \{\emptyset \rightsquigarrow \mathsf{List}\ s_x\,(\lambda i.p_x\,i)\},\qquad \mathtt{y} : \{\emptyset \rightsquigarrow \mathsf{List}\ s_y\,(\lambda j.p_y\,j)\}\\
&\quad \mathtt{hd} : \{\emptyset \rightsquigarrow p_x\,(s_x - 1)\},\quad \mathtt{tl} : \{\emptyset \rightsquigarrow \mathsf{List}\,(s_x - 1)\,(\lambda i.p_x\,i)\}\\
&\}\\
&\quad \vdash \mathtt{cons}\ \mathtt{hd}\ (\mathsf{concat}\ \mathtt{tl}\ y) : \{\\
&\qquad \emptyset \rightsquigarrow \mathsf{List}\ (s_x - 1 + s_y + 1)\ (\\
&\qquad\quad \mathsf{Shift}\ (\mathsf{Shift}\ \ p_y\ \ s_y\ \ p_x)\ (s_x - 1 + s_y)\ (\lambda i.\,(p_x(s-1)))\\
&\qquad)\\
&\quad \}
\end{aligned}$$

The expression $(\widehat{\lambda}^s_p.s)\eta_1$ in the rule Match equals s_x, so finally we can obtain the following assumption set for the body of the function.

$$\begin{aligned}
\mathcal{C}'' = \{ & \\
& \{s_x = 0\} \rightsquigarrow \mathsf{List}\ \ s_y\ (\lambda j.p_y\, j) \\
& \{s_x >= 1\} \rightsquigarrow \mathsf{List}\ (s_x - 1 + s_y + 1) \\
& \qquad (\mathsf{Shift}\ (\mathsf{Shift}\ \ p_y\ \ s_y\ \ p_x)\ (s_x - 1 + s_y)\ (\lambda i.\,(p_x(s-1)))) \\
\} &
\end{aligned}$$

To finish the Bind rule, our last step is to prove the following entailments.

$$\begin{aligned}
& s_x = 0\ \Vdash \mathsf{List}\ \ s_y\ (\lambda j.p_y\, j) = \mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x) \\
& s_x >= 1\ \Vdash \mathsf{List}\ (s_x - 1 + s_y + 1) \\
& \qquad (\mathsf{Shift}\ (\mathsf{Shift}\ \ p_y\ \ s_y\ \ p_x)\ (s_x - 1 + s_y)\ (\lambda i.\,(p_x(s-1)))) = \\
& \qquad \mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x)
\end{aligned}$$

The following verification conditions are obtained.

$$s_x = 0\ \Vdash s_y = s_x + s_y \tag{1}$$

$$s_x = 0, s_i < s_x - 1 + s_y + 1\ \Vdash p_y\, s_i = \mathsf{Shift}\ p_y\, s_y\, p_x\, s_i \tag{2}$$

$$s_x >= 1\ \Vdash s_x - 1 + s_y + 1 = s_x + s_y \tag{3}$$

$$\begin{aligned}
& s_x >= 1, s_i < s_x - 1 + s_y + 1\ \Vdash \\
& \qquad \mathsf{Shift}\ (\mathsf{Shift}\ \ p_y\ \ s_y\ \ p_x)\ (s_x - 1 + s_y)\ (\lambda i.\,(p_x(s-1))) \\
& \qquad = \mathsf{Shift}\ p_y\, s_y\, p_x
\end{aligned} \tag{4}$$

Here, (1) and (3) are both true, and (2) can be reduced as follows.

$$s_x = 0, s_i < s_x - 1 + s_y + 1, s_i < s_y\ \Vdash p_y\, s_i = p_y\, s_i \tag{5}$$

$$s_x = 0, s_i < s_x - 1 + s_y + 1, s_i >= s_y\ \Vdash p_y\, s_i = p_x\,(s_i - s_y) \tag{6}$$

Note that (5) is obviously true and in conditions of (6) are not satisfiable. Equation (4) can be resolved similarly.

t3 In the introduction we considered the function t3. Now we show how to express the size dependency of this function in our language. The size checking is straightforward, so it is left for the reader.

$$\mathsf{t3}\ f\, x :: (\alpha \to \alpha) \to \alpha \to \alpha :: \lambda s.\lambda p.s(s(sp)) = f(f(f\, x))$$

The interesting case is, how this function behaves when applied to different kind of arguments (i.e. a function on lists, a function on integers and a function on functions). The following example demonstrates this.

$$\mathsf{t27addone}\ x :: \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) :: \widehat{\lambda}^s_f.\,\mathsf{List}(s + 27)(\lambda x.\,\mathsf{Unsized}) = (\mathsf{t3}\ \mathsf{t3})\ \mathsf{addone}\ x$$

Using the derivation rules, the assumed size for the application t3 t3 is

$$\{\emptyset \rightsquigarrow \big(\lambda s_f.\lambda s_x.s_f(s_f(s_f s_x))\big)\big(\lambda s_f.\lambda s_x.s_f(s_f(s_f s_x))\big)\}$$

which can be reduced to

$$\{\emptyset \rightsquigarrow \lambda s_f.\lambda s_x.\, \underbrace{s_f(s_f(s_f \ldots (s_f s_x)\ldots)}_{27 \text{ applications of } s_f}\}$$

To continue, we use the fact that the function addone has size $\{\emptyset \rightsquigarrow \eta_{\mathsf{addone}}\}$ where $\eta_{\mathsf{addone}} = \widehat{\lambda}^s_f.\,\mathsf{List}\,(s+1)\,(\lambda x.\,\mathsf{Unsized})$. Assuming that the application of the function MGS gives the size expression $\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized})$ for the argument x, we can apply the APP rule.

$$(\lambda x.\, \underbrace{f_{1:}(f_{1:} \ldots (f_{1:}x)\ldots)}_{27 \text{ applications of } f_{1:}})(\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized})) \rightarrow$$

$$\rightarrow \underbrace{f_{1:}\Big(f_{1:} \ldots \Big((\widehat{\lambda}^s_f.\,\mathsf{List}\,(l+1)\,(\lambda x.\,\mathsf{Unsized}))(\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized}))\Big)\ldots\Big)}_{27 \text{ applications of } f_{1:}} \rightarrow$$

$$\rightarrow \underbrace{f_{1:}\Big(f_{1:} \ldots \Big(\mathsf{List}\,(a+1)\,(\lambda x.\,\mathsf{Unsized})\Big)\ldots\Big)}_{26 \text{ applications of } f_{1:}} \rightarrow^* \mathsf{List}\,(a+27)\,(\lambda x.\,\mathsf{Unsized})$$

The following two definitions can be checked similarly.

$\mathsf{t9addone}\ x :: \mathsf{L}(\alpha) \rightarrow \mathsf{L}(\alpha) :: \widehat{\lambda}^s_f.\,\mathsf{List}\,(s+9)\,\lambda y.\,\mathsf{Unsized} = \mathsf{t3}\ (\mathsf{t3}\ \mathsf{addone})\ x$
$\mathsf{add3}\ x :: \mathtt{Int} \rightarrow \mathtt{Int} :: \lambda x.\,\mathsf{Unsized} = \mathsf{t3}\ \mathtt{succ}\ x$

5 Normalizability

Being an extended lambda calculus, our size expressions are certainly Turing complete. It means that, with the help of some tricky constructs, any function of the language can be translated directly into size expression. The function t3 was an examples of that, as its size expressions and body were the same.

On one hand, it is good, because it proves that our language is able to express the size dependencies of any function. On the other hand, it makes the size analysis undecidable.

Size expressions can be seen as a simplification of the function capturing only the necessary information. This is especially important for a recursive function, as checking recursive size expressions can lead to infinite reduction. In size expressions, recursion can be expressed by using the fixed-point combinator ($Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$. With its help, the body of the function fix can be translated directly to size expression.

$$\mathsf{fix}\ f :: (\alpha \rightarrow \alpha) \rightarrow \alpha :: \lambda s.Ys = f\ (\mathsf{fix}\ f)$$

Using the derivation rules, the assumed size of the function body and the obtained verification condition is the following. In this entailment reduction leads to infinite loop.

$$\{\emptyset \rightsquigarrow s_f(\lambda s.Ys)s_f)\} \qquad \Vdash Ys_f = s_f(Ys_f)$$

Another problem is to ensure that the assumption set (Fig. 5) contains comparisons only between integers. To overcome these problems, we restrict size expressions to normalizable lambda expressions. We know that a typed lambda calculus is normalizable, so to check normalizability, we use a type system.

Similarly to our language, we use a Hindley-Milner type system, extended with the type constructor $\alpha \times \beta$, which creates the product of types α and β. We also assume the usual `Bool`, `Nat` and `Unit` types, and selector functions for pairs (π_1 and π_2) are defined.

5.1 Types of Size Operators

The Unsized can be easily represented by the `Unit` type:

$$\mathsf{Unsized} = \mathsf{unit} : \mathtt{Unit}$$

List corresponds to the data constructor of a pair, which expresses the fact that a size of a list is a pair of the length of the list and a map holding the sizes of the elements of the list.

$$\begin{aligned}\mathsf{List} &= \lambda s f.\ {<}s, f{>}\\ &: \mathtt{Nat} \to (\mathtt{Nat} \to \alpha) \to \mathtt{Nat} \times (\mathtt{Nat} \to \alpha)\end{aligned}$$

The expression $\widehat{\lambda}^s_f.e$ can be rewritten to $\mathsf{Unlist}(\lambda s f.e)$, where Unlist is defined as follows.

$$\begin{aligned}\mathsf{Unlist} &= \lambda f t. f\,(\pi_1 t)\,(\pi_2 t)\\ &: (\mathtt{Nat} \to (\mathtt{Nat} \to \alpha) \to \beta) \to \mathtt{Nat} \times (\mathtt{Nat} \to \alpha) \to \beta\end{aligned}$$

The last thing to do is to define the Shift function:

$$\begin{aligned}\mathsf{Shift} &= \lambda f\,n\,g\,x.\ \mathsf{if}\ x < n\ \mathsf{then}\ f\,x\ \mathsf{else}\ g\,(x - n)\\ &: (\mathtt{Nat} \to \alpha) \to \mathtt{Nat} \to (\mathtt{Nat} \to \alpha) \to \mathtt{Nat} \to \alpha\end{aligned}$$

It is also possible to check the inferred type of the size expression against the type of the function, which can be obtained by the following function.

$$\begin{aligned}\mathrm{ST}(\alpha \to \beta) &= \mathrm{ST}(\alpha) \to \mathrm{ST}(\beta)\\ \mathrm{ST}(\mathsf{L}(\alpha)) &= \mathtt{Nat} \times (\mathtt{Nat} \to \alpha)\\ \mathrm{ST}(\alpha) &= \alpha\\ \mathrm{ST}(\mathtt{Int}) &= \mathtt{Unit}\end{aligned}$$

6 Implementation

The algorithm described in the previous sections is partially implemented in Haskell. Two embedded languages has been created – one to create size expressions and one to define functions. The implementation focuses on the generation of the conditional equations, not on solving them and it is out of the scope of this paper.

The first step is the syntax driven verification condition generation, described in Section 3. The verification conditions are then preprocessed to eliminate lambda expressions. The checker reduces the expressions (hence the size expression must be normalizable), then it eliminates most of the Shift and List rules as it was shown in Section 4. The trivial equations are also dropped at this stage.

The library SBV [3] is used to compile the constraints to Z3 [4], currently it is used only to solve integer constraints. Unfortunately, it is not possible to create higher-order uninterpreted functions in Z3, so the uncompilable expressions are replaced by uninterpreted symbols. This further decreases the expressiveness of the size checker. In practice the normalizability is a much stronger restriction, so the current focus of work is to weaken the normalizability restriction before implementing a better algorithm.

The implementation contains 29 test cases including some non-linear and higher-order examples (e.g. map or zipWith). Running the whole test suite takes less then 3 second on a computer equipped with a 2.66GHz Intel Core2 processor.

7 Related Work

Paper [5,6] are the closest to our approach. In the first one, Brady and Hammond have created a dependently typed framework to express and check size dependencies of source language functions. Size relations are not expressed in the source language but in the dependently typed framework. There, the types of the dependently typed functions are checked by the type system. In this way they can express arbitrary size relations, but the source language function must be rewritten in such a way that it proves the size relation within the dependently typed framework. This rewriting is performed by translating a function from a source language to the framework. This leads to the type inhabitation problem, so they are using a theorem prover to create the translation.

This paper addressed the same problem, to express and check exact size relations, but in a different way. We do not change the source language function but express its size dependency separately without using dependent types. What Brady and Hammond have created is a method to translate functions and size relations to proofs, which are checked by the type checker of the dependently typed framework. In contrast, our system is a way to check size relations. For this purpose we use an SMT solver.

Sized types [7] were originally used to detect non-termination and deadlocks in a reactive system. They were subsequently developed to ensure space bounds in Embedded ML [8]. Chin and Khoo [9] introduced type inference for sized types using Presburger arithmetics, which leads to linear approximations. *Mini-Agda* [10] integrates sized types and dependent types. In this language, sized types are used to prove termination of programs.

In contrast to size expressions, sized types are utilized to express upper bounds on data structures. Size expressions are, however, designed to describe exact size relations of higher-order functions.

The structure of size expressions in our research is close to the approach of A. Abel [11], who applied sized types for termination analysis of higher-order functional programs. For instance, in his notation sized lists of type A of length $\imath$ are defined as $\lambda\,\imath\,A.\mu^{\imath}.\mathbf{1}+A\times X$, and size expressions are higher-order arithmetic expressions with λ-abstraction as well. The difference is that in that work one uses linear arithmetic over *ordinals*, where ordinals represent zero-order sizes. Moreover, in that research size information is not a stand-alone formalism, but a part of a dependent type system.

Vasconcelos and Hammond [12] go beyond linear arithmetic. For a given higher-order functional program, they obtain a set of first-order arithmetic constraints over unknown cost functions f. Solving these constraints w.r.t. f gives the desired cost estimates of the program. The underlying arithmetic is the arithmetic over naturals, extended with undefined ϵ and unbounded ω values, equipped with a natural linear order. Size expressions admit addition $+$, multiplication $*$ and subtraction of a constant $-n$, thus such expressions are monotonic. Function types are annotated with natural numbers (*latencies*), e.g. $\alpha \xrightarrow{l} \beta$, so it may be conveniently interpreted as an increment in cost consumption, like l clock ticks if the resource of interest is time. Our approach is different in the sense that we aim at expressing size dependencies directly in terms of sizes of inputs, bypassing latencies.

In paper [13] the authors approach complexity analysis of an imperative language, which is a version of Gödel's T. This is done via abstract interpretation of programs in a semiring of matrices.

In a recent paper [14] the authors develop amortized cost analysis for a higher-order functional language *Schopenhauer*. The analysis is generic, viz. it is applicable to different sorts of resources: heap usage, stack size and the number of function calls. The type-derivation procedure generates linear constraints, solving of which gives the desired upper bounds. The analysis always succeeds, if bounds are linear. So far, the methodology does not support polymorphic recursion.

The *COSTA System* [15] is able to infer upper bounds for object-oriented bytecode. They infer linear size relations among program variables at different program points.

In the first-order language *Safe* [16], the authors use region inference to predict the upper bounds on heap and stack consumption. They assume that upper bounds on sizes of all expressions of *Safe* functions are known. To obtain the upper bounds, they use a term rewriting system [17].

In paper [18] the authors created an amortized analysis for the language Resource Aware ML, which is a first-order fragment of the OCAML language. They can infer multivariate cost characteristics. A similarity of that paper to ours is that they also cope with nested lists.

8 Conclusion and Future Plans

We have presented a formalism that can be used to express size dependencies of higher-order functions in a higher-order polymorphic language. The presented size expressions do not use subtyping: they are to express exact size relations. Size expressions are based upon the lambda-calculus extended with arithmetic operations and special operators for finite maps representing sizes of the elements of lists.

Syntax driven derivation rules are introduced which can be used to implement an algorithm to generate verification conditions. The size expressions are generally not normalizable, but we have shown a sufficient condition to ensure normalizability. Normalizability is required (but not sufficient) to ensure the termination of the verification condition solver. One of the most important future work is to weaken this restriction, and improve the current solver.

A basic implementation has been created as an embedded language in Haskell, which is able to check most of the normalizable expressions. A future direction is to find a way to express the size of any algebraic data types. A possible solution is to make the embedded language extensible, so programmers can define the size relations of their own type.

While designing our size calculus, our aim was to make it easier to identify normalisable fragments and decidable fragments in the size calculus. In this paper we have laid the foundation for that. The actual identification of such fragments is ongoing work. Moreover, future work will focus on providing size inference for a higher-order functional language, where the inference is based on polynomial interpolation. Size expressions are planned to be used to describe the inferred size dependencies of the functions, and to check the inferred size expressions. We believe that checking the verification conditions can be made decidable by adding syntactical restrictions, similarly to [1].

References

1. Shkaravska, O., van Eekelen, M.C.J.D., van Kesteren, R.: Polynomial size analysis of first-order shapely functions. Logical Methods in Computer Science 5(2) (2009)
2. Góbi, A., Shkaravska, O., van Eekelen, M.: Size analysis of higher-order functions. In: Peña, R., van Eekelen, M. (eds.) Proceedings of the 12th International Symposium on Trends in Functional Programming, TFP 2011. LNCS, vol. 7193, pp. 77–91 (2011); Also: Tech. Rep. SIC-07/11
3. Erkok, L.: SMT Based Verification: Symbolic Haskell theorem prover using SMT solving
4. de Moura, L., Bjørner, N.S.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. Brady, E., Hammond, K.: A dependently typed framework for static analysis of program execution costs. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 74–90. Springer, Heidelberg (2006)

6. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 133–144. ACM, New York (2008)
7. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1996, St. Petersburg Beach, Florida, USA, pp. 410–423. ACM (1996)
8. Hughes, J., Pareto, L.: Recursion and dynamic data structures in bounded space: Towards embedded ML programming. In: Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming, ICFP 1999, Paris, France, pp. 70–81. ACM (1999)
9. Chin, W.N., Khoo, S.C.: Calculating sized types. Higher-Order and Symbolic Computation 14, 261–300 (2001)
10. Abel, A.: Miniagda: Integrating sized and dependent types. In: Bove, A., Komendantskaya, E., Niqui, M. (eds.) Workshop on Partiality And Recursion in Interative Theorem Provers, PAR 2010, Satellite Workshop of ITP 2010 at FLoC (2010)
11. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. PhD thesis, Ludwig-Maximilians University, Munich (2006)
12. Vasconcelos, P.B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 86–101. Springer, Heidelberg (2004)
13. Avery, J., Kristiansen, L., Moyen, J.Y.: Static complexity analysis of higher order programs. In: van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2009. LNCS, vol. 6324, pp. 84–99. Springer, Heidelberg (2010)
14. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. SIGPLAN Not. 45, 223–236 (2010)
15. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science 413(1), 142–159 (2012); Quantitative Aspects of Programming Languages (QAPL 2010)
16. Montenegro, M., Peña, R., Segura, C.: A space consumption analysis by abstract interpretation. In: van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2009. LNCS, vol. 6324, pp. 34–50. Springer, Heidelberg (2010)
17. Lucas, S., Pena, R.: Rewriting techniques for analysing termination and complexity bounds of safe programs. In: Proc. Logic-Based Program Synthesis and Transformation, LOPSTR, vol. 8, pp. 43–57 (2008)
18. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011 (2011) (to appear)

Well-Typed Islands Parse Faster

Erik Silkensen and Jeremy Siek

University of Colorado
Boulder, CO, USA
{erik.silkensen,jeremy.siek}@colorado.edu

Abstract. This paper addresses the problem of specifying and parsing the syntax of domain-specific languages (DSLs) in a modular, user-friendly way. We want to enable the design of *composable* DSLs that combine the natural syntax of external DSLs with the easy implementation of internal DSLs. The challenge in parsing these DSLs is that the composition of several languages is likely to contain ambiguities. We present the design of a system that uses a type-oriented variant of island parsing to efficiently parse the syntax of composable DSLs. In particular, we argue that the running time of type-oriented island parsing doesn't depend on the number of DSLs imported. We also show how to use our tool to implement DSLs on top of a host language such as Typed Racket.

1 Introduction

Domain-specific languages (DSLs) provide high productivity for programmers in many domains, such as computer systems, linear algebra, and other sciences. However, a series of trade-offs face the prospective DSL designer today. On one hand, many general-purpose languages include a host of tricks for implementing *internal*, or embedded DSLs, e.g., templates in C++ [2], macros in Scheme [25], and type classes in Haskell [11]. These features allow DSL designers to take advantage of the underlying language and to enjoy an ease of implementation. However, the resulting DSLs are often leaky abstractions, with a syntax that is not quite right, compilation errors that expose the internals of the DSL, and a lack of diagnostic tools that are aware of the DSL [21]. On the other hand, one may choose to implement their language by hand or with parser generators *à la* YACC. The resulting *external* DSLs achieve a natural syntax and often provide more friendly diagnostics, but come at the cost of interoperability issues [3] and an implementation that requires computer science expertise.

In this paper, we make progress towards combining the best of both worlds into what we call *composable* DSLs. Since applications routinely use multiple DSLs, our goal is to enable fine-grained mixing of languages with the natural syntax of external DSLs and the interoperability of internal DSLs. At the core of this effort is a parsing problem: although the grammar for each DSL in use may be unambiguous, programs, such as the one in Figure 1, need to be parsed using the union of their grammars, which is likely to contain ambiguities [14]. Instead of relying on the grammar author to resolve them (as in the LALR tradition), the parser for such an application must efficiently deal with ambiguities.

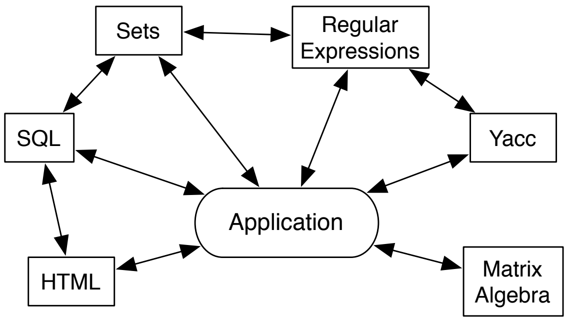


Fig. 1. Our common case: an application using many DSLs

We should emphasize that our goal is to create a parsing system that provides much more syntactic flexibility than is currently offered through operator overloading in languages such as C++ and Haskell. However, we are not trying to build a general purpose parser; that is, we are willing to place restrictions on the allowable grammars, so long as those restrictions are easy to understand (for our users) and do not interfere with composability.

As a motivating example, we consider an application that imports DSLs for matrix algebra, sets, and regular expressions. Suppose the grammars for these languages are written in the traditional style, including the following rules, with associativity specified separately.

```
Expr ::= Expr "+" Expr | Expr "-" Expr
              (Matrix DSL)

Expr ::= Expr "+" Expr | Expr "-" Expr    Expr ::= Expr "+"
              (Set DSL)                          (Regexp DSL)
```

The union of these individually unambiguous grammars is greatly ambiguous, so importing them can increase the parse time by orders of magnitude without otherwise changing programs containing expressions such as `A + B + C`. An obvious fix is to merge the grammars and refactor to remove ambiguity. However, that would require coordination between the DSL authors which is not scalable.

1.1 Type-Oriented Grammars

To address the problem of parsing composed DSLs, we observe that different DSLs typically define different types. We suggest an alternate style of grammar organization inspired by Sandberg [20] that we call *type-oriented grammars*. In this style, a DSL author creates one nonterminal for each type in the DSL and uses the most specific nonterminal/type for each operand in a grammar rule. For example, the above `Expr` rules would instead be written

```
Matrix ::= Matrix "+" Matrix | Matrix "-" Matrix

Set ::= Set "+" Set | Set "-" Set    Regexp ::= Regexp "+"
```

1.2 Type-Based Disambiguation

While we can rewrite the DSLs for matrix algebra, regular expressions, and sets to be type oriented, programs such as $\texttt{A} + \texttt{B} + \texttt{C} \cdots$ are still highly ambiguous if the variables `A`, `B`, and `C` can each be parsed as either `Matrix`, `Regexp`, or `Set`. Many prior systems [17, 5] use chart parsing [15] or GLR [26] to produce a parse forest and then type check to filter out the ill-typed trees. This solves the ambiguity problem, but these parsers are still inefficient on ambiguous grammars because of the large number of parse trees in the forest (see Section 4).

This is where our contribution comes in: *island parsing with eager, type-based disambiguation* is able to efficiently parse programs that simultaneously use many DSLs. We use a chart parsing strategy, called island parsing [23] (or bidirectional bottom-up parsing [19]), that enables our algorithm to grow parse trees outwards from what we call *well-typed terminals*. The statement

$$\textbf{declare}\ \texttt{A:Matrix, B:Matrix, C:Matrix}\ \{\ \ldots\}$$

gives the variables `A`, `B`, and `C` the type `Matrix` and brings them into scope for the region of code within the braces. We integrate type checking into the parsing process to prune ill-typed parse trees before they have a chance to grow, drawing inspiration from the field of natural language processing, where *selection restriction* uses types to resolve ambiguity [13].

Our approach does not altogether prohibit grammar ambiguities; it strives to remove ambiguities from the common case when composing DSLs so as to enable efficient parsing.

1.3 Contributions

1. We present the first parsing algorithm, *type-oriented island parsing* (Section 3), whose time complexity is *constant* with respect to (i.e., independent of) the number of DSLs in use, so long as the nonterminals of each DSL are largely disjoint (Section 4).
2. We present our extensible parsing system that adds several features to the parsing algorithm to make it convenient to develop DSLs on top of a host language such as Typed Racket [24] (Section 5).
3. We demonstrate the utility of our parsing system with examples (included along with the implementation available on Racket's PLaneT package repository) in which we embed syntax for DSLs in Typed Racket.

Section 2 introduces the basic definitions and notation used in the rest of the paper. We discuss our contributions in relation to the prior literature and conclude in Section 6.

2 Background

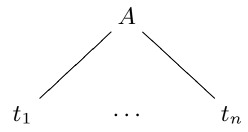
We review the definition of a grammar and parse tree and present a framework for comparing parsing algorithms based on the parsing schemata of Sikkel [22].

2.1 Grammars and Parse Trees

A *context-free grammar* (CFG) is a 4-tuple $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ where Σ is a finite set of terminals, Δ is a finite set of nonterminals, $\mathcal{P}$ is finite set of grammar rules, and S is the start symbol. We use a, b, c, and d to range over terminals and A, B, C, and D to range over nonterminals. The variables X, Y, Z range over symbols, that is, terminals and nonterminals, and $\alpha, \beta, \gamma, \delta$ range over sequences of symbols. Grammar rules have the form $A \to \alpha$. We write $\mathcal{G} \cup (A \to \alpha)$ as an abbreviation for $(\Sigma, \Delta, \mathcal{P} \cup (A \to \alpha), S)$.

We are ultimately interested in parsing programs, that is, converting token sequences into abstract syntax trees. So we are less concerned with the recognition problem and more concerned with determining the parse trees for a given grammar and token sequence. The *parse trees* for a grammar $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$, written $\mathcal{T}(\mathcal{G})$, are trees built according to the following rules.

1. If $a \in \Sigma$, then a is a parse tree labeled with a.
2. If $t_1, \ldots, t_n$ are parse trees labeled $X_1, \ldots, X_n$ respectively, $A \in \Delta$, and $A \to X_1 \ldots X_n \in \mathcal{P}$, then the following is a parse tree labeled with A.



We sometimes use a horizontal notation $A \to t_1 \ldots t_n$ for parse trees and we often subscript parse trees with their labels, so t_A is parse tree t whose root is labeled with A. We use an overline to represent a sequence: $\overline{t} = t_1 \ldots t_n$.

The *yield* of a parse tree is the concatenation of the labels on its leaves:

$$\begin{aligned} \textit{yield}(a) &= a \\ \textit{yield}([A \to t_1 \ldots t_n]) &= \textit{yield}(t_1) \ldots \textit{yield}(t_n) \end{aligned}$$

Definition 2.1. The set of parse trees for a CFG $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ and input w, written $\mathcal{T}(\mathcal{G}, w)$, is defined as follows

$$\mathcal{T}(\mathcal{G}, w) = \{t_S \mid t_S \in \mathcal{T}(\mathcal{G}) \text{ and } \textit{yield}(t_S) = w\}$$

2.2 Parsing Algorithms

We wish to compare the essential characteristics of several parsing algorithms without getting distracted by implementation details. Sikkel [22] introduces a high-level formalism for presenting and comparing parsing algorithms, called *parsing schemata*, that presents each algorithm as a deductive system. We loosely follow his approach, but make some minor changes to better suit our needs.

Each parsing algorithm corresponds to a deductive system with judgments of the form $H \vdash \xi$, where ξ is an *item* and H is a set of items. An item has the form $[p, i, j]$ where p is either a parse tree or a partial parse tree and the integers i and j mark the left and right extents of what has been parsed so far.

$$\text{(BU)}\,\dfrac{\text{(Fnsh)}\,\dfrac{\text{(BU)}\,\dfrac{\text{(Hyp)}\,\dfrac{[\mathtt{A},0,1]\in H}{H\vdash[\mathtt{A},0,1]} \qquad \mathtt{E}\to\mathtt{A}\in\mathcal{P}}{H\vdash[\mathtt{E}\to\mathtt{.A.},0,1]}}{H\vdash[\mathtt{E}\to\mathtt{A},0,1]} \qquad \mathtt{E}\to\mathtt{E+E}\in\mathcal{P}}{H\vdash[\mathtt{E}\to\mathtt{.[E}\to\mathtt{A].\ +\ E},0,1]}$$

Fig. 2. A partial (bottom-up Earley) derivation of the parse tree for `"A + B"`, having parsed `"A "` but not yet `"+ B"`

The set of *partial parse trees* is defined by the following rule: if $A \to \alpha\beta\gamma \in \mathcal{P}$, then $A \to \alpha.\overline{t}_\beta.\gamma$ is a partial parse tree labeled with A, where markers surround a sequence of parse trees for β, while α and γ remain to be parsed. We reserve the variables s and t for parse trees, not partial parse trees. A *complete parse* of an input w of length n is a derivation of $H_0(w) \vdash [t_S, 0, n]$, where $H_0(w)$ is the initial set of items that represent the result of tokenizing the input w.

$$H_0(w) = \{[w_i, i, i+1] \mid 0 \leq i < |w|\}$$

Definition 2.2. A bottom-up variation [22] of the Earley algorithm [8] applied to a grammar $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ is defined by the following deductive rules.

$$\text{(Hyp)}\,\dfrac{\xi \in H}{H \vdash \xi} \qquad \text{(Fnsh)}\,\dfrac{H \vdash [A \to .\overline{t}_\alpha., i, j]}{H \vdash [A \to \overline{t}_\alpha, i, j]}$$

$$\text{(BU)}\,\dfrac{H \vdash [t_X, i, j] \qquad A \to X\beta \in \mathcal{P}}{H \vdash [A \to .t_X.\beta, i, j]}$$

$$\text{(Compl)}\,\dfrac{H \vdash [A \to .\overline{s}_\alpha.X\beta, i, j] \qquad H \vdash [t_X, j, k]}{H \vdash [A \to .\overline{s}_\alpha t_X.\beta, i, k]\}}$$

Example 2.1. Figure 2 shows the beginning of the bottom-up Earley derivation of a parse tree for `A + B` with the grammar:

```
E ::= E "+" E | "A" | "B"
```

3 Type-Oriented Island Parsing

The essential ingredients of our parsing algorithm are type-based disambiguation and island parsing. In Section 4, we show that an algorithm based on these two ideas parses with time complexity that is independent of the number of DSLs in use, so long as the nonterminals of the DSLs are largely disjoint. (We also make this claim more precise.) But first, in this section we introduce our type-oriented island parsing algorithm (TIP) as an extension of the bottom-up Earley algorithm (Definition 2.2).

Island parsing [23] is a bidirectional, bottom-up parsing algorithm that was developed in the context of speech recognition. In that domain, some tokens can be identified with a higher confidence than others. The idea of island parsing is to begin the parsing process at the high confidence tokens, the so-called islands, and expand the parse trees outward from there.

Our main insight is that if our parser can be made aware of variable declarations, and if a variable's type corresponds to a nonterminal in the grammar, then each occurrence of a variable can be treated as an island. We introduce the following special form for declaring a variable a of type A that may be referred to inside the curly brackets.

$$\textbf{declare}\ a : A\ \{\ldots\}$$

Specifically, if $t_X \in \mathcal{T}(\mathcal{G} \cup \{A \to a\})$, then the following is a parse tree in $\mathcal{T}(\mathcal{G})$.

$$X \to \textbf{declare}\ a : A\ \{t_X\}$$

To enable temporarily extending the grammar during parsing, we augment the judgments of our deductive system with an explicit parameter for the grammar. So judgments now have the form

$$\mathcal{G}; H \vdash \xi$$

This adjustment also enables the import of grammars from different modules.

We define the parsing rule for the **declare** form as follows.

$$(\textsc{Decl})\ \frac{\mathcal{G} \cup (A \to a); H \vdash [t_X, i+5, j]}{\mathcal{G}; H \vdash [X \to \textbf{declare}\ a : A\ \{t_X\}, i, j+1]}$$

Note the $i + 5$ accounts for "**declare** $a : A$ {" and $j + 1$ for "}".

Next we replace the bottom-up rule (BU) with the following (BU-Islnd) rule. The (BU-Islnd) rule is no different than the (BU) rule when $X \in \Delta$, except that X can now appear anywhere on the right-hand side. When $X \in \Sigma$, however, we require that α and β are both sequences of terminals.

$$(\textsc{BU-Islnd})\ \frac{\begin{array}{c}\mathcal{G}; H \vdash [t_X, i, j] \\ A \to \alpha X \beta \in \mathcal{P} \quad \mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S) \\ X \in \Sigma \implies \neg\exists k.\ \alpha_k \in \Delta \lor \beta_k \in \Delta\end{array}}{\mathcal{G}; H \vdash [A \to \alpha . t_X . \beta, i, j]}$$

This restriction ensures that when $X \in \Sigma$, the (BU-Islnd) rule only triggers the formation of an island using grammar rules that arise from variable declarations and literals (constants) defined in a DSL; by allowing α and β to be nonempty, we support literals defined by more than one token. For example, the (BU-Islnd) rule doesn't apply when X = `"+"` in `E ::= E "+" E`. In this case, the grammar rule is not defining a variable declaration or constant, and only the `E`'s on either side of the `"+"` give *type* information, so we shouldn't start parsing from `"+"`. We motivate and discuss this rule further in Section 4.

Finally, because islands appear in the middle of the input string, we need both left and right-facing versions of the (Compl) rule.

$$(\textsc{RCompl})\frac{\mathcal{G}; H \vdash [A \to \alpha.\overline{s}_\beta.X\gamma, i, j] \qquad \mathcal{G}; H \vdash [t_X, j, k]}{\mathcal{G}; H \vdash [A \to \alpha.\overline{s}_\beta t_X.\gamma, i, k]\}}$$

$$(\textsc{LCompl})\frac{\mathcal{G}; H \vdash [t_X, i, j] \qquad \mathcal{G}; H \vdash [A \to \alpha X.\overline{s}_\beta.\gamma, j, k]}{\mathcal{G}; H \vdash [A \to \alpha.t_X\overline{s}_\beta.\gamma, i, k]\}}$$

Definition 3.1. The *type-oriented island parsing* algorithm is defined as the deductive system comprised of the rules (Hyp), (Fnsh), (Decl), (BU-Islnd), (RCompl), and (LCompl).

The TIP algorithm is correct in that it can derive a tree for an input string if and only if there is a valid parse tree whose yield is the input string.

Theorem 3.1. For some i and j, $\mathcal{G}; H_0(\mathit{yield}(t_X)) \vdash [t_X, i, j]$ iff $t_X \in \mathcal{T}(\mathcal{G})$.

Proof. By induction on derivations (soundness) and trees (completeness).

The implementation of our algorithm explores derivations in order of *most specific first*, which enables parsing of languages with overloading (and parameterized rules, as in Section 5.2). For example, consider the following rules with an overloaded `+` operator.

```
Float ::= Float "+" Float | Int    Int ::= Int "+" Int
```

The program `1 + 2` can be parsed at least three different ways: with zero, one, or two coercions from `Int` to `Float`. Our algorithm returns the parse with no coercions, which we call most specific: $\texttt{Int} \to [\texttt{Int} \to 1] \;+\; [\texttt{Int} \to 2]$

Definition 3.2. If $B \to A \in \mathcal{P}$, then we say A *is at least as specific as* B, written $A \geq B$, where $\geq$ is the reflexive and transitive closure of this relation. We extend this ordering to terminals and sequences by defining $a \geq b$ iff $a = b$, $\alpha \geq \beta$ iff $|\alpha| = |\beta|$, and $\alpha_i \geq \beta_i$ for $i \in \{1, \ldots, |\alpha|\}$. A parse tree $A \to \overline{s}_\alpha$ is at least as specific as another parse tree $B \to \overline{t}_\beta$ iff $A \geq B$ and $\overline{s}_\alpha \geq \overline{t}_\beta$.

We implement this strategy by comparing the parse trees for a part of the input (e.g., from i to j) and pursuing only the most-specific tree. We save the others on a stack, instead of discarding them as we would for associativity or precedence conflicts (Section 5.1); if the current most-specific parse eventually fails, we pop the stack and resume parsing one of the earlier attempts.

4 Experimental Evaluation

In this section we evaluate the performance of type-oriented island parsing. Specifically, we are interested in the performance of the algorithm for programs that are held constant while the size of the grammar increases.

Chart parsing algorithms have a general worst-case running time of $O(|\mathcal{G}|n^3)$ for a grammar $\mathcal{G}$ and string of length n. In our setting, $\mathcal{G}$ is the union of the grammars for the k DSLs that are in use within a given scope, that is $\mathcal{G} = \bigcup_{i=1}^{k} \mathcal{G}_i$, where $\mathcal{G}_i$ is the grammar for DSL i. We claim that the total size of the grammar $\mathcal{G}$ is not a factor for type-oriented island parsing, and instead the time complexity is $O(mn^3)$ where $m = \max\{|\mathcal{G}_i| \mid 1 \leq i \leq k\}$. This claim deserves considerable explanation to be made precise.

Technically, we assume that $\mathcal{G}$ is *sparse*, which we define as follows.

Definition 4.1. Form a Boolean matrix with a row for each nonterminal and a column for each production rule in a grammar $\mathcal{G}$. A matrix element (i, j) is `true` if the nonterminal i appears on the right-hand side of the rule j, and it is `false` otherwise. We say that $\mathcal{G}$ is *sparse* if its corresponding matrix is sparse, that is, if the number of nonzero elements is on the order of $m + n$ for an $m \times n$ matrix.

We conjecture that, in the common case, the union of many type-oriented grammars (or DSLs) is sparse.

To verify that both the type-oriented style of grammar and the island parsing algorithm are necessary for this result, we show that removing either of these ingredients results in parse times that are dependent on the size of the entire grammar. We consider the performance of the top-down and bottom-up Earley algorithms, in addition to island parsing, with respect to untyped, semi-typed, and type-oriented grammars, which we explain in the following subsections.

We implemented all three algorithms using a chart parsing algorithm, which efficiently memoizes duplicate items. The chart parser continues until it has generated all items that can be derived from the input string. (It does not stop at the first complete parse because it needs to continue to check whether the input string is ambiguous, which means the input would be in error.[1]) Also, we should note that our system currently employs a fixed tokenizer, but that we plan to look into scannerless parsing.

To capture the essential, asymptotic behavior of the parsing algorithms, we count the number of items generated during the parsing of a program with untyped, semi-typed, and typed grammars. For this experiment, the program is the expression `--A`.

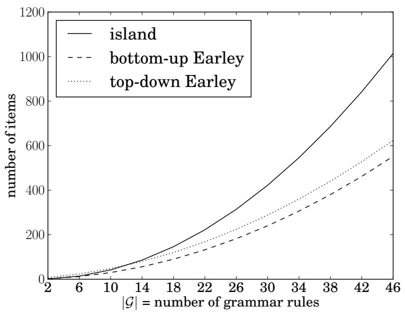
4.1 Untyped Grammar Scaling

In the untyped scenario, all grammar rules are defined in terms of an expression nonterminal (`E`), and variables are simply parsed as identifiers (`Id`).

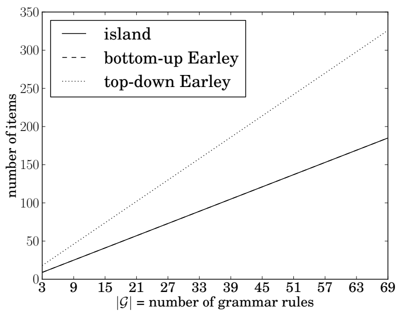
```
E ::= "-" E | Id
```

The results for parsing `--A` after importing k copies of the grammar, for increasing k, are shown in Figure 3(a). The y-axis is the number of items generated by each parsing algorithm and the x-axis is the total number of grammar rules at

[1] While ambiguous input is allowed if there is a single most-specific parse tree, there may be more than one since the $\geq$ relation is not necessarily a total order.



(a) Untyped Grammar



(b) Semi-typed Grammar

Fig. 3. Comparison of Earley and island parsing with two styles of grammars

each k. In the untyped scenario, the size of the grammar affects the performance of each algorithm, with each generating $O(k^2)$ items.

We note that the two Earley algorithms generate about half as many items as the island parser because they are unidirectional (left-to-right) instead of bidirectional.

4.2 Semi-typed Grammar Scaling

In the semi-typed scenario, the grammars are nearly type-oriented: rules are defined in terms of `V` (for vector) and `M` (for matrix); however, variables are again parsed as identifiers. We call this scenario *semi-typed* because it doesn't use variable declarations to provide type-based disambiguation.

```
E ::= V | Mi    V ::= "-" V | Id    Mi ::= "-" Mi | Id
```

The results for parsing `--A` after importing the `V` rules followed by k copies of the `M` rules (i.e., `M1 ::= "-" M1`, `M2 ::= "-" M2`, ...) are shown in Figure 3(b). The lines for bottom-up Earley and island parsing coincide. Each algorithm generates $O(k)$ items, so we see that type-oriented grammars are not, by themselves, enough to achieve constant scaling with respect to grammar size.

We note that the top-down Earley algorithm generates almost twice as many items as the bottom-up algorithms: the alternatives for the start symbol `E` grow with the input length n, which affects the top-down strategy more than the bottom-up strategy.

4.3 Typed Grammar Scaling

The typed scenario is identical to semi-typed except that it no longer includes the `Id` nonterminal. Instead, programs must declare their own typed variables.

```
E ::= V | Mi    V ::= "-" V    Mi ::= "-" Mi
```

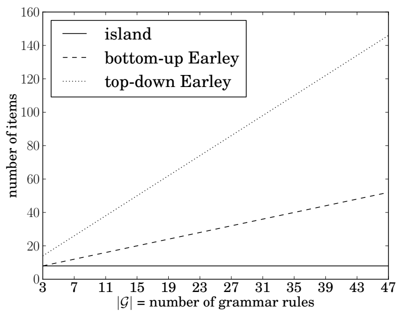


Fig. 4. Comparison of Earley and island parsing with type-oriented grammars

In this scenario, the grammars are sparse and the terminal `V` is well-typed. The results for parsing `--A`, after declaring `A:V` and importing the `V` rules followed by k copies of the `M` rules, are shown in Figure 4. The island parsing algorithm generates a constant number of items as the size of the grammar increases, while the Earley algorithms remain linear. Thus, the *combination* of type-based disambiguation, type-oriented grammars, and island parsing provides a scalable approach to parsing programs that use many DSLs.

4.4 Discussion

The reason type-oriented island parsing scales is that it is more conservative with respect to prediction than either top-down or bottom up, so grammar rules from other DSLs that are irrelevant to the program fragment being parsed are never used to generate items.

In top-down Earley parsing, any grammar rule that produces the nonterminal B, regardless of which DSL it resides in, will be entered into the chart via a top-down prediction rule. Such items have a zero-length extent which indicates that the algorithm does not yet have a reason to believe that this item will be able to complete.

Looking at the (BU) rule of bottom-up Earley parsing, we see that all it takes for a rule (from any DSL) to be used is that it starts with a terminal that occurs in the program. However, it is quite likely that different DSLs will have rules with some terminals in common. Thus, the bottom-up algorithm also introduces items from irrelevant DSLs.

Finally, consider the (BU-Islnd) rule of our island parser. The difference between this rule and (BU) is that it doesn't apply to a terminal on the right-hand side of a grammar rule when it could apply to some other nonterminal (which corresponds to a type) instead. For example, by avoiding the `"-"` in the above grammars, the (BU-Islnd) rule proceeds directly to the rule for `V` without introducing items from DSLs with only `M`i terms that could not complete.

5 A System for Extensible Syntax

In this section we describe the parsing system that we have built as a front end to the Racket programming language. In particular, we describe how we implement four features that are needed in a practical extensible parsing system: associativity and precedence, parameterized grammar rules, grammar rules with variable binders and scope, and rule-action pairs, which combine the notions of semantic actions, function definitions, and macros. We also extend type-oriented grammars so that nonterminals can represent structural types.

Users may define DSLs by writing grammar rules inside a **module** block; the input to our tool consists of programs written in the language of the DSLs that they **import**. For example, one might write

```
module MatrixAlgebra {
  Matrix ::= Matrix "+" Matrix
         ⋮          ⋮
}
```

and then **import** `MatrixAlgebra` in a program using the matrix algebra DSL.

An implementation containing all the features described below is available on Racket's PLaneT package repository. To install, start the Racket interpreter and enter `(require (planet esilkensen/esc))`.

5.1 Associativity and Precedence

Our treatment of associativity and precedence is largely based on that of Visser [27], although we treat this as a semantic issue instead of an optimization issue. From the user perspective, we extend rules (and similarly parse trees) to have the form $A \rightarrow_{\ell,p} \alpha$ where $\ell \in \{\mathsf{left}, \mathsf{right}, \mathsf{non}, \bot\}$ indicates the associativity and $p \in \mathbb{N}_\bot$ indicates the precedence. Concretely, we annotate rules with associativity and precedence inside square brackets in the following way.

```
Matrix ::= Matrix "+" Matrix [left,1]
```

The change to the island parsing algorithm to handle precedence and associativity is straightforward. We simply make sure that a partial parse tree does not violate an associativity or precedence rule before converting it into a (complete) parse tree. We replace the (FNSH) rule with the following.

$$(\textsc{FnshP})\frac{\mathcal{G}; H \vdash [A \rightarrow_{\ell,p} .\overline{t}_\alpha., i, j] \qquad \neg \mathit{conflict}(A \rightarrow_{\ell,p} \overline{t}_\alpha)}{\mathcal{G}; H \vdash [A \rightarrow_{\ell,p} \overline{t}_\alpha, i, j]}$$

Definition 5.1. We say that a parse tree t has a *root priority conflict*, written $\mathit{conflict}(t)$, if one of the following holds.

1. It violates the right, left or non-associativity rules, that is, t has the form
   – $A \rightarrow_{\ell,p} (A \rightarrow_{\ell,p} \overline{t}_{A\alpha})\overline{s}_\alpha$ where $\ell = \mathsf{right}$ or $\ell = \mathsf{non}$; or
   – $A \rightarrow_{\ell,p} \overline{s}_\alpha(A \rightarrow_{\ell,p} \overline{t}_{\alpha A})$ where $\ell = \mathsf{left}$ or $\ell = \mathsf{non}$.
2. It violates the precedence rule, that is, t has the form:

$$t = A \rightarrow_{\ell,p} \overline{s}(B \rightarrow_{\ell',p'} \overline{t})\overline{s'} \text{ where } p' < p.$$

5.2 Parameterized Rules

With the move to type-oriented grammars, the need for parameterized rules immediately arises. For example, consider how one might translate the following grammar rule for conditional expressions into a type-oriented rule.

```
Expr ::= "if" Expr "then" Expr "else" Expr
```

By extending grammar rules to enable the parameterization of nonterminals, we can write the following, where `T` stands for any type/nonterminal.

```
forall T. T ::= "if" Bool "then" T "else" T
```

Parameterized rules have the form $\forall \overline{x}.\, A \to \alpha$, where $\overline{x}$ is a sequence of *variables* (containing no duplicates). We wish to implicitly instantiate parameterized rules, that is, automatically determine which nonterminals to substitute for the parameters. Towards this end, we define a partial function named *match* that compares two symbols with respect to a substitution σ and a sequence of variables and produces a new substitution σ' (if the match is successful). We augment partial parse trees with substitutions to incrementally accumulate the matches, giving them the form $\forall \overline{x}.\, A \to^{\sigma} \alpha.\overline{t}_{\beta}.\gamma$.

Using these definitions, we can implement parameterized rules with a few changes to the base island parser, such as (PRCompl) below.

$$(\textsc{PRCompl})\ \frac{\begin{array}{c}\mathcal{G}; H \vdash [\forall \overline{x}.\, A \to^{\sigma_1} \alpha.\overline{s}_{\beta}.X'\gamma, i, j] \\ \mathcal{G}; H \vdash [t_X, j, k] \qquad \mathit{match}(X', X, \sigma_1, \overline{x}) = \sigma_2\end{array}}{\mathcal{G}; H \vdash [\forall \overline{x}.\, A \to^{\sigma_2} \alpha.\overline{s}_{\beta} t_X.\gamma, i, k]\}}$$

5.3 Grammar Rules with Variable Binders

Consider what would be needed to define a type-oriented grammar rule to parse a `let` expression such as the following, with `n` in scope between the curly brackets.

```
let n = 7 { n * n }
```

We need a way for the rule to refer to the parse tree for `Id` and to say that the identifier has type `T1` inside the brackets. To facilitate such binding forms, we add labeled symbols [12] and a scoping construct [7] to our system.

For example, the `let` rule below binds the variable `x` to the identifier with `x:Id`; the unquoted brackets mark the scoping construct, and `x:T1` says `x` should have type `T1` inside the brackets (any nonempty sequence of bindings may appear before the semicolon):

```
forall T1 T2. T2 ::= "let" x:Id "=" T1 { x:T1; T2 }
```

We implement these rules by parsing in phases, where initially, all regions enclosed in curly brackets are ignored. Once enough surrounding text has been parsed, a region is "opened" and the next phase of parsing begins with an extended grammar. In the `let` example, the grammar is extended with the rule `T1` $\to$ `x` (with `T1` instantiated and `x` replaced by its associated string).

5.4 Rule-Action Pairs

Sandberg [20] introduces the notion of a *rule-action pair*, which pairs a grammar rule with a semantic action that provides code to give semantics to the syntax. In his paper, rule-action pairs behave like macros; we provide ones that behave like functions as well (with call-by-value semantics). Thus users of our system can embed their DSLs in Typed Racket with two kinds of rule-action pairs.

The ⇒ operator defines a *rule-function*: we compile these rules to functions whose parameters are the variables bound on the right-hand side, and whose body is the Typed Racket code after the arrow. Below is an example adapted from Sandberg's paper giving syntax for computing the absolute value of an integer.

```
Integer ::= "|" i:Integer "|" ⇒ (abs i)
```

Similarly, the = operator defines a *rule-macro*: we simply compile a rule-macro to a macro instead of a function. Macros are necessary in some situations. For example, we need a macro to embed the `let` rule, which we can do as follows.

```
forall T1 T2. T2 ::= "let" x:Id "=" e1:T1 { x:T1; e2:T2 } =
  (let: ([x : T1 e1]) e2)
```

We translate DSLs to Typed Racket by generating the appropriate function or macro call for parsed instances of rule-action pairs.

5.5 Structural Nonterminals

We support representations of structural types in type-oriented grammars by enabling the definition of *structural nonterminals*. In our system, the reserved symbol `Type` gives the syntax of types/nonterminals, and the ≡ operator maps parse trees to Typed Racket types.

Users may define structural nonterminals, as long as they are mapped to Typed Racket types, by writing new rules for `Type` inside a **types** block. For example, consider the following rule for a product type.

```
Type ::= T1:Type "×" T2:Type ≡ (Pairof T1 T2)
```

We can then use this syntax in any grammar rules inside the module; for example, we could write the rule below for accessing the first element of a pair.

```
forall T1 T2. T1 ::= p:(T1 × T2) "." "fst" ⇒ (car p)
```

5.6 Examples

Our implementation includes concrete examples of using the features from this section to embed DSLs in the host language Typed Racket. We show how to give ML-like syntax to several operators and forms of Typed Racket, and how to combine this DSL with literal syntax for set and vector operations.[2]

[2] To access the examples, enter `raco docs` at the command line and look under "Parsing Libraries" for the documentation.

6 Related Work and Conclusions

There have been numerous approaches to extensible syntax for programming languages. In this section, we summarize the approaches and discuss how they relate to our work. We organize this discussion in a roughly chronological order.

Aasa et al. [1] augments the ML language with extensible syntax for dealing with algebraic data types. They develop a generalization of the Earley algorithm that performs Hindley-Milner type inference during parsing. However, Pettersson and Fritzson [18] report that the algorithm was too inefficient in practice. Pettersson and Fritzson [18] build a more efficient system based on LR(1) parsing. Of course, LR(1) parsing is not suitable for our purposes because LR(1) is not closed under union, which we need to compose DSLs. Several later works also integrate type inference into the Earley algorithm [16, 28]. It may be possible to integrate these ideas with our approach to handle languages with type inference.

Several extensible parsing systems use Ford's Parsing Expression Grammar (PEG) formalism [9]. PEGs are stylistically similar to CFGs; however, PEGs avoid ambiguity by introducing a prioritized choice operator for rule alternatives and PEGs disallow left-recursive rules. We claim that these two restrictions are not appropriate for composing DSLs. The order in which DSLs are imported should not matter and DSL authors should be free to use left recursion if that is the most natural way to express their grammar.

The MetaBorg [5] system provides extensible syntax in support of embedding DSLs in general purpose languages. MetaBorg is built on the Stratego/XT toolset which in turn used the syntax definition framework SDF [10]. SDF uses scannerless GLR to parse arbitrary CFGs. The MetaBorg system performs type-based disambiguation after parsing to prune ill-typed parse trees from the resulting parse forest. Thus, the performance of MetaBorg degrades where there is considerable ambiguity. Our treatment of precedence and associativity is based on their notion of disambiguation filter [4]. We plan to explore the scannerless approach in the future. Bravenboer and Visser [6] look into the problem of composing DSLs and investigate methods for composing parse tables. We currently do not create parse tables, but we may use these ideas in the future to further optimize the efficiency of our algorithm.

In this paper we presented a new parsing algorithm, *type-oriented island parsing*, that is the first parsing algorithm to be constant time with respect to the size of the grammar under the assumption that the grammar is sparse. (Most parsing algorithms are linear with respect to the size of the grammar.)

We have developed an extensible parsing system that provides a front-end to Typed Racket, enabling the definition of macros and functions together with grammar rules that provide syntactic sugar. Our implementation provides features such as parameterized grammar rules and grammar rules with variable binders and scope.

In the future we plan to both analytically evaluate the performance of our algorithm and to continue testing our hypothesis about the sparsity of the union of several type-oriented DSLs in practice, with larger and more real-world grammars. In our implementation we plan to provide diagnostics for helping programmers resolve remaining ambiguities that are not addressed by typed-based disambiguation.

References

1. Aasa, A., Petersson, K., Synek, D.: Concrete syntax for data objects in functional languages. In: ACM Conference on LISP and Functional Programming, LFP, pp. 96–105. ACM (1988)
2. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. C++ in Depth Series. Addison-Wesley Professional (2004)
3. Beazley, D.M.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Fourth Annual USENIX Tcl/Tk Workshop (1996)
4. den van Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002)
5. Bravenboer, M., Vermaas, R., Vinju, J., Visser, E.: Generalized type-based disambiguation of meta programs with concrete object syntax. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 157–172. Springer, Heidelberg (2005)
6. Bravenboer, M., Visser, E.: Parse Table Composition. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 74–94. Springer, Heidelberg (2009)
7. Cardelli, L., Matthes, F., Abadi, M.: Extensible syntax with lexical scoping. Tech. Rep. 121, DEC SRC (February 1994)
8. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13, 94–102 (1970)
9. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, pp. 111–122. ACM (2004)
10. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism sdf—reference manual—. SIGPLAN Not. 24(11), 43–75 (1989)
11. Hudak, P.: Modular domain specific languages and tools. In: ICSR 1998: Proceedings of the 5th International Conference on Software Reuse, p. 134. IEEE Computer Society, Washington, DC (1998)
12. Jim, T., Mandelbaum, Y., Walker, D.: Semantics and algorithms for data-dependent grammars. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 417–430. ACM, New York (2010)
13. Jurafsky, D., Martin, J.: Speech and Language Processing. Pearson Prentice Hall (2009)
14. Kats, L.C., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010, pp. 918–932. ACM, New York (2010)
15. Kay, M.: Algorithm schemata and data structures in syntactic processing, pp. 35–70. Morgan Kaufmann Publishers Inc., San Francisco (1986)
16. Missura, S.: Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing. Ph.D. thesis, ETH Zurich (1997)

17. Paulson, L.C.: Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer (1994)
18. Pettersson, M., Fritzson, P.: A general and practical approach to concrete syntax objects within ml. In: ACM SIGPLAN Workshop on ML and its Applications (June 1992)
19. Quesada, J.F.: The scp parsing algorithm: computational framework and formal properties. In: Procesamiento del lenguaje natural, vol. (23) (1998)
20. Sandberg, D.: Lithe: a language combining a flexible syntax and classes. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1982, pp. 142–145. ACM, New York (1982)
21. Siek, J.G.: General purpose languages should be metalanguages. In: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, pp. 3–4. ACM, New York (2010), `http://doi.acm.org/10.1145/1706356.1706358`
22. Sikkel, K.: Parsing schemata and correctness of parsing algorithms. Theoretical Computer Science 199(1-2), 87–103 (1998)
23. Stock, O., Falcone, R., Insinnamo, P.: Island parsing and bidirectional charts. In: Conference on Computational Linguistics, COLING, pp. 636–641. Association for Computational Linguistics (1988)
24. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 395–406. ACM, New York (2008)
25. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 132–141. ACM, New York (2011), `http://doi.acm.org/10.1145/1993498.1993514`
26. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence, vol. 2, pp. 756–764. Morgan Kaufmann Publishers Inc., San Francisco (1985)
27. Visser, E.: A case study in optimizing parsing schemata by disambiguation filters. In: International Workshop on Parsing Technology, IWPT 1997, pp. 210–224. Massachusetts Institute of Technology, Boston (1997)
28. Wieland, J.: Parsing Mixfix Expressions. Ph.D. thesis, Technische Universitat Berlin (2009)

Higher-Order Strictness Typing

Sjaak Smetsers[1] and Marko van Eekelen[1,2]
{S.Smetsers,M.vanEekelen}@cs.ru.nl

[1] Institute for Computing and Information Sciences, Radboud University Nijmegen
[2] School of Computer Science, Open University of the Netherlands

Abstract. We extend an existing first-order typing system for strictness analysis to the fully higher-order case. The resulting fully higher-order strictness typing system has an expressivity beyond that of traditional strictness analysis systems.

This extension is developed with the explicit aim to formally prove that the higher-order strictness typing is sound with respect to a natural operational semantics. A key aspect of our approach is that we introduce the proof assistant in an early stage, namely during development of the proof, and as such the language theoretic concepts are designed with the aid of the theorem prover.

The combination of reporting on a new result together with its formal proof, can be seen as a case study towards the achievement of the long term PoplMark Challenge. The proof framework developed for this case study can be used further in other typing systems case studies.

1 Introduction

In this paper we propose the use of proof assistants not only for the reconstruction of hand-written proofs, but also to introduce the tool *during the development* of language theoretic concepts. By introducing the tool in this stage, the consistency of technical concepts can be verified in the process of designing them. This is accomplished by checking properties linking these concepts. This paper and the accompanying proof files comprise examples of such concepts and properties. Inaccuracies or mistakes made during the development were most often detected in an early stage, avoiding time consuming and inevitably failing attempts to construct correctness proof of the main properties. This approach was used for the soundness proof of a non-standard typing system for a simple functional programming language. We combine standard Hindley-Milner typing with strictness information, specifying termination properties of higher-order functions. Strictness information can be used to change inefficient *call-by-need* evaluation into efficient *call-by-value* evaluation. This gain in efficiency lies in the fact that construction of unevaluated expressions (so-called *closures*) is circumvented.

Combining standard typing with some form of input/ouput analysis is quite common. We mention a few examples. *Substructural type systems* ([1]) regulate the order and number of uses of data by ensuring that some values be used at most once, at least once, or exactly once. E.g., linear typing systems (such as uniqueness typing) can be used to identify *unique objects*. These unique objects are suitable for *compile-time garbage collection* which is essential for incorporating destructive updates in functional languages (e.g., See [2,3]). Security-typed languages ([4], for instance) track information flow within programs to enforce security properties such as data confidentiality and integrity. This information can be used to prevent unintentional information leaks. [5] describes a dedicated typing system for predicting the heap space usage of first-order, strict functional programs. This information can be used in several ways, most notably to ensure that a program allocates sufficient free memory.

Another view on this paper is that it reports on a case study of computer aided verification of theories about syntactic objects. Syntactic theories such as *operational semantics* and *type systems* play an important role in the (static) analysis of computer programs and the construction of reliable implementations of programming languages. The usability and reliability of syntactic techniques can undoubtly be improved by using automated proof assistants. This need is recognized by many researchers. Most notably, the PoplMark Challenge [6] calls for experiments on verifications of metatheory and semantics using proof tools. The concrete proposal is to formalize existing proofs of properties of type systems with different proof assistants. The long term goal is far more ambitious. It envisions "... a future in which the papers in conferences such as Principles of Programming Languages (POPL) and the International Conference on Functional Programming (ICFP) are routinely accompanied by mechanically checkable proofs of the theorems they claim."

The contribution of our work is threefold. The first contribution is the formalization of a non-standard typing system for strictness analysis of functional programs. A first-order version of this typing system was presented in [7]. Compared to traditional strictness analyzers, it has two main advantages. Firstly, it can be combined with ordinary typing: the compiler does not require an additional analysis phase. Secondly, it avoids fixed point computations, resulting in a much more efficient implementation. In this paper we augment first order typing with function types, in effect making it fully higher-order. Compared to common strictness analyses, the resulting system has an additional advantage: it permits the specification and derivation of strictness properties *between* the function arguments. We prove that our system is *sound* with respect to a given natural operational semantics. Thereafter, we discuss the extension of the system needed to deal with recursive data-types.

Secondly, it can be seen as a methodological experiment. We assess the usability of theorem provers for formalizing complex semantical issues, not only after the construction of the proofs by hand, but especially during the development of basic theory. The complexity of the typing system in our case study is of a similar level as that of the PoplMark challenge. However, the main proof methods are not known on forehand, as is the case in the challenges, but are to be developed during the proof process.

Finally, the PVS formalization can be used as a framework for developing other metatheoretical concepts. The framework can be used as a basis for developing other type based analyses together with their formal soundness proof, living up to the ambition of the long term PoplMark Challenge.

$$\frac{}{\Box \Downarrow \Box}(\mathit{unit}) \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}(\mathit{abs})$$

$$\frac{M \Downarrow \lambda x.B \qquad B[x \leftarrow N] \Downarrow V}{M\,N \Downarrow V}(\mathit{appl})$$

$$\frac{M \Downarrow \langle X, Y\rangle \quad X \Downarrow V}{\mathsf{fst}\, M \Downarrow V}(\mathit{fst}) \qquad \frac{M \Downarrow \langle X, Y\rangle \quad Y \Downarrow V}{\mathsf{snd}\, M \Downarrow V}(\mathit{snd})$$

$$\frac{}{\langle X, Y\rangle \Downarrow \langle X, Y\rangle}(\mathit{pair})$$

$$\frac{}{\mathsf{inl}\, L \Downarrow \mathsf{inl}\, L}(\mathit{inl}) \qquad \frac{}{\mathsf{inr}\, R \Downarrow \mathsf{inr}\, R}(\mathit{inr})$$

$$\frac{S \Downarrow \mathsf{inl}\, L \qquad G\,L \Downarrow V}{\mathsf{case}\, S\, \mathsf{of}\, G\, \mathsf{or}\, H \Downarrow V}(\mathit{case\text{-}L}) \qquad \frac{S \Downarrow \mathsf{inr}\, R \qquad H\,R \Downarrow V}{\mathsf{case}\, S\, \mathsf{of}\, G\, \mathsf{or}\, H \Downarrow V}(\mathit{case\text{-}R})$$

$$\frac{M[x \leftarrow \mu x.M] \Downarrow V}{\mu x.M \Downarrow V}(\mathit{fix})$$

Fig. 1. Evaluation rules

2 Extended Lambda Calculus

In this section we introduce the core functional language used throughout the paper. This language captures essential aspects such as basic values, abstraction, application, data constructors and destructors, and recursion.

Definition 1. *Let* $\mathbb{V} = \{x, y, z, x_0, x_1, \ldots\}$ *be an infinite set of* term variables

– *The set* Λ *of* (lambda) expressions *is defined by the following abstract syntax.*

$$\Lambda ::= \mathbb{V} \mid \Box \mid \lambda\mathbb{V}.\Lambda \mid \Lambda\,\Lambda \mid \langle\Lambda, \Lambda\rangle \mid \mathsf{fst}\,\Lambda \mid \mathsf{snd}\,\Lambda \mid$$
$$\mathsf{inl}\,\Lambda \mid \mathsf{inr}\,\Lambda \mid \mathsf{case}\,\Lambda\,\mathsf{of}\,\Lambda\,\mathsf{or}\,\Lambda \mid \mu\mathbb{V}.\Lambda.$$

– *The set of free variables of* M *is denoted by* $\mathrm{FV}(M)$. *Let* $\vec{x} = (x_1, \ldots, x_n)$. *We write* $\Lambda^{\vec{x}}$ *for the set of* λ*-terms closed by* $\vec{x}$, *i.e.,* $\{M \in \Lambda | \mathrm{FV}(M) \subseteq \vec{x}\}$. *We write* Λ° *instead of* $\Lambda^{()}$ *(expressions with no free variables, so called* closed expressions*).*

The constructor $\Box$ represents all basic values (integers, booleans, etc.). Pairs are constructed using the expression $\langle e_x, e_y\rangle$, and destructed using projections $\mathsf{fst}\, e$ and $\mathsf{snd}\, e$. The constructors **inl** and **inr** are sum left and right injections of the disjoint unions, whereas **case** is the destructor for these expressions.

Semantics The *value* V of a closed expression M is defined via a standard natural 'big step' operational semantics expressed as judgements of the form $M \Downarrow V$. This evaluation will yield an (also closed) expression in head-normal form.

Definition 2. *Let* $M \in \Lambda^{\circ}$.

- *We write $M \Downarrow V$, and say that M* evaluates to V *if this statement is derivable using the rules in Fig. 1.*
- *M is* defined *or* convergent *(notation $M \Downarrow$) if $M \Downarrow V$ for some value V. Otherwise M is* undefined *or* divergent *(notation $M \Uparrow$).*
- *The set of undefined (closed) expressions (i.e., $\{M \in \Lambda^{\circ} | M \Uparrow\}$) is denoted by $\mathbb{O}$.*

Lemma 1. *$M \Downarrow V$ and $M \Downarrow W \quad \Rightarrow \quad V = W$*

$\mathbb{O}$ contains a canonical inhabitant $\mu x.x$, commonly denoted as $\perp$, that will be used to introduce *finite unfoldings.*

Definition 3. *Let $F_x \in \Lambda^x$. The n^{th} (finite) unfolding (notation F_x^n) is defined inductively by:*

$$F_x^0 = \perp \qquad F_x^{n+1} = F_x[x \leftarrow F_x^n]$$

The following property (the so-called *syntactical continuity property*, formally proved in [8]) relates the evaluation of closed fix-expressions to the evaluation of finite unfoldings, and vise versa.

Proposition 1. *Let $x, y \in \mathbb{V}$, and $C_y \in \Lambda^y$ and $F_x \in \Lambda^x$.*

$$C_y[y \leftarrow \mu x.F_x] \Downarrow \quad \Leftrightarrow \quad \exists m \geq 0 : C_y[y \leftarrow F_x^m] \Downarrow$$

A disadvantage of a 'big step approach' is that reasoning about individual evaluation steps can be awkward. To circumvent this problem the following equivalence relation appears to be useful.

Definition 4. *Two expressions $M, N \in \Lambda^{\circ}$ are* reduction equivalent *(notation $M =_{\beta} N$) if for all $H \in \Lambda^{\circ}$*

$$M \Downarrow H \quad \Leftrightarrow \quad N \Downarrow H$$

3 Strictness

Plain strictness is usually defined as follows.

Definition 5. *Let $\vec{x} = (x_1, \ldots, x_n)$. An expression $E \in \Lambda^{\vec{x}}$ is* strict in x_i *($1 \leq i \leq n$) if for all $\vec{A} \in (\Lambda^{\circ})^n$*

$$A_i \Uparrow \quad \Rightarrow \quad E[\vec{x} \leftarrow \vec{A}] \Uparrow$$

A drawback of this notion of strictness is the lack of compositionality: strictness of a compound expression cannot always be determined by combining strictness of its constituents. For example the expression $\textsf{fst}\, x$ is strict in x and the expression $\langle x, y \rangle$ not (and, of course, also not strict in y). However the compound expression $\textsf{fst}\, \langle x, y \rangle$ *is* strict in x. Our aim is to refine this notion of strictness in such a way that the evaluation properties of expressions are captured more accurately. For instance, the function $\textsf{fst}$ not only evaluates its argument to head-normal form, but successively also evaluates the first component of the resulting pair. Moreover, the expression $\langle x, y\rangle$ is strict in x if it appears in a context that not only needs a pair but also the value of the first component of that pair (or of the second component to become strict in y instead of x), as is the case in our example. These *evaluation contexts* will be expressed as *strictness types.*

Strictness Types

A strictness type is a standard type annotated with so-called *strictness attributes.* The idea is to formulate strictness of E in x by a typing statement

$$x{:}\sigma^! \vdash E : \tau^!$$

The refinement mentioned is accomplished by admitting attributes to more than only the outermost level of a type. For example,

1. $x{:}(\sigma^! \times \tau)^! \vdash \textsf{fst}\, x : \sigma^!$
2. $x{:}\sigma^! \vdash \langle x, x\rangle : (\sigma^! \times \sigma)^!$

Typing (1) expresses that **fst** will evaluate its first argument as indicated above. Typing (2) expresses that in a context in which the first component of a pair is needed (which is indicated by the result type $(\sigma^! \times \sigma)^!$) the expression itself becomes strict in x. Observe that if the second component of the pair was needed, a typing for (2) would be $x{:}\sigma^! \vdash \langle x, x\rangle : (\sigma \times \sigma^!)^!$. To avoid confusion, we now introduce an explicit notation for the absence of strictness information, namely ? (pronounced as *lazy*).

Definition 6. – *Let* $\Phi = \{\alpha, \beta, \alpha_0, \alpha_1, \ldots\}$ *be an infinite set of* type variables, *and* $A = \{!, ?\}$ *the set of* strictness attributes *(ranged over by meta-variables* u, v*),* $\mathbb{T} = \Sigma \cup \Pi$ *denotes the set* strictness types*. Here* Σ *and* Π *are defined by the following abstract syntax.*

$$\begin{aligned} \Sigma &::= \Pi^A \\ \Pi &::= \Phi \mid \mathbb{1} \mid \Sigma \to \Sigma \mid \Sigma \times \Sigma \mid \Sigma + \Sigma \end{aligned}$$

The outermost attribute of $S \in \Sigma$ *is denoted by* $[S]$*. To avoid brackets, we will write* $(\ldots \to \ldots)^u$ *as* $\ldots \overset{u}{\to} \ldots$*.*

– *Let* $|T|$ *denote the 'stripped' version of* T*, i.e.,* T *without any strictness attributes. We consider two types* T_1, T_2 *as* equivalent *(notation* $T_1 \sim T_2$*) if* $|T_1| \equiv |T_2|$*. So, types are equivalent if their underlying standard types are identical.*

Definition 7. – *Strictness attributes are ordered as follows:* $! \leq ?$

– *This ordering on attributes induces the following* coercion relation *on* $\mathbb{T}$*.*

$$\begin{array}{rcl} u \leq v \text{ and } \sigma \leq \tau & \Rightarrow & \sigma^u \leq \tau^v \\ & \mathbb{1} \leq \mathbb{1} & \\ & \alpha \leq \alpha & \\ S_1 \leq S_2 \text{ and } T_1 \leq T_2 & \Rightarrow & S_2 \to T_1 \leq S_1 \to T_2 \text{ and} \\ & & S_1 \times T_1 \leq S_2 \times T_2 \text{ and} \\ & & S_1 + T_1 \leq S_2 + T_2 \end{array}$$

Note the contravariance in the first argument of $\rightarrow$.

- *The* infimum *of two attributes* u, v *(notation* $u \sqcap v$*) if the miminum of* u, v *w.r.t.* $\leq$
- *The predicate* inf *on* $(\mathbb{T}, \mathbb{T}, \mathbb{T})$ *is defined by induction:*

$$\begin{aligned}
\mathit{inf}(\sigma^u, \tau^v, \rho^w) &= u = v \sqcap w \ \textit{and}\ \mathit{inf}(\sigma, \tau, \rho)\\
\mathit{inf}(\alpha, \alpha, \alpha) &= \mathit{true}\\
\mathit{inf}(\mathbb{1}, \mathbb{1}, \mathbb{1}) &= \mathit{true}\\
\mathit{inf}(S \rightarrow T, S_1 \rightarrow T_1, S_2 \rightarrow T_2) &= S = S_1 = S_2 \ \textit{and}\ T = T_1 = T_2\\
\mathit{inf}(S \times T, S_1 \times T_1, S_2 \times T_2) &= \mathit{inf}(S, S_1, S_2) \ \textit{and}\ \mathit{inf}(T, T_1, T_2)\\
\mathit{inf}(S + T, S_1 + T_1, S_2 + T_2) &= \mathit{inf}(S, S_1, S_2) \ \textit{and}\ \mathit{inf}(T, T_1, T_2)\\
\mathit{inf}(\cdot, \cdot, \cdot) &= \mathit{false}
\end{aligned}$$

The last rule should only be used if none of the othes rules applies, i.e. if t_1, t_2 *and* t_3 *are not equivalent.*

The inf predicate is used in our typing system to combine typing assumptions of different occurrences of the same expression variable, commonly called *contraction*. Consider, for example the following function:

$$\lambda x. + (\mathbf{fst}\, x)\, (\mathbf{snd}\, x)$$

If we assume that + is strict in both arguments (say of type N), then the first occurrence of x will get type $(N^! \times N^?)^!$, and the second $(N^? \times N^!)^!$. The occurrences are combined by taking the infimum of their types, being $(N^! \times N^!)^!$.

The meaning of strictness types is formalized by interpreting each strictness type S as a subset of Λ°. We will need two interpretations: $[\![\cdot]\!]_?$ and $[\![\cdot]\!]_!$. For a strictness type S, $[\![S]\!]_?$ denotes *all* expressions that inhabit type S, including $\mathbb{O}$. $[\![S]\!]_!$ denotes the set of expressions that diverge when used in a context with type S. For instance, $[\![\mathbb{1}^!]\!]_?$ contains all expressions that either evaluate to $\Box$ or diverge. $[\![\mathbb{1}^?]\!]_?$ is the same set. $[\![\mathbb{1}^!]\!]_!$ is equal to $\mathbb{O}$, whereas $[\![\mathbb{1}^?]\!]_!$ is empty, since type $\mathbb{1}^?$ used in the latter corresponds to a lazy context. Slightly more complicated is the following example in which we take $(\mathbb{1}^! + \mathbb{1}^?)^!$ as S. Now $[\![S]\!]_?$ contains all expressions that either diverge or evaluate to $\mathbf{inl}\, I$ or to $\mathbf{inl}\, R$, with $I \in [\![\mathbb{1}^!]\!]_?, R \in [\![\mathbb{1}^?]\!]_?$. $[\![S]\!]_!$, however, contains besides all divergent expressions only expressions evaluating to $\mathbf{inl}\, I$, with $I \in [\![\mathbb{1}^!]\!]_! = \mathbb{O}$. The case $\mathbf{inl}\, R$ is impossible since this would require that $R \in [\![\mathbb{1}^?]\!]_! = \emptyset$.

Definition 8. – *Let* $A, B \subseteq \Lambda^{\circ}$.

$$\begin{aligned}
\underline{\mathbb{1}} &= \{M \in \Lambda^{\circ} \mid M \Downarrow \Box\}\\
A \mathbin{\underline{\times}} B &= \{M \in \Lambda^{\circ} \mid \exists a \in A, b \in B : M \Downarrow \langle a, b\rangle\}\\
A \mathbin{\underline{+}} B &= \{M \in \Lambda^{\circ} \mid \exists a \in A : M \Downarrow \mathbf{inl}\, a \ \textit{or}\ \exists b \in B : M \Downarrow \mathbf{inr}\, b\}\\
A \mathbin{\underline{\rightarrow}} B &= \{M \in \Lambda^{\circ} \mid \forall a \in A : (M\, a) \in B\}
\end{aligned}$$

- *The interpretations* $[\![S]\!]_?$ *and* $[\![S]\!]_!$ *are defined by mutual induction in Fig. 2.*

The following property (based on finite unfoldings, see Definition 3) provides an induction scheme for proofs in which fixed point expressions are involved; e.g., see Theorem 1.

$$\begin{aligned}
\llbracket \sigma^u \rrbracket_? &= \mathbb{O} \cup \llbracket \sigma \rrbracket_? \\
\llbracket \alpha \rrbracket_? &= \emptyset \\
\llbracket \mathbb{1} \rrbracket_? &= \underline{\mathbb{1}} \\
\llbracket S \rightarrow T \rrbracket_? &= \llbracket S \rrbracket_? \,\underline{\rightarrow}\, \llbracket T \rrbracket_? \cap \llbracket S \rrbracket_! \,\underline{\rightarrow}\, \llbracket T \rrbracket_! \\
\llbracket S \times T \rrbracket_? &= \llbracket S \rrbracket_? \,\underline{\times}\, \llbracket T \rrbracket_? \\
\llbracket S + T \rrbracket_? &= \llbracket S \rrbracket_? \,\underline{+}\, \llbracket T \rrbracket_?
\end{aligned}$$

$$\begin{aligned}
\llbracket \sigma^u \rrbracket_! &= \emptyset, \text{ if } u = ? \\
&= \mathbb{O} \cup \llbracket \sigma \rrbracket_!, \text{ if } u = ! \\
\llbracket \alpha \rrbracket_! &= \emptyset \\
\llbracket \mathbb{1} \rrbracket_! &= \emptyset \\
\llbracket S \rightarrow T \rrbracket_! &= \llbracket S \rrbracket_? \,\underline{\rightarrow}\, \llbracket T \rrbracket_! \\
\llbracket S \times T \rrbracket_! &= \llbracket S \rrbracket_! \,\underline{\times}\, \llbracket T \rrbracket_? \cup \llbracket S \rrbracket_? \,\underline{\times}\, \llbracket T \rrbracket_! \\
\llbracket S + T \rrbracket_! &= \llbracket S \rrbracket_! \,\underline{+}\, \llbracket T \rrbracket_!
\end{aligned}$$

Fig. 2. Semantics of strictness types

Proposition 2. *Let $T \in \mathbb{T}$, and $x \in \mathbb{V}, F_x \in \Lambda^x$. Then, for all $s \in \{!, ?\}$*

$$(\forall n \in \mathbb{N} : F_x^n \in \llbracket T \rrbracket_s) \quad \Rightarrow \quad \mu x.F_x \in \llbracket T \rrbracket_s$$

The proof of this property in which Proposition 1 plays a crucial role, is quite complex. In PVS it necessitates approximately 1000 proof steps in addition to several non-trivial helper lemmas. The complexity is caused by the fact that our purely syntactical approach requires tedious manipulations of various constructs. In a formalization on paper this would have been barely feasible.

Both interpretations are closed under beta-equivalence.

Proposition 3. *Let $M =_\beta N$. Then, for all strictness types T, and $s \in \{!, ?\}$:*

$$M \in \llbracket T \rrbracket_s \quad \Leftrightarrow \quad N \in \llbracket T \rrbracket_s$$

4 Strictness Typing

In this section we present a type system for deriving strictness information of terms and formally prove that this system is *sound*. Soundness here means that if a term M can be typed with strictness type S, then indeed M is a member of both $\llbracket S \rrbracket_?$ and $\llbracket S \rrbracket_!$. In essence, strictness typing can be characterized as a backwards analysis (E.g., see [9]): strictness properties are determined by relating the effect on demands on the arguments to the effect of demands on the result.

Definition 9. – *A* basis *(or* environment*) is a finite set of declarations of the form $x : S$, where $x \in \mathbb{V}, S \in \mathbb{T}$. For a given basis, all variables are assumed to be distinct. We will sometimes use the 'functional notation' $\Gamma(x)$ to obtain the type assigned to x by Γ.*
- *By $\Gamma^?$ we denote a* lazy basis *containing only declarations of the form $x : \sigma^?$.*
- *Two bases Γ_1, Γ_2 are* equivalent*, denoted as $\Gamma_1 \sim \Gamma_2$, if for each x one has $\Gamma_1(x) \sim \Gamma_2(x)$.*

$$\frac{S \le S'}{\Gamma^?, x{:}S' \vdash x : S}(var) \qquad \frac{}{\Gamma^? \vdash \Box : \mathbb{1}^u}(unit)$$

$$\frac{\Gamma_1 \vdash M : S \overset{[R]}{\to} R \quad \Gamma_2 \vdash N : S \quad [R] \le [S] \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash M\,N : R}(app)$$

$$\frac{\Gamma, x{:}S \vdash B : R \quad u \le [R]}{\Gamma \vdash \lambda x.B : S \overset{u}{\to} R}(abs)$$

$$\frac{\Gamma_1 \vdash X : S \quad \Gamma_2 \vdash Y : T \quad u \le [S] \sqcap [T] \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash \langle X, Y \rangle : (S \times T)^u}(pair)$$

$$\frac{\Gamma \vdash P : (S \times T)^{[S]} \quad [T] = ?}{\Gamma \vdash \mathsf{fst}\, P : S}(fst) \qquad \frac{\Gamma \vdash P : (S \times T)^{[T]} \quad [S] = ?}{\Gamma \vdash \mathsf{snd}\, P : T}(snd)$$

$$\frac{\Gamma \vdash L : S \quad u \le [S]}{\Gamma \vdash \mathsf{inl}\, L : (S + T)^u}(inl) \qquad \frac{\Gamma \vdash R : T \quad u \le [T]}{\Gamma \vdash \mathsf{inr}\, R : (S + T)^u}(inr)$$

$$\frac{\Gamma_1 \vdash I : (S + T)^{[U]} \quad \begin{array}{c} \Gamma_2 \vdash L : S \overset{[U]}{\to} U \\ \Gamma_2 \vdash R : T \overset{[U]}{\to} U \end{array} \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash \mathsf{case}\, I\, \mathsf{of}\, L\, \mathsf{or}\, R : U}(case)$$

$$\frac{\Gamma, x{:}\uparrow S \vdash B : S}{\Gamma \vdash \mu x.B : S}(fix)$$

Fig. 3. Rules for strictness type assignment

- *The* inf *predicate for types extends to (equivalent) bases in a straightforward manner: Let $\Gamma \sim \Gamma_1 \sim \Gamma_2$. Then* $\mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)$ *if* $\mathsf{inf}(\Gamma(x), \Gamma_1(x), \Gamma_2(x))$ *for all x.*
- *The* lazy variant *of a type (notation $\uparrow T$) is defined by:*

$$\uparrow(\sigma^u) = (\uparrow \sigma)^?$$

$$\begin{aligned} \uparrow \alpha &= \alpha \\ \uparrow \mathbb{1} &= \mathbb{1} \\ \uparrow(S \to T) &= S \to T \\ \uparrow(S \times T) &= (\uparrow S) \times (\uparrow T) \\ \uparrow(S + T) &= (\uparrow S) + (\uparrow T) \end{aligned}$$

Note that the removal of strict attributes stops at arrows.

Definition 10. *A* strictness typing statement *is an expression of the form $\Gamma \vdash M : S$, where Γ is a basis. Such a statement is* valid *if it can be derived by using the rules in Fig. 3.*

We emphasize some significant rules. The rule for variables actually enforces that each strictness assumption $x{:}\sigma^!$ in the environment should be 'consumed' by a strict occurrence of x (otherwise, the premise of this rule cannot be valid). The case-rule is subtle. Remember the evaluation of an expression **case** I **of** L **or** R causes the evaluation of I first (explaining the attribute $[U]$ on the corresponding

$$\dfrac{\dfrac{\dfrac{\alpha^! \le \alpha^!}{x{:}\alpha^!, y{:}\beta^? \vdash x : \alpha^!}(var) \qquad ! \le [\alpha^!]}{x{:}\alpha^! \vdash \lambda y.x : \beta^? \xrightarrow{!} \alpha^!}(abs) \qquad ! \le [\beta^? \xrightarrow{!} \alpha^!]}{\vdash \lambda x.\lambda y.x : \alpha^! \xrightarrow{!} \beta^? \xrightarrow{!} \alpha^!}(abs)$$

Fig. 4. A derivation for $\vdash \lambda x.\lambda y.x : \alpha^! \xrightarrow{!} \beta^? \xrightarrow{!} \alpha^!$

sum type), followed by the evaluation of either L (applied to the left component of the result of I) or R (applied to the right component), but not both. So **case** I **of** L **or** R can only be strict in a variable x if either I is strict in x or *both* L and R are strict in x. The latter is accomplished by supplying the type derivations L and R with the same basis Γ_2. To prepare for type inference, the type assignment system is formulated in a fully syntax directed fashion. I.e., non-structural rules, such as contraction, subsumption and weakening (that are usually defined separately), have been incorporated in the structural rules of the system.

As an example, a strictness type derivation for the expression $\lambda x.\lambda y.x$ is given in Fig. 4. More examples can be found in Section 6.

5 Soundness

In this section we demonstrate that our typing system is sound. As previously stated, plain strictness is formulated solely in terms of undefinedness of a function argument as a consequence of undefinedness of the function result. However in our system strictness properties of function arguments can influence each other. Consider, for example, the function $\mathsf{AP} = \lambda f.\lambda x.f\,x$. Then strictness of AP in x depends on the strictness properties of the other argument f. If AP is applied to a strict function then the result will be strict in x. This is expressed by the following valid strictness typing for AP[1]

$$\mathsf{AP} :: (\alpha^! \to \beta^!) \to \alpha^! \to \beta^!$$

At this point it is important to see that Definition 5 is no longer adequate: whether $\mathsf{AP}\,F \perp \Uparrow$ also depends on F. More specifically, F should be a strict function, and not just any arbitrary expression as in Definition 5. E.g., if we take $\lambda x.\Box$ for F then $\mathsf{AP}\,F \perp \Downarrow \Box$.

Definition 11. – *An* expression environment *is a function ρ from $\mathbb{V}$ to Λ^{o}. Such an environment can be lifted to Λ in the obvious way. The result of applying ρ to an expression E is denoted by E^ρ.*
– *Let Γ be a basis. An environment ρ* is valid for *Γ (notation $\rho \Vdash \Gamma$), if $\forall(x{:}S) \in \Gamma : \rho(x) \in [\![S]\!]_?$.*

[1] For clarity, we have omitted the strictness attributes on the arrows. A full typing can be found in Lemma 2.

– *Similarly,* ρ satisfies Γ *(notation $\rho \Vdash \Gamma$) if* $\exists (x{:}S) \in \Gamma : \rho(x) \in [\![S]\!]_!$.

Now the soundness of the type system (with respect to the semantics given in Definition 8) can be formulated as follows:

Theorem 1.

$$\Gamma \vdash E : S \quad \Rightarrow \quad \forall \rho : \rho \Vdash \Gamma \quad \Rightarrow \quad \begin{cases} & \qquad E^\rho \in [\![S]\!]_? \ (1) \\ \rho \Vdash \Gamma & \Rightarrow \quad E^\rho \in [\![S]\!]_! \ (2) \end{cases}$$

Observe that conclusion (2) is essentially a reformulation of Definition 5. The problem about the strictness dependencies between arguments is solved by allowing only *valid* environments. Take, for instance, the expression $f x$ being the body of the AP function in the above example. A possible strictness typing (in which we have substituted $\mathbb{1}$ for both α and β) for this expression is:

$$f{:}\mathbb{1}^! \overset{!}{\rightarrow} \mathbb{1}^!, x{:}\mathbb{1}^! \vdash f\, x : \mathbb{1}^!$$

Soundness of this typing requires that any environment ρ for $f x$ should be both valid for and satisfying $\Gamma = f{:}\mathbb{1}^! \overset{!}{\rightarrow} \mathbb{1}^!, x{:}\mathbb{1}^!$. If such a ρ substitutes $\bot$ for x it indeed satisfies Γ, since $\bot \in [\![\mathbb{1}^!]\!]_!$. However, the validity of ρ prohibits $\lambda x.\Box$ to be substituted for f, because $\lambda x.\Box$ is not a member of $[\![\mathbb{1}^! \overset{!}{\rightarrow} \mathbb{1}^!]\!]_?$, since $\lambda x.\Box \notin [\![\mathbb{1}^!]\!]_! \underline{\rightarrow} [\![\mathbb{1}^!]\!]_!$; see Definition 8.

6 The PVS Formalization

In this section we discuss the formalization of the strictness typing system and its soundness proof in PVS.

We will not assume any preliminary knowledge about PVS and, as in the previous sections, continue to use tool independent notations. The actual formalization is straightforward and barely uses PVS specific constructs. Therefore, it should be relatively easy to convert our PVS specification to other proof assistants like Coq or Isabelle.

Firstly, it must be determined how to represent the variable bindings occurring in abstraction and fixed point expressions. We have chosen the De Bruijn notation mainly because our previous work uses the same representation and the system in this paper does not require significant fine-tuning. In the De Bruijn notation variables are identified by *indices*: natural numbers indicating the number of abstractions which must be skipped in order to localize the corresponding binder. If the variable number exceeds the number of surrounding abstractions, the variable is considered free.

Definition 12. – *The set* Λ_{B} *of* lambda terms with the De Bruijn indices *is defined by the following abstract syntax.*

$$\begin{aligned}\Lambda_{\mathsf{B}} ::= \mathbb{N} \mid \Box \mid \lambda \Lambda_{\mathsf{B}} \mid \Lambda_{\mathsf{B}}\, \Lambda_{\mathsf{B}} \mid \langle \Lambda_{\mathsf{B}}, \Lambda_{\mathsf{B}} \rangle \mid \textbf{fst}\, \Lambda_{\mathsf{B}} \mid \textbf{snd}\, \Lambda_{\mathsf{B}} \mid \\ \textbf{inl}\, \Lambda_{\mathsf{B}} \mid \textbf{inr}\, \Lambda_{\mathsf{B}} \mid \textbf{case}\, \Lambda_{\mathsf{B}}\, \textbf{of}\, \Lambda_{\mathsf{B}}\, \textbf{or}\, \Lambda_{\mathsf{B}} \mid \mu \Lambda_{\mathsf{B}}\end{aligned}$$

– *The predicate* $\mathit{closed}_n(M)$ *checks whether none of the free variables of* M *exceeds* n. *The definition of this predicate is obvious.*

With the De Bruijn notation one can avoid alpha-conversion during evaluation. However, the substitution itself is more complicated because one has to prevent that in $M[x \leftarrow N]$ the free variables of a N get captured by the binders of M. This mandates an adjustment of the free variables of M. Usually the correction of N is performed by an auxiliary function, whereas M is adjusted on-the-fly (e.g., see [10] for a formal definition of these operations).

The soundness property is formulated almost in exactly the same way as Theorem 1. We have proved (1) and (2) in this theorem simultaneously by induction on the derivation of $\Gamma \vdash E : S$. The different cases require on average 200 proof steps each, which sums up to approximately 2200 steps all together. The complete proof files can be downloaded from `www.cs.ru.nl/~sjakie/papers/strictnesstyping/`.

One of the advantages of having a full formalization is that one can actually prove that examples are indeed correct explaining why we have formulated them as a lemma.

Lemma 2. *1.* $\lambda x.\lambda y.x : \alpha^! \xrightarrow{!} \beta^? \xrightarrow{!} \alpha^!$

2. $\lambda f.\lambda x.f\,x : (\alpha^! \xrightarrow{!} \beta^!) \xrightarrow{!} \alpha^! \xrightarrow{!} \beta^!$

3. $\mu f.\lambda n.\textsf{case}\,n\,\textsf{of}\,\lambda x.\textsf{inl}\,\Box\,\textsf{or}\,\lambda x.f\,(\textsf{inr}\,x) : N^! \xrightarrow{!} N^!$

4. $\mu f.\lambda x.\lambda y.\textsf{case}\,x\,\textsf{of}\,\lambda z.f\,(\textsf{inr}\,\Box)\,y\,\textsf{or}\,\lambda z.f\,y\,(\textsf{inr}\,\Box) : N^! \xrightarrow{!} N^! \xrightarrow{!} N^!$

The proof of these typing statements can also be found in the PVS files. In Example 3 and 4, N stands for the type $\mathbb{1} + \mathbb{1}$ which we use to represent natural numbers. The actual values of these numbers are not relevant: It suffices if we can distinguish 0 (represented as $\textsf{inl}\,\Box$) from all other numbers (represented as $\textsf{inr}\,\Box$). Example 3 resembles the factorial function (without the usual multiplication and subtraction), and Example 4 (showing a recursive function that is strict in both arguments) is taken from [11].

Conducting the Proof

Automated theorem proving is very time consuming. We estimate that the construction of the entire proof (including the development of the necessary theoretic concepts) comprises about six man-months work, maybe even more. On the other hand, it revealed mistakes which had been made in the previous (first-order) version of the type system. Moreover, extending the language with higher-order constructs made the system significantly more complex. For example, the treatment of function types in Def. 7 showed to be non-intuitive. Various attempts were made preceding the final version. The PVS formalization helped us by enabling quick verification of modifications in definitions.

Our work can be considered as a contribution in the spirit of [6]. One of the PoplMark challenges is the treatment of variable binding. Most of the solutions that have been reported to this challenge are based on de Bruijn indices. Though [6] argues that this representation introduces too much overhead in formal proofs, and should therefore be avoided, this is not confirmed by our approach or by any of these solutions. The low-level variable representation does not lead to any significant increase in the complexity of the proofs. With a few simple, and easy–to–prove auxiliary lemmas, one can effectively hide all implementation details. The main proof itself is not affected. In fact, the implicit bindings of De Bruijn's representation allow environments to be represented as simple lists rather than lists of pairs.

We believe that, besides obtaining full confidence, one of the main advantages of possessing a formal proof is the possibility to replay the proof in a tool. For instance, if one cannot immediately follow the given explanations, in principle one can fall back on the fully elaborated formal version.

7 Recursive Data Structures

Thus far we have only considered non-recursive data structures. In this section we will describe an extension of the theory to lists. This extension can serve as a bases for the treatment of other recursive data types.

Lists are built up from constructors **nil** (the empty list) and **cons**. We incorporate these constructs in our syntax together with a destructor called **list** leading to the following extension of Definition 1

$$\Lambda ::= \cdots \mid \mathsf{nil} \mid \mathsf{cons}\,\Lambda\,\Lambda \mid \mathsf{list}\,\Lambda\,\Lambda\,\Lambda$$

By $\mathsf{L}(\sigma)$ we denote the (standard) type of lists of σ objects.

The evaluation rules for these construct are straightforward: hnf-evaluation of list objects stops at the outermost constructor.

$$\frac{}{\mathsf{nil} \Downarrow \mathsf{nil}}(\mathit{nil}) \qquad \frac{}{\mathsf{cons}\,H\,T \Downarrow \mathsf{cons}\,H\,T}(\mathit{cons})$$

$$\frac{L \Downarrow \mathsf{nil} \qquad N \Downarrow V}{\mathsf{list}\,L\,N\,C \Downarrow V}(\mathit{list}\text{-}N) \qquad \frac{L \Downarrow \mathsf{cons}\,H\,T \qquad C\,H\,T \Downarrow V}{\mathsf{list}\,L\,N\,C \Downarrow V}(\mathit{list}\text{-}C)$$

Besides plain hnf-evaluation, it is useful to distinguish other evaluation forms. The most common ones are *spine evaluation* and *full evaluation.* For instance, the function *length* computing the size of the list will enforce the complete evaluation of the list structure, but leave the elements unaffected. A function *sum* that sums all the elements of a list will not only evaluate the spine completely, but also all of its elements.

The evaluation contexts induced by these functions are again encoded in an appropriate strictness type. This leads to the following extension of Definition 6:

$$\Pi ::= \cdots \mid \mathsf{L}^{A}(\Sigma)$$

Only part of the types that can be constructed according to this syntax is well-formed. This is due to the fact that for data structures, strictness 'propagates outwards': in order to evaluate inner components of a data structure, the structure itself has to be evaluated before these components can be accessed.

For example, for a list of $\mathbb{1}$ elements we have 4 different valid strictness variants: $(\mathsf{L}^?(\mathbb{1}^?))^?$, $(\mathsf{L}^?(\mathbb{1}^?))^!$, $(\mathsf{L}^!(\mathbb{1}^?))^!$, and $(\mathsf{L}^!(\mathbb{1}^!))^!$, corresponding to no, hnf, spine and full evaluation, respectively.

The typing rules are extended according to our intended semantics.

$$\frac{}{\Gamma^? \vdash \textsf{nil} : \mathbb{1}^u}(nil)$$

$$\frac{\Gamma_1 \vdash H : S \quad \Gamma_2 \vdash L : (\mathsf{L}^v(S))^v \quad u \le v \quad v \le [S] \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash \textsf{cons}\, H\, L : (\mathsf{L}^v(S))^u}(cons)$$

$$\frac{\Gamma_1 \vdash L : (\mathsf{L}^v(S))^{[T]} \quad \begin{array}{c}\Gamma_2 \vdash N : T \\ \Gamma_2 \vdash C : S \overset{[T]}{\to} (\mathsf{L}^v(S))^v \overset{[T]}{\to} T\end{array} \quad [T] \le v \quad \mathsf{inf}(\Gamma, \Gamma_1, \Gamma_2)}{\Gamma \vdash \textsf{list}\, L\, N\, C : T}(list)$$

For a soundness proof, all definitions based on either expressions or on types have to be adjusted in order to deal with list constructs. All of these adjustments are reasonably straightforward, and can be found in the PVS formalization. However, the formalization does not yet contain a completely formalized proof; i.e., the list cases are still missing, mainly due to the increased complexity of the proof.

Again, we formulate some examples as lemmas which enables us to formally prove that the typing statements are indeed correct.

Lemma 3. *1.* $\mu l.\lambda x.\textsf{list}\, x\, (\textsf{inl}\,\square)\, \lambda h.\lambda t.l\, t : (\mathsf{L}^!(\alpha^?))^! \overset{!}{\to} N^!$
2. $\mu s.\lambda x.\textsf{list}\, x\, (\textsf{inl}\,\square)\, \lambda h.\lambda t.{+}\, h\, (s\, t) : (\mathsf{L}^!(N^!))^! \overset{!}{\to} N^!$
3. $\forall u, v, w \in A, v \le w, w \le u : \mu r.\lambda x.\lambda y.\textsf{list}\, x\, y\, \lambda h.\lambda t.r\, t\, (\textsf{cons}\, h\, y) :$
$(\mathsf{L}^v(\alpha^u))^v \overset{v}{\to} (\mathsf{L}^w(\alpha^u))^w \overset{v}{\to} (\mathsf{L}^w(\alpha^u))^v$

We use the same representation for natural numbers as in Example 2. The operation $+$ is simply represented by $\perp$, because the only relevant aspect of $+$ is this example us that it is strict in both arguments, and $\perp$ can be typed as a strict binary operation. Example 1 is the *length* function (without addition), and Example 2 the *sum* function. Example 3 is the well-known *reverse* function, that transfers each element of the first list argument to the second argument. One will usually call this function with an empty list as second argument. In that case, all elements of the first list will appear in reverse order in the final result. By using quantified attributes, we obtain a polymorphic strictness typing for reverse. In this typing the difference between the first and the second argument becomes apparent: Even simple hnf-evaluation of an application of reverse will result in the complete evaluation of the spine of the first argument. For the second argument this is not the case. This argument will only be evaluated when the whole spine of the reverse's result is needed.

8 Discussion

We compared several existing techniques for strictness analysis by giving a brief outline of their main ideas.

In [12,13] a non-standard type inference is introduced using conjunction types. The main properties of the system are formulated and proved with respect to a denotational semantics of their language. The difference with our approach is that the strictness information is restricted to traditional head-normal form evaluation only, which, as we argued, hampers modularity. In [14] a typing framework is presented focusing on algorithmic aspects, by providing a checking algorithm for a variation of Jensen's system.

The system described by [11] is most similar to ours. For a language resembling the core functional language introduced in Section 2, the authors describe both a strictness and a totality analysis using a non-standard type inference system. The main difference with our approach is that conjunction types are used. In our system the strictness properties of all function arguments are captured by a single strictness type, whereas the system of [11] requires a conjunction of these properties. The advantage of our approach is that it can be incorporated directly in a standard Hindley-Milner type inference algorithm.

The system introduced by [15] is based on *relevance typing*. Similar to our system, and the backwards strictness analysis used in the Glasgow Haskell Compiler [16], the (evaluation) context in which variables are used determines whether these variables are *relevant* if such a context is evaluated. In [15] the emphasis is on exploiting strictness information by defining a transformation replacing ordinary function applications by a more efficient *eager* applications.

Strictness analysis by *abstract interpretation* introduces a non-standard semantics by translating functions into abstract versions over finite domains, notably over finite lattices. The bottom elements of these domains play the role of generic 'undefined' values. Recursive abstract functions are defined by a fixed-point construction. The main property of this alternative interpretation is to yield a decidable approximation of the (in general undecidable) strictness property, even in the higher-order case. This abstraction inevitably leads to information loss. The standard form of abstract interpretation uses the two-point lattice as ground domain. See [17], [18] and [19] for more information. Due to the complexity of finding fixed points in abstract domains, abstract interpretation is not very useful for implementing strictness analysis in compilers for functional languages.

Abstract reduction analyzes evaluation of expressions by mimicking reduction on sets of concrete values extended with special elements for undefinedness. This technique approximates ordinary computations closer than for instance abstract interpretation or strictness typing. Rewriting semantics is adjusted by specifying the behaviour of functions on non-standard elements. Abstract reduction sequences may not terminate. A special technique called *reduction path analysis* is used to cut off these sequences in a way that keeps most of the strictness information intact; see [20], [21]. The main disadvantage of this approach is the lack of modularity; it requires the implementations of the involved functions to perform the analyses effectively.

Strictness typing is a purely syntactic ('intentional') way of deriving strictness information. The resulting strictness information merely depends on the structure of the expressions, particularly on the occurrences of case clauses, and (as in the case of abstract interpretation) not on the computational behaviour on concrete values. The advantage of strictness typing over abstract interpretation is that the first method can be combined with standard typing. For more information, the reader is referred to [22].

9 Conclusions and Future Work

In this paper we presented a strictness typing system which is fully higher-order. Moreover, it enables to specify arbitrary evaluation contexts which is essential for supporting modularity. Like many other meta-theoretical expositions we used De Bruijn indices to represent term variables. Despite the objections that have been raised against this low-level representation (e.g., See [6]), we encountered no real issues that significantly hampered our proofs.

We have demonstrated that proof assistants are not only useful in formalizing existing proofs but also to develop new language theoretic concepts. One major concern, however, remains the fact that the construction of a formalized proof remains very time consuming. Compared to a pen and paper version the difference in development time is probably (much) more than a factor of three. It is difficult to determine whether this is worth the investment.

Furthermore, we have developed a prototype implementation of higher-order strictness typing. This algorithm can be used to show that type assignment has the principal typing property: if an expression E is typable, there exists a principal typing for E, i.e., a 'schematic' type of which all other typings can be obtained via instantiation.

Future work. The formal proof does not yet cover soundness of the complete system: the proof work on recursive data types is not fully completed yet. The proof formalization of the property that principal types can be computed effectively, is also part of future work.

References

1. Walker, D.: Substructural type systems. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, pp. 3–44. MIT Press (2004)
2. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Mathematical Structures in Computer Science 6, 579–612 (1996)
3. Turner, D.N., Wadler, P., Mossin, C.: Once upon a type. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA 1995, pp. 1–11. ACM, New York (1995)
4. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. 4, 167–187 (1996)
5. Shkaravska, O., van Eekelen, M.C.J.D., van Kesteren, R.: Polynomial size analysis of first-order shapely functions. Logical Methods in Computer Science 5(2) (2009)
6. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Babu, C. S., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLMARK challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)

7. Barendsen, E., Smetsers, S.: Strictness analysis via resource typing. In: Reflections on Type Theory, Lambda Calculus, and the Mind, Nijmegen, Netherlands, pp. 29–40 (December 2007)
8. Smetsers, S.: The syntactic continuity property: A computer verified proof. In: Majkic, Z., Hsieh, S.-Y., Ma, J., El Emary, I.M.M., Husain, K.S. (eds.) International Conference on Theoretical and Mathematical Foundations of Computer Science, TMFCS 2010, pp. 135–142. ISRST (2010)
9. Davis, K., Wadler, P.: Backwards strictness analysis: Proved and improved. In: Proceedings of the 1989 Glasgow Workshop on Functional Programming, London, UK, pp. 12–30. Springer (1990)
10. Kamareddine, F.: Reviewing the classical and the de Bruijn notation for λ-calculus and pure type systems. Logic and Computation 11, 11–13 (2001)
11. Coppo, M., Damiani, F., Giannini, P.: Strictness, totality, and non-standard-type inference. Theor. Comput. Sci. 272(1-2), 69–112 (2002)
12. Jensen, T.: Abstract interpretation in logical form. PhD thesis, Datalogisk Institut, Københavns Universitet (1992)
13. Benton, P.: Strictness analysis of lazy functional programs. PhD thesis, Computer Laboratory, University of Cambridge (1993)
14. Hankin, C., Métayer, D.L.: Deriving algorithms from type inference systems: Application to strictness analysis. In: POPL, pp. 202–212 (1994)
15. Holdermans, S., Hage, J.: Making "stricterness" more relevant. In: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, pp. 121–130. ACM, New York (2010)
16. Jones, S.L.P., Hall, C., Hammond, K., Cordy, J., Kevin, H., Partain, W., Wadler, P.: The glasgow haskell compiler: a technical overview (1992)
17. Mycroft, A.: Abstract interpretation and optimising transformations for applicative programs. PhD thesis, University of Edinburgh (1981)
18. Burn, G., Hankin, C., Abramsky, S.: The theory of strictness analysis for higher order functions. In: Ganzinger, H., Jones, N.D. (eds.) Programs as Data Objects. LNCS, vol. 217, pp. 42–62. Springer, Heidelberg (1986)
19. Wadler, P.: Strictness analysis over non-flat domains. In: Abstract Interpretation of Declarative Languages. Ellis Horwood (1987)
20. Nöcker, E.: Strictness analysis using abstract reduction. In: Proc. of Conference on Functional Programming Languages and Computer Architecture, FPCA 1993, Kopenhagen, pp. 255–266. ACM Press (1993)
21. Clark, D., Hankin, C., Hunt, S.: Safety of strictness analysis via term graph rewriting. In: SAS 2000. LNCS, vol. 1824, pp. 95–114. Springer, Heidelberg (2000)
22. Leung, A., Mishra, P.: Reasoning about simple and exhaustive demand in higher-order lazy languages. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 328–351. Springer, Heidelberg (1991)

Call-by-Value Semantics for Mutually Recursive First-Class Modules

Judith Rohloff and Florian Lorenzen

Technische Universität Berlin
Compiler Construction and Programming Languages
Ernst-Reuter-Platz 7, D-10587 Berlin
{judith.rohloff,florian.lorenzen}@tu-berlin.de

Abstract. We present a transformation based denotational semantics for a call-by-value language with first-class, hierarchical and recursive modules. We use the notion of *modules* as proposed in [1]. They merge dynamic data structures with aspects of modularisation and name binding in functional programming languages. *Modules* are first-class values which capture recursive definitions, lexical scoping, hierarchical structuring of programs and dynamically typed data structures in a single construction. We define a call-by-value language ModLang and explain what problems occur in combining nested, recursive and first-class modules. We then show how to solve these problems by defining a dependency analysis to determine the evaluation order, enabling a transformation into an intermediate representation. Finally, we present a denotational call-by-value semantics.

1 Introduction

Modularisation in an essential tool for managing complexity in the design of large scale software. To be effective, module systems must organise code into fine grained hierarchies without impeding reusability. The competing nature of these goals has given rise to several approaches for designing module systems. With parametric modules, we obtain adaptable code that is easy to reuse at different parts of the architecture. First-class modules [2,3,4], i.e. modules that can be arguments and results of functions, allow parametric modules to be defined as normal functions, so no special construct is needed. Furthermore, first-class modules enable runtime reconfiguration of the architecture. With nested modules [5] a fine grained module hierarchy is possible. Disallowing mutually recursive modules [6,7,8] often destroys the natural structure of a program.

Recent approaches [9,8,10] have attempted to combine all three features in a call-by-value language, but none of them have achieved full support. In this paper we describe a way to give a call-by-value semantics to a language with higher-order functions and nested, first-class, mutually recursive modules.

We use the grouping concept proposed in [1] as modules and call the proposed construct *module*. Pepper introduces the concept of *modules* as a unified construct for bindings and data structures. *Modules* are sets of definitions. As first-class values, they can also be used as records. They distinguish from records in that all definitions within a module can see each other. In [1] *modules* are treated coalgebraically and regarded as objects implicitly defined by their observers.

In contrast to this semantic approach, we focus here on an efficient implementation by defining formal semantics and analyses for *modules*. For this purpose we define a small functional call-by-value language ModLang, which is similar to an untyped λ-calculus extended by *modules*.

As *modules* are first-class values, have binding definitions and are allowed to be mutually recursive, it is not possible to give a straightforward semantics. In this paper we describe a way to give a transformation-based call-by-value semantics to *modules*.

The paper starts with a short introduction of the *module* concept. In Sec. 3 we describe why it is not possible to give a straightforward semantics and introduce our solution. In a call-by-value semantics, an evaluation order is needed, as every variable must be bound to a value before it is used. There are two possibilities to obtain this order. Either the programmer has to state all definitions in dependency order (e.g. ML), or the order is determined via dependency analysis (e.g. Opal or Modula-3 [11]). As *modules* are sets of definitions, no order is given. The dependency analysis for our language is described by an example in Sec. 4. In Sec. 5 the result of the dependency analysis is used to transform the input program into an intermediate representation with explicit dependency order. In Sec. 6 we present a denotational semantics for our intermediate representation. Sec. 7 discusses related work and Sec. 8 summarises our results and gives an overview of current and future work.

2 The ModLang Language

ModLang is a functional language with *modules*. We describe the language ModLang by some examples. The syntax is given in Sec. 4.

2.1 Introductory Example: Lists

Abstractions, applications and conditionals have their usual meaning. We focus on *modules* and selections.

Following [1] we define a *module* as a set of named definitions. Listing 1.1 shows the well-known example of lists. The *module* `List` contains five definitions `nil`, `cons`, `head`, `tail` and `enum`.[1] In this example the *module* represents a module encapsulating definitions. Since the order of the definitions is irrelevant, it can be seen as a set — hence the surrounding curly braces.

[1] A usable list implementation would require the discriminators `isNil` and `isCons`. They can be implemented using the *module* discriminator `DEFINES`, which checks if a given *module* defines exactly the given variables. As `DEFINES` is a dynamic check, we do not consider it in this paper.

```
List = {
    nil  = {}
    cons = λx.λxs. { hd=x  tl=xs }
    head = λxs. xs.hd
    tail = λxs. xs.tl
    enum = λn. { enum = λi.
                       IF i==n THEN nil ELSE cons i (enum (i+1))
                   }.enum 0
}
```

Listing 1.1. Lists in ModLang

Both constructors, `nil` and `cons`, implement the list elements with *modules* as well — `nil` is the empty *module* and `cons` returns a *module* containing the head `hd` and the rest `tl` of the list. In this case *modules* are used as data structures, which can be built up and changed at runtime.

The function `head` (`tail`) selects the first element (the rest) of the given list via the selection operator "`.`". In this manner these functions abstract the selectors `.hd` and `.tl`.

The function `enum` forms a list of numbers between `0` and `n`. This definition illustrates several aspects of our language design:

– All variables of the outer *module* can be used on the right hand side of a definition. In this case these are `nil` and `cons`.
– Within the expression an anonymous *module* containing the definition `enum` is defined. We call a *module* anonymous if it is not the top level node of the syntax tree of a right-hand side of a definition.
– In the definition of `enum` of the anonymous *module* the variable `enum` is used. Because of the lexical scoping, the innermost definition is addressed. Thus it is `enum` of the anonymous *module* and not `List.enum`.
– On the right hand side of `List.enum` the selector `.enum` is called. The anonymous *module* and its selection are used as a local recursive "let-in" or "where".

2.2 Functional-Object-Oriented Programming

Our second example shows, how we can use *modules* to program in an object oriented manner. Listing 1.2 shows an eval-apply-interpreter with environments for the untyped λ-calculus. A term is either a variable `Var`, an abstraction `Abs` or an application `App`. These term constructors are quite similar to the "objects-as-closures" implementation [12], but here they are evaluated to *modules* instead of closures. Those *modules* contain an evaluation function `eval`, which uses an environment to compute the value of the term. In doing so, the sub-terms' evaluation functions may also be used.

During the evaluation of a variable, its value is determined from the environment Γ (line 2). To evaluate an abstraction, a closure (i.e. *module*) with the abstract variable `var`, the function body `body` and the current environment `env` must be built (line 3).

```
Term = {
  Var    = λx. { eval = λΓ. Γ.lookup x }
  Abs    = λx.λt. { eval = λΓ. { var=x  body=t  env=Γ } }
  App    = λt1.λt2. { eval  = λΓ. apply (t1.eval Γ) (t2.eval Γ)
                      apply = λf.λa.
                        f.body.eval (f.env.add f.var a) }
}
```

Listing 1.2. An eval-apply-interpreter in a functional-objekt-oriented style

The implementation of the environment is shown in Listing 1.3. An environment is a list of records `{var=x val=v}` using the list implementation in Listing 1.1. An environment has the functions `add` to add a new binding and `lookup` to determine the value of a variable. The constructor `Env` creates an environment containing all bindings of `bdgs`.

In contrast to terms new environments must be created during evaluation. In `add` a new environment is created with `Env` which contains all previous bindings as well as the new binding `Bdg x v`

```
Env = λbdgs. {
  Bdg    = λx.λv. { var=x  val=v }
  add    = λx.λv. Env (List.cons (Bdg x v) bdgs)
  lookup = ... // searching in list
}
```

Listing 1.3. Environment for eval-apply-interpreter

As demonstrated by this example:

- Objects can be realised as *modules*. Methods are definitions within a *module*.
- Constructors of object oriented programming languages can be seen as functions returning *modules*.
- Variables, abstracted by λ within a constructor play the role of fields. It is also possible to define them in definitions, making them accessible from outside by selection.
- The binding rules for *modules* enable methods to have access to the methods and fields of its enclosing object. A special reference like `this` in Java or `self` in Smalltalk does not exist.

2.3 Mutually Recursive Modules

The list example has already shown that definitions may be recursive. Mutually recursive definitions are allowed as well, even across *module* borders. An example for mutually recursive *modules* is given in Listing 1.4. This program will be used as a running example throughout the rest of this paper. In this example the two functions `E.even` and `O.odd` are defined. Jointly they decide whether a given number is even or not. `O.odd` uses `E.even` and vice versa. Moreover these two functions are used by `E.is2even` and `O.is2odd`.

```
{ E = { even    = λn. IF n==0 THEN true ELSE O.odd (n-1)
        is2even = even val
        val     = 2 }
  O = { odd    = λn. IF n==0 THEN false ELSE E.even (n-1)
        is2odd = odd 2 }
}
```

Listing 1.4. Mutually recursive *modules*

3 Towards an Evaluation

In a call-by-name semantics no evaluation order is needed, as we can evaluate all definitions on demand. Therefore, a call-by-name semantics to ModLang can be readily given. Although it is a matter of taste which semantics one prefers, call-by-value languages are easier to combine with parallel programming and side effects. For these reasons, it is desirable to give a call-by-value semantics to ModLang.

As previously mentioned, implementing call-by-value semantics for ModLang requires a dependency analysis. As we combine all three features of our modules it is not easy to find the evaluation order. We describe the difficulties involved by starting with only nested modules and then adding mutual recursion. Finally we make our modules first-class values.

3.1 Nested Modules

We start with nested modules, which enable building up a hierarchy of *modules*. *Modules* are only allowed as top-level expression or as the right hand side of a definition. Definitions are not allowed to be mutually recursive. It follows that within every *module*, an order for all definitions can be found.

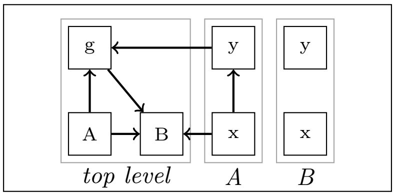
To obtain the evaluation order for the example in Listing 1.5 it is obvious that we first have to evaluate the complete *module* `B` with both definitions. Then the definition `g` can be evaluated, followed by the *module* `A`.

```
{A = {x = B.x + y   y = g}
 g = B.x + B.y
 B = {x = 1   y = 3}
}
```

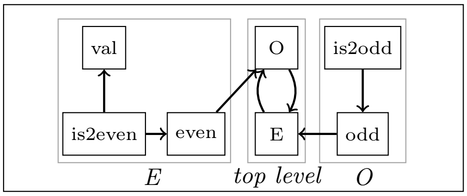
Listing 1.5. Example for evaluation order

For programs without first-class and mutually recursive modules, the order could be determined by regarding only the free variables of all right hand sides and computing a normal dependency graph. The dependency graph for this example is given in Fig. 1(a).

As will be shown in the next section, this approach is insufficient when mutually recursive modules are allowed.



(a) Dependency graph for the example in Listing 1.5.



(b) Dependency graph for the example in Listing 1.4.

Fig. 1. Dependency graphs using free variables

3.2 Mutually Recursive Modules

We now allow nested *modules* to be recursive. As before, they are still not allowed to be defined within an expression. As recursion in a call-by-value semantics only terminates if all definitions are functions we forbid cycles where at least one definition is not a function, as typical in most call-by-value languages. Even with this constraint it is still not easy to find the evaluation order.

To understand the problem with mutually recursive *modules*, we will again consider the even-odd example in Listing 1.4.

In a call-by-value semantics recursion is only possible for λ-abstraction, since those definitions evaluate to closures. Therefore, it is not possible to build up a closure for the entire *module*. This would only be possible if the auxiliary functions `is2even` and `is2odd` were not present. We have to find an order for all definitions within the two *modules* `E` and `O`. For the given example, the call-by-value semantics requires that `even` and `odd` must be evaluated before `is2odd` and `is2even`. It is not possible to find this order by only regarding the free variables. The dependency graph for this example with free variables is shown in Fig. 1(b). There is a cycle between `E` and `O`, which is forbidden as both definitions are *modules* and not functions. This contradicts the goal of allowing mutually recursive *modules*.

To calculate the correct evaluation order we look at dependencies across *module* borders and also consider selection chains (see definition in Sec. 4.2). In this example, a possible evaluation order is: `E.val`, `E.even`, `O.odd`, `E.is2even`, `E`, `O.is2odd`, `O`.

By using the complete selection chain for the dependency, the hierarchy is broken up and the mutually recursive *modules* are handled simultaneously.

In [1] a flattening mechanism is proposed to find an evaluation order. Here the hierarchy is broken up and all definitions occur at the same level. The result of the flattening for example in Listing 1.4 is shown in Listing 1.6. Instead of just one identifier, the name on the left-hand-side of all definitions is the complete path with dot notation. Furthermore, all variables must be replaced by the complete name within the *module*, e.g `even` in the definition `is2even` must be replaced by `E.even`. Renaming is necessary, to ensure unique names after flattening as it could be possible to have the same definition name in two *modules*.

```
E.even    = λn. IF n==0 THEN true ELSE O.odd (n-1)
E.is2even = E.even val
E.val     = 2
O. odd    = λn. IF n==0 THEN false ELSE E.even (n-1)
O.is2odd = O.odd 2
```

Listing 1.6. Example for flattening

After flattening, we can build up a dependency graph. But instead of just using the free variables, we now use the complete selection chain.

Flattening is only possible for mutually recursive nested modules. The technique is not always applicable to first-class values, as shown by the example in the next section.

3.3 An Undecidable Problem

Up to now *modules* were only allowed as top level expressions or as the right-hand-side of a definition. If we allow *modules* to be first-class values, i.e. they are allowed at any position in an expression, then flattening is not possible anymore.

In the previous section we regarded the complete selection chain for the dependencies. This mechanism is quite similar to path resolution, which is undecidable as proven in [9]. We will just give an informal explanation here.

In Listing 1.7 an artificial example is given. For flattening the expression representing the complete selection chain `s.a.b` must be found. In this example this name only exists if the input fulfils some predicate `isValid`. This is generally not decidable.

```
λinput. {
  s      = IF isValid input THEN {a = 1}
                            ELSE {a = {b = 1}}
  output = s.a.b
}
```

Listing 1.7. A problematic example (Using `List` of Listing 1.1)

In this case it is sufficient to recognise that `output` depends on `s`, so the evaluation order is: `s`, `output`. In the case that `s.a` has no selector `b`, a runtime error will occur, which is acceptable and common for failed selection.

Consequentially, there are some cases where we have to regard the selection chain and some cases where parts of the chain should not be regarded. For every selection chain we regard only the "static" part and ignore the "dynamic" part. The static part of a selection chain is the variable and all selectors representing a path within the current *module* hierarchy. In this example it is just the lexically visible variable `s`, `.a.b` is the dynamic part, as these names only occur at runtime. The proper way to calculate the dependencies is described in Sec. 4.

3.4 Solution

As previously mentioned, we start with a special dependency analysis to determine the evaluation order. The result of the dependency analysis is used to

$$\begin{array}{lll} E & ::= & M \mid T \\ T & ::= & [\lambda\ x\ .\ E]^\ell \mid [E\ E]^\ell \mid x^\ell \mid [E\ .\ x]^\ell \\ M & ::= & \{\ D^*\ \}^\ell \\ D & ::= & x = M \mid x = T \end{array}$$

Fig. 2. Syntax of labelled expressions

transform the whole input program into an intermediate representation with explicit ordering. This intermediate representation allows a simpler definition of the semantics.

4 Dependency Analysis

In this section we illustrate our special dependency analysis by means of the running example. We begin with the following definitions.

4.1 Labelled Expressions

The syntax of ModLang is given in Fig. 2. We distinguishes between *modules* and other expressions by splitting E into *modules* M and terms T and we omit conditionals and constants since they are not important for the rest of the paper. Furthermore we annotate all expressions $M \cup T$ with a unique label of the set ℓ and use brackets to clarify which part of an expression is labelled. The inverse of the labelling is the mapping $EXP : \ell \hookrightarrow M \cup T$. We use the labelling to attach additional information to expressions.

4.2 Names

Selector / Name. A *selector* is an identifier with a prefix dot. A *selector chain* is a sequence of selectors.

A *name* is a variable (identifier without a prefix dot) followed by a (possibly empty) selector chain. The set of all names is denoted by N.

Free and Available Names. Every expression E of a program is mapped to two sets using the labels ℓ:

$FN \ : \ell \hookrightarrow \mathbb{P}N$	set of free names
$AVM : \ell \hookrightarrow \mathbb{P}x$	set of *module*-bound available variables

Free names extend the λ-calculus concept of free variables to names. Names are free iff the leading variable is free. The function $\mathcal{F}$ of Fig. 3 maps every expression to its free names.

Calculation of free variables and free names differs only in equation $(*)$ in Fig. 3. Here, the whole name $x_1 . \cdots . x_m$ is inserted instead of the variable x_1. The mapping FN is the composition $FN = \mathcal{F} \circ EXP$.

$$\mathcal{F} : \; M \cup T \to \mathbb{P}N$$

$$\begin{array}{lll}
\mathcal{F}\llbracket \lambda x \,.\, E \rrbracket & = \mathcal{F}\llbracket E \rrbracket \ominus \{x\} & \\
\mathcal{F}\llbracket E_1 \; E_2 \rrbracket & = \mathcal{F}\llbracket E_1 \rrbracket \cup \mathcal{F}\llbracket E_2 \rrbracket & \\
\mathcal{F}\llbracket x_1 . \cdots . x_m \rrbracket & = \{x_1 . \cdots . x_m\} & (*) \\
\mathcal{F}\llbracket E \,.\, x \rrbracket & = \mathcal{F}\llbracket E \rrbracket & \\
\mathcal{F}\llbracket x \rrbracket & = \{x\} & \\
\mathcal{F}\llbracket \{\!\!\{ \, x_i = E_i{}^{i \in 1..m} \, \}\!\!\} \rrbracket & = \Big\{ N \mid N = y_1 . \cdots . y_p \wedge N \in \bigcup_{i \in 1..m} \mathcal{F}\llbracket E_i \rrbracket \wedge y_1 \notin \{x_i{}^{i \in 1..m}\} \Big\} &
\end{array}$$

Fig. 3. Function calculating free names

$$\mathcal{A}^{\alpha} : \; \alpha \to \mathbb{P}x \to (\ell \hookrightarrow \mathbb{P}x) \qquad \text{for } \alpha \in \{E, T, M, D\}$$

$$\begin{array}{lll}
\mathcal{A}^{E}\llbracket T \rrbracket \mathit{\Gamma} & = \mathcal{A}^{T}\llbracket T \rrbracket \mathit{\Gamma} & \\
\mathcal{A}^{E}\llbracket M \rrbracket \mathit{\Gamma} & = \mathcal{A}^{M}\llbracket M \rrbracket \emptyset & (\dagger) \\
\mathcal{A}^{T}\llbracket [\lambda x \,.\, E]^{\ell} \rrbracket \mathit{\Gamma} & = \{\ell \mapsto \mathit{\Gamma}\} \cup (\mathcal{A}^{E}\llbracket E \rrbracket \mathit{\Gamma} \setminus \{x\}) & \\
\mathcal{A}^{T}\llbracket [E_1 \; E_2]^{\ell} \rrbracket \mathit{\Gamma} & = \{\ell \mapsto \mathit{\Gamma}\} \cup \mathcal{A}^{E}\llbracket E_1 \rrbracket \mathit{\Gamma} \cup \mathcal{A}^{E}\llbracket E_2 \rrbracket \mathit{\Gamma} & \\
\mathcal{A}^{T}\llbracket [E \,.\, x]^{\ell} \rrbracket \mathit{\Gamma} & = \{\ell \mapsto \mathit{\Gamma}\} \cup \mathcal{A}^{E}\llbracket E \rrbracket \mathit{\Gamma} & \\
\mathcal{A}^{M}\llbracket \{\!\!\{ \, D_i{}^{i \in 1..m} \, \}\!\!\}^{\ell} \rrbracket \mathit{\Gamma} & = \{\ell \mapsto \mathit{\Gamma}\} \cup \left(\bigcup_{i \in 1..m} \mathcal{A}^{D}\llbracket D_i \rrbracket (\mathit{\Gamma} \cup \{x_i{}^{i \in 1..m}\}) \right) & \\
 & \qquad \text{where} \quad D_i = x_i = T_i \text{ or } D_i = x_i = M_i & \\
\mathcal{A}^{D}\llbracket x = T \rrbracket \mathit{\Gamma} & = \mathcal{A}^{T}\llbracket T \rrbracket \mathit{\Gamma} & \\
\mathcal{A}^{D}\llbracket x = M \rrbracket \mathit{\Gamma} & = \mathcal{A}^{M}\llbracket M \rrbracket \mathit{\Gamma} & (\ddagger)
\end{array}$$

Fig. 4. Calculation of *module*-bound available variables

Available variables of an expression T are variables introduced by the outer context of T. We are especially interested in variables introduced by a so called uninterrupted *module* hierarchy. The syntax tree of an uninterrupted *module* hierarchy only contains the non-terminals M and D of Fig. 2. This set of variables is called *module-bound available variables*. It is stored in the mapping AVM which is calculated by the family of functions $\mathcal{A}^E$, $\mathcal{A}^T$, $\mathcal{A}^M$, $\mathcal{A}^D$ of Fig. 4 (one for each nonterminal in the grammar of Fig. 2).

The second argument of function $\mathcal{A}^E$ is the set of *module*-bound available variables of the outer context of an expression E. The mapping AVM of a program E is defined as $AVM = \mathcal{A}^E\llbracket E \rrbracket \emptyset$, since the outer context is empty.

In equation ($\dagger$) the set of *module*-bound available variables is cleared because at that point a new hierarchy starts. In contrast, $\mathit{\Gamma}$ in equation ($\ddagger$) is not modified. The possibility to define the function $\mathcal{A}^\alpha$ in this simple form was one of the reasons to distinguish between *modules* and other expressions in the syntax.

4.3 Analysis

In Sec. 5, expressions E will be transformed into a new intermediate form with an explicit evaluation order. This transformation needs the dependency order of all expressions. In this section, we will describe the calculation of this ordering

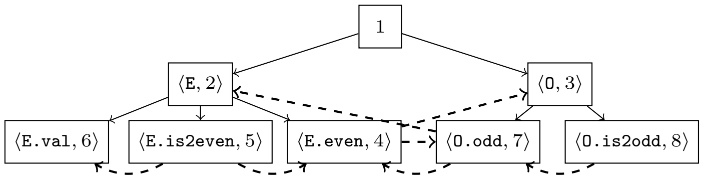


Fig. 5. *Module* tree (without dashed edges) and dependency graph for the example in Listing 1.4

Index	*FN*	*AVM*	*UNM*
4	`O.odd`	E, O, `even`, `is2even`, `val`	`O.odd`
5	`even,val`	E, O, `even`, `is2even`, `val`	`even,val`
6		E, O, `even`, `is2even`, `val`	
7	`E.even`	E, O, `odd`, `is2odd`	`E.even`
8	`odd`	E, O, `odd`, `is2odd`	`odd`

Fig. 6. Mapping label to *module*-bound used names

with the help of the running example. The complete function definitions can be found in [13].

The evaluation order has to be calculated for every uninterrupted *module* hierarchy. The analysis is split into the following three phases:

1. Build *module* tree
2. Calculate the dependency edges
3. Calculate the strongly connected components (e.g. Sharir's algorithm [14])

We use our earlier example of recursive *modules* (see Listing 1.4) to illustrate the analysis.

At the beginning a *module* tree for this uninterrupted *module* hierarchy is built. The tree is shown in Fig. 5. Every node represents a definition. The left-hand side is represented by the complete name within this *module*-hierarchy and the right-hand side by its label. The root of each *module* tree is a special node. It has no name, as the *module* is anonymous. The label is that of the outermost *module*; in our running example it is 1. The children of a node are the definitions of its associated *module*.

In phase two, all dependency edges have to be calculated. We have to consider all leaves. In our example, these are all nodes with labels 4 – 8. First, the set of all free names, where the variable is in the set of *module*-bound available variables must be calculated. This set is called *module-bound used names* (*UNM*). In Fig. 6 all three sets are calculated for each label. In this example, the set of *module*-bound used names is identical to the set of free names. This is true in all cases, as names from the outer context may be used.

For every name in the set *UNM*, the representing node has to be found. The tree is searched upwards from the current node to find the variable that corresponds to the name. For example, we consider the node with label 4 and the *module*-bound used name `O.odd`.

The parent node has no child with the definition `O` so we must look at the next parent. That one has such a child — the node with label 3. Starting with this node, the selector chain is considered. A suitable child must be found for every selector in the chain. In this case, the next selector is `.odd` and the suitable child is node 6. As there are no further selectors, node 6 is the desired node. Therefore, a dependency edge is inserted from 4 to 6. If there is no suitable child, there are two possibilities:

– The current node represents a *module*: this is an error, because the selector will never exist.
– The current node represents another expression: this means the current node is the one we have been searching for (e.g. this is the case for the example in Sec. 3.3[2]).

As all dependency edges are calculated for the *module* tree the recognised static part is equal to the path in the tree.

Additional edges are added from the start node x to all ancestors of the found node y, except for nodes which are also ancestors of x. In this example, an edge from 4 to 3 must be added. These additional edges are necessary to properly handle recursive *modules*, in this case of `E` and `O`. The complete dependency graph is illustrated in Fig. 5.

Sharir's algorithm [14] is used to calculate the strongly connected components (scc) in dependency order for this graph. The result is a list of the sets of nodes. As the top-level *module* is always the last component in the scc list and is never recursive, we can ignore this component. In our example the nodes 2–8, except 6, are in one scc. The result is the following list:

$$\{\langle \texttt{E.val}, 6\rangle\},$$
$$\{\langle \texttt{E}, 2\rangle, \langle \texttt{O}, 3\rangle, \langle \texttt{E.even}, 4\rangle, \langle \texttt{E.is2even}, 5\rangle, \langle \texttt{O.odd}, 7\rangle, \langle \texttt{O.is2odd}, 8\rangle\}$$

The result of the dependency analysis is used in the transformation described in the next Sec. 5.

5 Transformation

In this section we describe the transformation based on the dependency order of Sec. 4.3. Every abstract syntax tree (Fig. 2) is transformed into a tree with explicit dependency order (Fig. 7).

In this syntax, *modules* are represented by a sequence `Seq` of definitions in dependency order. Mutually recursive *modules* are combined in one `Seq` and all their definitions are children of this node. Every definition is either recursive (`RecDef`) or non-recursive (`Def`) or a *module* (`Seq`). All expressions T are represented by S.

[2] The *module* tree for the outer *module* does not contain `a` and therefore the dependency edge for the free name `s.a.b` is the edge from `output` to `s` and so the dependency analysis gives the order `s`, `output`.

$$\begin{aligned} A &::= C \mid S \\ S &::= \lambda\, x \;.\; A \mid x \mid A \;.\; x \mid \ldots \\ C &::= \texttt{Def}\; N\; S \mid \texttt{RecDef}\; (N\; S)^{+} \mid \texttt{Seq}\; C^{*} \end{aligned}$$

Fig. 7. Abstract syntax with explicit dependency order

Only the transformation of *modules* is non-trivial. First, the strongly connected components have to be calculated for the whole *module* as described in the previous section. The result is used to transform the complete *module* hierarchy into the syntax of Fig. 7. For lack of space we omit the concrete transformation function, which can be found in [13].

The result of the transformation for the running example is given in Listing 1.8. The top level group is represented with the outer `Seq`, the mutually recursive *modules* `E` and `O` are combined in the inner `Seq`. In the inner `Seq` all definitions of both *modules* are given in dependency order.

```
Seq (Seq (Def E.val, 2)
         (RecDef (E.even, λn. IF n==0 THEN true ELSE O.odd (n-1))
                 (O.odd, λn. IF n==0 THEN false ELSE E.even (n-1)))
         (Def E.is2even, even val)
         (Def O.is2odd, odd 2))
```

Listing 1.8. Transformed even-odd example.

During the transformation the definitions for mutual recursive *modules* are ordered and so mutual recursion is allowed as long as all mutual recursive definitions are functions (see Sec. 3.2).

We will give a denotational semantics for the result of this transformation in the next section.

6 Denotational Semantics

The result of the transformation is mapped to an interpretation by the evaluation function $\mathcal{E}$. We use the established techniques of denotational semantics [15,16,17] without going into detail.

6.1 Semantic Domain and Auxiliary Functions

We map every expression A to a value V from the following semantic domains:

Values: $\mathsf{V} = \mathsf{N} + \mathsf{S} + \mathsf{B} + \mathsf{H} + \mathsf{F}$

Modules: $\mathsf{H} = x \hookrightarrow \mathsf{V}$ Functions: $\mathsf{F} = \mathsf{V} \to \mathsf{V}$

The set of all values consists of numbers N, strings S, booleans B, *module* values H and function values F. *Module* values H are partial functions with finite domain, mapping variables to values. Although the *module* value can be seen as an evaluation context that maps free variables of an expression to a value, we use a special evaluation context Γ in order to avoid confusion.

$$\mathcal{E} : A \to \Gamma \to \mathsf{H} \to \mathsf{V}$$

$$\mathcal{E}[\![\texttt{Def}\ N\ S]\!]\Gamma\ \mathsf{H} = N \mapsto\!\!\!> (\mathcal{E}[\![S]\!](\Gamma \lhd (\mathsf{H}|_N))\ \emptyset)$$

$$\begin{aligned}\mathcal{E}[\![\texttt{RecDef}\ (N_i\ S_i)^{i\in 1..n}]\!]\Gamma\ \mathsf{H} &= v_1 + \cdots + v_n \\ \text{where}\quad \Phi\,\langle \mathsf{F}_i\rangle^{i\in 1..n} &= \langle \mathcal{E}[\![S_i]\!](\Gamma \lhd (\mathsf{H}'|_{N_i}))\ \emptyset\rangle^{i\in 1..n} \\ &\quad \text{where}\quad v'_i = N_i \mapsto\!\!\!> F_i \\ &\qquad\qquad\quad \mathsf{H}' = \mathsf{H} + v'_1 + \cdots + v'_n \\ \left\langle \mathsf{F}_i^{\,i\in 1..m}\right\rangle &= \mathsf{FIX}\Phi \\ v_i &= N_i \mapsto\!\!\!> F_i\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![\texttt{Seq}\ C_i^{\,i\in 1..n}]\!]\Gamma\ \mathsf{H} &= v_1 + \cdots + v_n \\ &\text{where}\quad v_i = \mathcal{E}[\![C_i]\!]\Gamma\ (\mathsf{H}_{i-1} + v_{i-1}) \\ &\qquad\qquad \mathsf{H}_0 = \mathsf{H}\end{aligned}$$

$$\mathcal{E}[\![E\,.x]\!]\Gamma\ \mathsf{H} = \begin{cases} v, & \text{if } x \rightharpoonup v \in (\mathcal{E}[E]\Gamma\ \mathsf{H}) \\ \text{ERROR}, & \text{otherwise}\end{cases}$$

$$\ldots$$

Fig. 8. Evaluation function

The environment Γ maps all free variables to a value. The operator $\lhd$ combines two environments Γ_1 and Γ_2 where the right mapping overrides definitions of the left. For syntactical distinction between the mapping within a *module* value and the environment, the environment mapping uses $\mapsto$ and the *module* value $\rightharpoonup$. Furthermore, we define three auxiliary functions for *module* values:

– The operation $\mapsto\!\!\!>$ constructs a *module* value for a given name and value.

$$x_1\,.\cdots.\,x_n \mapsto\!\!\!> v = \begin{cases} x_1 \rightharpoonup (x_2\,.\cdots.\,x_n \mapsto\!\!\!> v), & \text{if } n > 1 \\ x_1 \rightharpoonup v, & \text{otherwise}\end{cases}$$

– The operator $+$ combines two *module* values.

$$\begin{aligned}\mathsf{H}_1 + \mathsf{H}_2 = &\{x \rightharpoonup (v_1 + v_2) \mid x \rightharpoonup v_1 \in \mathsf{H}_1 \wedge x \rightharpoonup v_2 \in \mathsf{H}_2\} \\ \cup &\{x \rightharpoonup v_1 \mid x \rightharpoonup v_1 \in \mathsf{H}_1 \wedge x \rightharpoonup v_2 \notin \mathsf{H}_2\} \\ \cup &\{x \rightharpoonup v_2 \mid x \rightharpoonup v_2 \in \mathsf{H}_2 \wedge x \rightharpoonup v_1 \notin \mathsf{H}_1\}\end{aligned}$$

– The projection $|$ creates the environment Γ for the given Name.

$$\mathsf{H}|_{x_1\,.\cdots.\,x_n} = \begin{cases} \Gamma' \lhd v|_{x_2\,.\cdots.\,x_n}, & \text{if } x_1 \rightharpoonup v \in \mathsf{H} \\ \Gamma' & \text{if } n \geq 1 \\ \emptyset, & \text{otherwise}\end{cases}$$
$$\text{where}\quad \Gamma' = \{x \mapsto v \mid x \rightharpoonup v \in \mathsf{H}\}$$

6.2 Evaluation Function

Figure 8 shows the evaluation function for *modules* and definitions. The evaluation function has three arguments: the expression to evaluate, a normal environment mapping variables to values and a *module* value H representing the current *module* hierarchy. All definitions are evaluated to a *module* value representing the path within the current *module* hierarchy. The right-hand side of each definition is evaluated within a fresh (empty) hierarchy and an environment where the current scope is visible. Therefore the current environment is enriched by the projection of the current hierarchy into the scope of the definition. The projection adds all definitions of the current hierarchy that are visible for this definition. Recursive definitions are evaluated, as usual, via a fixpoint operator.

Modules are evaluated to a *module* value containing all inner definitions. The definitions are evaluated in evaluation order, i.e. in the order of the given list. The values of the children are added to the hierarchy value and the next child is evaluated. The result is the combination of all child values.

The result of a selection $E.x$ is either the value v, if x maps to v in the result of the evaluation of E or it is an error. There are two possible errors: the value of E is not a *module* or it does not contain an x.

The rest of the evaluation function is omitted as it is the usual definition with an environment and the hierarchy value is ignored.

7 Related Work

There are various approaches for flexible modularisation in statically and dynamically typed functional languages. We focus here on those combining nested mutually recursive modules and some kind of abstraction for modules — either special functors or first-class modules.

In [2] a module system for Haskell is proposed, where records and modules are joined in one concept. These record-modules are, as in our approach, first-class values with dot notation. As Haskell is typed, these records are typed as well. The type of every record must be given and provides information about all defined names of the module. Allowed selections are detectable by type inference for every expression. As Haskell is lazy it is not necessary to calculate the dependency order.

In [10] a calculus is proposed for first-class modules that are allowed to be mutually recursive. Their approach unifies classes and objects, so this construct equates to our *modules*. In contrast, an undecidable type system is proposed and no explicit semantics is given.

SML provides a special module language. The flexibility of ML modules is given by module functions. Using these functions, one can create new modules and change modules. Mutually recursive modules are not natively supported, but some extensions do allow it.

One of the first approaches for recursive modules in ML was mixin modules [18]. Mixins are, as all ML modules, not first-class values. The dependencies to other modules must be declared within the module. Missing definitions are assigned by gluing modules over the function *sum*. This approach differs from ours, as all dependencies must be given and so the dependency order is explicit.

In [19] Owens and Flatt describe the concept of "units". Units are first-class nested modules and it is possible to define recursive units by a compound similar to mixins. When two or more mutual recursive units are combined a new unit is created containing all definitions of the combined units in the given order and in this order the units are evaluated. This suits well for mutual recursive functions spread over module borders, but it does not allow all kinds of mutual recursive modules. For instance, our even-odd example is not possible, since both modules contain a definition that is not a function. In our approach these inner definitions are sorted for a proper evaluation during the transformation.

In [8] Russo extends Mini-SML by recursive modules. As for all ML-languages, the evaluation order must be given by the programmer. All forward referenced structures must be introduced at the beginning of the recursive structure. During evaluation, those forward references are assumed to be undefined. After evaluation, they are updated in the heap. In this approach an exception is raised if the forward reference is used. Therefore, all occurrences of forward references must be under abstraction. Our running example is not possible in this approach. Although it is type correct, the evaluation would raise an exception, as the tests `is2even` and `is2odd` are not abstractions.

Garrigue and Nakata study in [7] applicative modules with polymorphic functors and recursion based on paths. They use path resolution to find and handle recursion. As they have proven in [9], path resolution is not decidable for flexible systems such as ours.

In his PhD thesis [4], Claus Reinke developed a module system for functional call-by-value languages. His modules are called frames and are quite similar to our *modules*. In particular, they are dynamically typed first-class values. Frames may contain a set of definitions and these definitions may be mutually recursive. Recursion over frame boundaries is not allowed and all names of other frames must be explicitly imported. These imports give the evaluation order for frames. Mutually recursive modules can only be implemented via functions where the relevant modules are parameters.

8 Conclusion and Future Work

The introduced concept of *modules* is a very flexible, expressive and homogeneous mechanism for dynamic data structures, bindings and modularisation. Using *modules* one can define hierarchical, mutually recursive first-class modules. Using a special dependency analysis a transformation based call-by-value semantics is given.

Finally, we have already extended the introduced language ModLang by *module* morphisms. They allow *modules* to not only be created at runtime, but also to be extended or restricted. This improves the flexibility and reuseability of *modules*. Furthermore, we have a mechanism for imports based on a control flow analysis and the ability to control visibility when using imports and exports.

At the moment, we are working on a translation into an untyped functional language with “let”, “letrec” and dynamic records. In addition, we are developing a concept for separate compilation. Furthermore, we will study correctness properties of our algorithms.

Acknowledgement. We would like to thank Peter Pepper and Christoph Höger for helpful discussions and comments about language design and implementation. We are very grateful to Doug Smith of Kestrel Institute, Palo Alto, where most of the presented work was done. Furthermore, we would like to thank the TFP reviewers for the helpful review and Joe Harjung for proof reading and his comments.

References

1. Pepper, P., Hofstedt, P.: Funktionale Programmierung – Sprachdesign und Programmiertechnik. Springer (2006)
2. Peyton Jones, S.L., Shields, M.B.: First class modules for Haskell. In: 9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon, pp. 28–40 (January 2002)
3. Russo, C.V.: First-class structures for standard ml. Nordic J. of Computing 7, 348–374 (2000)
4. Reinke, C.: Functions, Frames, and Interactions – completing a lambda-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments. PhD thesis, Universität Kiel (1997)
5. Blume, M.: Hierarchical Modularity and Intermodule Optimization. PhD thesis, Princeton University (November 1997)
6. Crary, K., Harper, R., Puri, S.: What is a recursive module? SIGPLAN Not. 34(5), 50–63 (1999)
7. Nakata, K., Garrigue, J.: Recursive modules for programming. SIGPLAN Not. 41, 74–86 (2006)
8. Russo, C.V.: Recursive structures for standard ml. SIGPLAN Not. 36(10), 50–61 (2001)
9. Nakata, K., Garrigue, J.: Path resolution for nested recursive modules. Higher-Order and Symbolic Computation, 1–31 (May 2012)
10. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 201–224. Springer, Heidelberg (2003)
11. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 report (revised). ACM SIGPLAN Notices 27(8), 15–42 (1992)
12. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs, 2nd edn. MIT Press (1996)
13. Lorenzen, F., Rohloff, J.: Gruppen: Ein Ansatz zur Vereinheitlichung von Namensbindung und Modularisierung in strikten funktionalen Programmiersprachen (Langfassung). Technical Report 2011-12, TU Berlin (2011)
14. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications 7(1), 67–72 (1981)
15. Tennent, R.D.: The denotational semantics of programming languages. Commun. ACM 19(8), 437–453 (1976)
16. Gordon, M.J.C.: The Denotational Description of Programming Languages. Springer (1979)
17. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Computer Science Series. MIT Press (1981)
18. Hirschowitz, T., Leroy, X.: Mixin modules in a call-by-value setting. ACM Trans. Program. Lang. Syst. 27, 857–881 (2005)
19. Owens, S., Flatt, M.: From structures and functors to modules and units. SIGPLAN Not. 41(9), 87–98 (2006)

The Design of a Practical Proof Checker for a Lazy Functional Language

Adam Procter[1,⋆], William L. Harrison[1], and Aaron Stump[2]

[1] Dept. of Computer Science, University of Missouri, Columbia, Missouri, USA
[2] Dept. of Computer Science, University of Iowa, Iowa City, Iowa, USA

Abstract. Pure, lazy functional languages like Haskell provide a sound basis for formal reasoning about programs in an equational style. In practice, however, equational reasoning about correctness proofs is underutilized. In the context of Haskell, we suggest that part of the reason for this is the lack of accessible tools for machine-checked equational reasoning. This paper outlines the design of MProver, a proof checker which fills just that niche. MProver features first-class support for reasoning about potentially undefined computations (particularly important in a lazy setting), and an extended notion of Haskell-like type classes, enabling a highly modular style of program verification that closely follows familiar functional programming idioms.

1 Introduction

The grand promise of pure functional languages is a mathematically rigorous style of programming—a style in which the meaning of a program is defined precisely and compositionally, and program properties may be reasoned about statically according to intuitive yet precise laws. The use of a lazy or non-strict semantics, as exemplified by Haskell, enables a wide array of proof techniques based on the simple unifying principle of *equational reasoning*: if it can be shown that subterm t in a program always evaluates to the same thing as t', we may subsitute t with t' without fear of changing the program's meaning in subtle ways.

The strong, static type system of Haskell is a highly successful example of "lightweight formal methods", capable of detecting and preventing many kinds of programming errors. However, it does not have the power to express, let alone enforce, many useful properties that can be proved via external equational reasoning. Yet there is a gap when it comes to tools: while the Haskell type checker automatically decides whether a program is well typed, few tools in widespread use support automatic checking of equational reasoning proofs. We believe that this lack is a serious obstacle to broader adoption of equational reasoning, and that developing such a tool would make equational reasoning accessible to a wider audience.

⋆ Supported by the US Department of Education under grant number P200A100053.

This paper describes the design of a new system called *MProver* for proving equational properties of programs in a pure, lazy, functional language. Our main motivation in developing MProver is to support machine-checked equational reasoning proofs about programs using "pure" monads (i.e. monads other than *IO*) – hence the *M* in *MProver.* The system is, however, useful for all kinds of functional programming idioms; it is not limited, nor even specifically tailored, to monadic programs.

In comparing MProver to related systems such as Coq [1], Agda [27], and Sparkle [2], three design decisions stand out:

Type Classes Are Extended with Proof Obligations. Haskell's type class system enables programs to be written in terms of signatures, rather than particuar type structures. Perhaps the most common example is that of monads, which can be used to model a wide variety of "notions of computation", such as I/O, mutable state, and nondeterminism. Programs written to target the *Monad* type class can then be reused in any computational setting, and new computational settings may be added to the *Monad* class as long as operations for sequencing (*bind*) and for injection of effect-free computations (*return*) are defined.

In general, a type class is associated not just with a set of type signatures, but also with an implicit *specification* or *contract* governing how the operations are supposed to behave; for example, *Monad* instances are, very loosely stated, supposed to have the properties that sequential composition of computations is associative and that *return* is a left- and right-unit with respect to sequential composition. Haskell's type system does not check these properties; indeed, it cannot even express them directly. MProver makes the contract explicit; it augments the *Monad* class by adding *proof obligations* for the monad laws, as pictured in Figure 1. This approach allows not just programs, but also proofs, to be parameterized over all monadic notions of computation, enabling a modular style of proving that closely parallels the familiar vocabulary of functional programming idioms.

"Bottom" as a First-Class Citizen. In a lazy language, undefinedness is an ever-present concern, in that variables are not necessarily bound to well-defined values. For this reason, MProver, like Sparkle, treats the undefined value (that is, the value of diverging or erroring computations) as a "first-class citizen." For example, proofs by case analysis must consider as one case the possibility that the expression being analyzed is undefined. By the same token, properties like "*f* is strict in its third argument" can be expressed directly in the logic.

More General Notion of Equality for Potentially Infinite Structures. Systems like Coq have a rather restrictive notion of equality for coinductive types. Equality in Coq is intensional: that is, two expressions of the same type are equal if and only if they evaluate to the same normal form. Coq will generally refuse to β-reduce applications of functions producing a coinductive type, since this may result in nontermination and thus compromise logical soundness. Thus when working with coinductive types, one generally must define a weaker notion of *bisimulation* in lieu of equality *per se.* Because MProver separates the universes of programs and proofs—program terms are only an *object* of logical reasoning and are not themselves treated as logical proofs—it is possible to define a notion of equality over infinite structures that is much easier to deal with.

```
class Monad m where
  (>>=)     :: m a → (a → m b) → m b
  return    :: a → m a
  leftunit  ::: ∀ (x :: a) (f :: a → m b), return x >>= f = f x
  rightunit ::: ∀ (x :: m a), x >>= return = x
  assoc     ::: ∀ (x :: m a) (f :: a → m b) (g :: b → m c),
                (x >>= f) >>= g = x >>= λy → f y >>= g


instance Monad Maybe where
  (Just x) >>= f   = f x
  Nothing >>= _    = Nothing
  return           = Just
  leftunit         = leftunitMaybe
  rightunit        = rightunitMaybe
  assoc            = assocMaybe


leftunitMaybe   = Foralli (x :: a) (f :: a → Maybe b),
  join ::: return x >>= f = f x

rightunitMaybe = Foralli (x :: Maybe a), case x of
  undefined → join ::: undefined >>= return = undefined
  Nothing   → join ::: Nothing >>= return = Nothing
  Just v    → join ::: Just v >>= return = Just v
assocMaybe     = Foralli (x :: Maybe a) (f :: a → Maybe b) (g :: b → Maybe c), case x of
  undefined → join ::: (undefined >>= f) >>= g = undefined >>= (λy → f y >>= g)
  Nothing   → join ::: (Nothing >>= f) >>= g = Nothing >>= (λy → f y >>= g)
  Just v    → join ::: (Just v >>= f) >>= g = Just v >>= (λy → f y >>= g)
```

Fig. 1. The *Monad* type class in MProver, and an example instance

The remainder of this paper proceeds as follows. Section 2 gives a definition of the core language of programmatic expressions, logical formulas, proof terms, and tactics, and concludes with a few simple examples. Section 3 sketches the extension of the language of Section 2 with type classes, illustrated by a monadic equational reasoning proof adapted from Gibbons and Hinze [3]. Section 4 discusses related work, and Section 5 concludes.

2 Expressions, Formulas, Proofs, and Tactics in MProver

In this section we describe the basic design of the MProver language. We shall defer discussion of the type class system to Section 3. The language may be divided into two parts: the *programming* fragment and the *proving* fragment. The programming fragment is a pure, lazy functional language with type classes—essentially a subset of Haskell 98 [4]. In the proving fragment, proofs are expressed as terms in a λ-calculus-like language, but this language is entirely distinct from the language of programs and is not intended for evaluation other than possibly for metatheoretic purposes.

At the outermost level, an MProver program is a *script* containing both definitions of type, type classes, and variables (just as in Haskell), and proofs of properties of those definitions (written in an MProver-specific proof notation). A script consists of one or more top-level declarations. A declaration may be a datatype declaration, a program term declaration, or a theorem declaration. The syntax for datatype and program term declarations is identical to (a subset of) Haskell, while theorem declarations have the form:

$$\begin{array}{l} \mathit{var} ::: \mathit{formula} \\ \mathit{var} = \mathit{proof} \end{array}$$

Here the *var* is the name of the theorem, the *formula* is the definition of the theorem, and the *proof* is a proof of the theorem. The symbol ::: should be read as "proves the formula", by analogy with the Haskell symbol :: which is read "has type". The languages of formulas and proofs, and the relationship between them, are discussed in Section 2.3.

2.1 Expressions

As a matter of terminology, we will refer to programmatic terms as *expressions* and proof terms as *proofs*. We will consider a subset of the full expression language; the grammar of this subset is given in Figure 2. The full language is more fully featured than this, but the simpler formulation of Figure 2 will suffice for our discussion of the essential characteristics of the language.

$$\begin{array}{rcll}
\mathit{expr} & ::= & \lambda\ \mathit{var} \rightarrow \mathit{expr} & \text{(abstraction)} \\
 & \mid & \mathit{expr}\ \mathit{expr} & \text{(application)} \\
 & \mid & \mathit{var} & \text{(variables)} \\
 & \mid & \mathit{ctor} & \text{(data constructors)} \\
 & \mid & \mathbf{let}\ (\mathit{var} = \mathit{expr})^*\ \mathbf{in}\ \mathit{expr} & \text{(let-binding)} \\
 & \mid & \mathbf{case}\ \mathit{expr}\ \mathbf{of}\ (\mathit{pat} \rightarrow \mathit{expr})+ & \text{(case)} \\
 & \mid & \mathbf{undefined} & \text{(undefined value)} \\
 & & & \\
\mathit{pat} & ::= & \mathit{var} & \text{(variables)} \\
 & \mid & \mathit{ctor}\ \mathit{pat}^* & \text{(constructed values)} \\
 & \mid & \_ & \text{(wildcard)}
\end{array}$$

Fig. 2. Grammar for MProver Expressions

The semantics of MProver expressions is a standard non-strict semantics. As in Haskell, **let** bindings are recursive.

2.2 Formulas and Proofs

The main design goal of MProver is to support equational reasoning. For this reason, the language of program properties—called *formulas*—comprises just statements of (in)equality between program terms, logical implication, and universal quantification over expressions. The grammar of formulas is given in Figure 3. Note that the ∀ symbol in formulas is unrelated to the `forall` keyword used for higher-rank polymorphism in Haskell.

$$
\begin{array}{rcll}
\mathit{formula} & ::= & \forall\ (\ \mathit{var}\ ::\ \mathit{type}\ ),\ \mathit{formula} & \text{(universal quantification)} \\
 & | & \mathit{formula} \rightarrow \mathit{formula} & \text{(implication)} \\
 & | & \mathit{expr} = \mathit{expr} & \text{(equality)} \\
 & | & \mathit{expr} \neq \mathit{expr} & \text{(inequality)}
\end{array}
$$

Fig. 3. Grammar for MProver Formulas

Figure 4 gives the grammar of proof terms. In order: the **Foralli** and **Assume** forms are used to introduce variables ranging over expressions and proofs of formulas, respectively. These variables may be referenced inside of a proof; universally quantified proof terms may be instantiated by expressions; implication proofs may be applied to proofs of the antecedent (modus ponens); proof expressions may be annotated with formulas as a hint to the proof checker; and **case**-proofs allow casewise splitting on program terms. Of the remaining proof forms, **eval**, **clash**, **refl**, **trans**, and **symm** may be viewed as directly reflecting equational judgments that follow from the reduction semantics. The **cong** form is used for congruence proofs (that is, substituting equals for equals).

$$
\begin{array}{rcll}
\mathit{proof} & ::= & \textbf{Foralli}\ (\ \mathit{var}\ ::\ \mathit{type}\ ),\ \mathit{proof} & \text{(forall-introduction)} \\
 & | & \textbf{Assume}\ (\ \mathit{var}\ :::\ \mathit{formula}\ ),\ \mathit{proof} & \text{(assumption)} \\
 & | & \mathit{var} & \text{(variables)} \\
 & | & \mathit{proof}\ \ \mathit{expr} & \text{(application 1)} \\
 & | & \mathit{proof}\ \ \mathit{proof} & \text{(application 2)} \\
 & | & \mathit{proof}\ :::\ \mathit{formula} & \text{(annotation)} \\
 & | & \textbf{case}\ \mathit{expr}\ \textbf{of}\ (\mathit{pat}_{\perp} \rightarrow \mathit{proof})^{*} & \text{(pattern matching)} \\
 & | & \textbf{eval} & \text{(evaluation)} \\
 & | & \textbf{clash} & \text{(constructor-clash)} \\
 & | & \textbf{cong}\ \mathit{proof} & \text{(congruence)} \\
 & | & \textbf{refl} & \text{(reflexivity)} \\
 & | & \textbf{trans}\ \mathit{proof}\ \ \mathit{proof} & \text{(transitivity)} \\
 & | & \textbf{symm}\ \mathit{proof} & \text{(symmetry)} \\
 & & & \\
\mathit{pat}_{\perp} & ::= & \mathit{var} & \text{(variables)} \\
 & | & \mathit{ctor}\ \mathit{pat}_{\perp}^{*} & \text{(constructed values)} \\
 & | & \_ & \text{(wildcard)} \\
 & | & \textbf{undefined} & \text{(bottom)}
\end{array}
$$

Fig. 4. Grammar for MProver Proofs

2.3 Classification Rules

The process of proof checking, that is determining that a proof proves a formula, is called *classification*, and is akin to type checking. The classification rules are given in Figure 5. Contexts Γ contain assumptions of three forms: (1) $x ::: \phi$, indicating that variable x ranges over proofs of formula ϕ; (2) $x :: t$, indicating that variable x ranges over expressions of type t; and (3) $x = e$, indicating that variable x is bound to expression e. The "initial" Γ used by the proof checker introduces all assumptions of the third type by binding top-level symbols to their definitions; there is no let-binding construct for proof terms.

The relation $\rightsquigarrow_\Gamma$, used in the rule for **eval**, is a single-step, call-by-name reduction relation on open terms. We subscript the relation with Γ so that the definitions of top-level expressions may be expanded on an as-needed basis. When performing reduction inside a λ or **let** expression, any definitions that are shadowed by the local bindings are dropped from Γ. Variables not bound in Γ are simply left as is, rather than expanded. The relation contains special rules for handling **undefined**: in particular, **undefined** $e \rightsquigarrow_\Gamma$ **undefined**, pattern-match failure reduces to **undefined**, and if a case expression forces evaluation of an **undefined** scrutinee, the entire case-expression reduces to **undefined**.

The rules in the left column are essentially standard rules for assumption, abstraction, and application. The rule for **case** proofs has a couple of important features: first, pattern matching must be exhaustive, and the **undefined** case must be considered. Second, the formula ϕ that is proven inside the case alternatives is parameterized by the different cases: for example, if a proof of the formula $\forall\ (b :: \mathit{Bool}), \mathit{not}\ (\mathit{not}\ b) = b$ is done by case analysis of b, the body of the respective case alternatives must prove $\mathit{not}\ (\mathit{not}\ \mathit{True}) = \mathit{True}$, $\mathit{not}\ (\mathit{not}\ \mathit{False}) = \mathit{False}$, and $\mathit{not}\ (\mathit{not}\ \mathbf{undefined}) = \mathbf{undefined}$. We assume, for the sake of brevity, that all patterns are either a constructor applied to variable patterns, **undefined**, or the wildcard pattern (i.e. that patterns are not nested).

With respect to **cong**, note that due to MProver's lazy semantics, this one rule enables us to do all kinds of substitutions. For example, if we know that $f = g$ and we want to use this to prove that $\mathit{map}\ f\ \mathit{xs} = \mathit{map}\ g\ \mathit{xs}$, we can just make up a new λ-abstraction, with a fresh variable representing the substitution sites, and use **cong** apply that to our proof that $f = g$. A few **eval** steps will then give us the substitution we desire. The price we pay for this parsimony is that it is a bit tedious if such proofs done by hand; however, the **subst** tactic described below automates this process.

2.4 Recursive Data Structures

The presence of recursive data structures necessitates some kind of support for coinduction. For this, we turn to Coq for inspiration. As an example, consider the type of lazy lists: Figure 6 gives two definitions. The first is the definition of the lazy list data type; the `CoInductive` keyword indicates that `LLists` may possibly be infinite. The second definition is of a *bisimulation* – that is, a coinductively defined predicate that is weaker (i.e. identifies strictly more expressions) than Coq's definitional equality.

To support reasoning about potentially infinite codata in MProver, each constructor in a data-declaration induces an analogous constructor form for equality proofs. In the example of Figure 7, the equality-proof forms for lazy lists are given; notice that they have essentially the same form as the constructors for the Coq bisimulation. In MProver, we simply overload the symbols *Nil* and *Cons* as constructors for a coinductively-defined bisimulation on lazy lists. This is the same bisimulation that we used in the Coq example: intuitively, if we have a proof p that expressions e and e' are equal, and a proof p_l that lists l and l' are equal, then just as we can "cons" e and e' onto l and l', so too can we cons p onto p_l, thus proving $e : l = e' : l'$.

$$\frac{(x ::: \phi) \in \Gamma}{\Gamma \vdash x ::: \phi} \qquad \frac{e_1 \rightsquigarrow_\Gamma e_2}{\Gamma \vdash \mathbf{eval} ::: e_1 = e_2}$$

$$\frac{\Gamma, (x :: t) \vdash p ::: \phi}{\Gamma \vdash \mathbf{Foralli}(x :: t),\ p ::: \forall(x :: t), \phi} \qquad \frac{C \neq C'}{\Gamma \vdash \mathbf{clash} ::: C\ e_1 \cdots\ e_n \neq C'\ e'_1 \cdots\ e'_m}$$

$$\frac{\Gamma \vdash p ::: e' = e''}{\Gamma \vdash \mathbf{cong}\ p ::: e\ e' = e\ e''}$$

$$\frac{\Gamma, (x ::: \phi) \vdash p ::: \phi'}{\Gamma \vdash \mathbf{Assume}(x ::: \phi),\ p ::: \phi \to \phi'} \qquad \frac{}{\Gamma \vdash \mathbf{refl} ::: e = e}$$

$$\frac{\Gamma \vdash p ::: \forall(x :: t), \phi \quad \Gamma \vdash e :: t}{\Gamma \vdash p\ e ::: [x/e]\phi} \qquad \frac{\Gamma \vdash p_1 ::: e_1 = e_2 \quad \Gamma \vdash p_2 ::: e_2 = e_3}{\Gamma \vdash \mathbf{trans}\, p_1\, p_2 ::: e_1 = e_3}$$

$$\frac{\Gamma \vdash p ::: e_1 = e_2}{\Gamma \vdash \mathbf{symm}\, p ::: e_2 = e_1}$$

$$\frac{\Gamma \vdash p ::: \phi \to \phi' \quad \Gamma \vdash p' ::: \phi}{\Gamma \vdash p\ p' ::: \phi'}$$

$$\frac{\begin{array}{c} \Gamma, x_{11} :: p_{11} \cdots x_{1m_1} :: t_{1m_1} \vdash p_1 ::: [x/(C_1\ x_{11}\ \cdots\ x_{1m_1})]\phi \\ \vdots \\ \Gamma, x_{n1} :: t_{n1} \cdots x_{nm_n} :: t_{nm_n} \vdash p_n ::: [x/(C_n\ x_{n1}\ \cdots\ x_{nm_n}]\phi \\ \Gamma \vdash p_\perp ::: [x/\mathbf{undefined}]\phi \end{array}}{\Gamma \vdash \left(\begin{array}{lll} \mathbf{case}\ e\ \mathbf{of} & (C_1\ x_{11}\ \cdots\ x_{1m_1}) & \to p_1 \\ & \cdots & \\ & (C_n\ x_{n1}\ \cdots\ x_{nm_n}) & \to p_n \\ & \mathbf{undefined} & \to p_\perp \end{array} \right) ::: [x/e]\phi}$$

(Note: The **case** rule assumes that $C_1 \cdots C_n$ are all the constructors for some data type, that $C_1 \cdots C_n$ have arities $m_1 \cdots m_n$ respectively, that no x_{ij} is free in ϕ, and that x is not the same variable any x_{ij}.)

Fig. 5. Proof Classification Rules

```
CoInductive LList (A:Type) : Type :=
| LCons : A -> LList A -> LList A
| LNil  : LList A.

CoInductive bisim (A:Type) : LList A -> LList A -> Prop :=
| bisim_LCons : forall (x:A) (l1 l2:LList A),
    bisim l1 l2 -> bisim (LCons x l1) (LCons x l2)
| bisim_LNil  :
    bisim (LNil A) (LNil A).
```

Fig. 6. Lazy lists and bisimilarity in Coq

$$\mathbf{data}\ \mathit{List}\ a\ =\ \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

$$\frac{}{\Gamma \vdash \mathit{Nil} ::: \mathit{Nil} = \mathit{Nil}} \qquad \frac{\Gamma \vdash p ::: x = x' \quad \Gamma \vdash ps ::: xs = xs'}{\Gamma \vdash \mathit{Cons}\, p\, ps ::: \mathit{Cons}\, x\, xs = \mathit{Cons}\, x'\, xs'}$$

Fig. 7. Lazy lists and their coinduction rules in MProver

Guardedness. Coinductive proofs, then, are constructed as corecursive proof terms—that is, they are defined in terms of themselves. As always, this raises a red flag: unrestricted use of recursion would render MProver's logic unsound, allowing us to prove any theorem by appeal to itself. In order to mantain soundness, we require that all recursive proof applications use *guarded* recursion. Here again, *guardedness* is the same guardedness condition used by Coq [5]. In a nutshell, this means that any recursive application of a proof term must occur inside an equality constructor application. The proof of Figure 8, to be discussed in greater detail shortly, illustrates this: the recursive application of `mapfusion` is an immediate argument to `Cons`, and thus the recursion is guarded.

2.5 Tactics and Syntactic Sugar

The core proof language of Figure 4, while quite expressive, is a bit inconvenient when large numbers of evaluation steps must take place. Consider, for example, the simple property $id\ (id\ id) = (id\ id)\ id$, where id is a global symbol defined as $id = \lambda x \rightarrow x$. Probably the most obvious proof of this requires four **eval** steps to reduce both sides of the equation to $id\ id$, and an application of **symm** to link the two halves of the proof together. The result may not be intimidating to the proof checker, but from a human's point of view it is rather tedious and unpleasant:

```
trans (eval ::: id (id id) = (\ x -> x) (id id))
      (trans (eval ::: (\ x -> x) (id id) = id id)
             (symm (trans (eval ::: (id id) id = ((\ x -> x) id) id)
                          (eval ::: ((\ x -> x) id) id = id id))))
```

For this reason, we extend the core language with a tactic called **join**. Given left- and right-hand side expressions e_1 and e_2, the **join** tactic works by repeatedly applying reduction steps to e_1 and e_2 until one of the following happens:

- e_1 and e_2 reach e_1' and e_2', respectively, where e_1' and e_2' are α-equivalent. Then **join** succeeds.
- e_1 and e_2 reach $C_1\ e_{11} \cdots\ e_{1n}$ and $C_2\ e_{21} \cdots\ e_{2m}$ where C_1 and C_2 are different constructors. Then **join** fails.
- e_1 and e_2 reach $C_1\ e_{11} \cdots\ e_{1n}$ and $C_2\ e_{21} \cdots\ e_{2n}$ where some e_{1i} is not α-equivalent to e_{2i}. Then for each e_{1j} and e_{2j} such that e_{1j} and e_{2j} are not α-equivalent, recursively attempt to join e_{1j} with e_{2j}. If all the recursive calls succeed, then **join** succeeds. If any recursive call fails, then **join** fails.

This procedure, which happens during the proof-checking phase, produces a proof term like the one given above; this reduces the burden on the user, who would otherwise have to construct tedious step-by-step reduction proofs by hand. At the same time, it does not complicate the underlying theory, since even if the tactic succeeds the generated proof term will still be checked according to the rules of Figure 2.3.

A second tactic, called **subst**, comes in handy when it is necessary to rewrite underneath constructors or inside λ-abstractions; given a proof p that $e_1 = e_2$, **subst** will construct a proof that $e = e'$ if e' can be obtained by substituting all occurrences of e_1 with e_2 in e. This tactic makes extensive use of the **cong** rule.

2.6 Syntactic Sugar for `trans`

The applicative syntax for constructing **trans**-proof terms is a little bit unwieldy in practice. This is unfortunate, considering that equational reasoning proofs, often quite transitivity-heavy, are exactly what MProver is meant for! To ameliorate this, we supply a little bit of syntactic sugar, vaguely inspired by Haskell's **do**-notation for monads. Any proof term of the form:

```
[ e1 = e2 { p1 } ... = en { pn-1 } ]
```

will be desugared to:

```
(trans (trans (trans ... (p1 ::: {e1 = e2}) (p2 ::: {e2 = e3}))
                     ... (pn-1 ::: {en-1 = en})))
```

The result, illustrated in Figure 8 and in Figure 10, bears a reassuringly close resemblance to a “textbook” equational reasoning proof.

2.7 Example: Map Fusion

Having built up the requisite machinery, we can now present a more involved example of an MProver proof, given in Figure 8. The proof is of the familiar map fusion property, over the lazy list type defined previously in Figure 7. Assume that `map` and the function composition operator `.` are defined in the standard way. The proof then breaks down into three cases: one where the input list is undefined, one where it is empty, and one where it is a cons cell. In the first two cases, the desired property follows simply from evaluation (`join`). The third case is slightly more complicated, requiring the use of coinduction. Here the first use of (`join`) pulls the `Cons` constructor out front, and combines the applications of `g` and `f` with function composition. We then appeal to the coinduction rule for `Cons` to rewrite the tail of the list into the desired form. In the final step, simple evaluation gives us the result we want.

```
mapfusion ::: Forall (f::a -> b) (g::b -> c) (l::List a),
  map g (map f l) = map (g . f) l
mapfusion = Foralli (f::a -> b) (g::b -> c) (l::List a),
  case l of
    undefined -> join ::: map g (map f undefined) = map (g . f) undefined
    Nil       -> join ::: map g (map f []) = map (g . f) []
    Cons x xs ->
      [ map g (map f (Cons x xs))
      = Cons ((g . f) x) (map g (map f xs)) { join }
      = Cons ((g . f) x) (map (g . f) xs)   { Cons refl
                                                   (mapfusion f g xs) }
      = map (g . f) (Cons x xs)             { join }
      ]
```

Fig. 8. Map fusion

3 Type Classes

Let us now turn our attention to MProver's extended notion of type classes. Type classes were introduced in Haskell to allow for ad-hoc polymorphism—that is, overloading of functions and operators—in a natural, extensible way. Viewed another way, type classes can be seen as supporting *modularity* and *abstraction* through well-defined interfaces: the programmer may declare a type to be an instance of any type class simply by supplying a set of functions or operators with the right type signature; programs whose types are parameterized over members of that class can then be reused on new instances.

The Haskell community has developed a rich vocabulary of functional programming abstractions grounded in abstract algebra and category theory, and type classes are the language in which these abstractions are expressed [6]. However, a type class is often associated not just with a set of type signatures, but also with one or more *laws*. For example, any instance type a of the class *Monoid* must be associated with operators $\mathit{mempty} :: a$ and $\mathit{mappend} :: a \to a \to a$. [1]

[1] There is also a third operator $\mathit{mconcat} :: [a] \to a$, which has a default implementation that may be overridden if desired for reasons of efficiency.

This requirement is enforced by Haskell's type system. It is *also* expected, however, that the operators follow certain laws: namely that *mempty* is a left and right identity with respect to *mappend*, and *mappend* is associative. This requirement is not checked mechanically by, nor even expressible in, Haskell's type system.

MProver supports richer specifications by extending type classes with associated formulas expressing the laws that instances of the class must obey. Any declared instance must contain not only definitions for the operators, but also proofs that the operators follow the laws. Figure 1 illustrates a well-known example of a type class in Haskell, that of *monads*[2], extended with proof obligations for the monad laws, along with a particular instance of that class (the *Maybe* type familiar to Haskell programmers).

3.1 Monadic Equational Reasoning

Our approach to monadic equational reasoning shares much with that taken by Gibbons and Hinze [3]. In particular, we program and prove in terms of interfaces that axiomatize the behaviors of particular monadic effects, rather than constructing a particular implementation of that interface (though this can certainly be done). Figure 9 contains an MProver definition of Gibbons and Hinze's *MonadFail* and *MonadExcept* classes: that is, subclasses of *Monad* supporting the throwing and handling of exceptions. In Figure 10, we use this to prove the purity (meaning no uncaught exceptions) of a "fast product" function which takes the product of a list, but first scans the list, throwing an exception if a zero is found; this exception will be caught, and the function will return zero in this case. Note that the fact that *fastprod*'s codomain is in *MonadFail* makes the proof instances for *exceptLeftUnit* and *exceptPure* available.

As it happens, there are a few stipulations having to do with definedness that are not emphasized that strongly in the cited work, but must be made explicit here (though proofs are omitted for space reasons). In particular, we require that scanning the list for zero—that is, evaluating the expression `0 ‘elem‘ xs`—will terminate. Second, we must stipulate that the product function itself is short-circuiting: the obvious definition `foldr (*) 1` will not work, because it is possible that this will diverge even when `0 ‘elem‘ xs = True`. These sorts of stipulations seem to come up sufficiently often in real-world equational proofs that it might well make sense to extend the logic to make them explicit; this is discussed further in Section 5.

4 Related Work

A major antecedent of this work is the Operational Type Theory implemented in the GURU programming language [7]. The most salient feature of Operational Type Theory for our purposes is its treatment of undefined computations:

[2] We omit *fail* from our definition since it is not really part of the mathematical notion of a monad, though its inclusion would cause no problems.

```
class Monad m ⇒ MonadFail m where
  fail          :: m a
  failLeftZero ::: ∀ (x :: m a), fail ≫ x = fail
class MonadFail m ⇒ MonadExcept m where
  catch            :: m a → m a → m a
  exceptLeftUnit  ::: ∀ (x :: m a), catch fail x = x
  exceptRightUnit ::: ∀ (x :: m a), catch x fail = x
  exceptAssoc     ::: ∀ (x y z :: m a), catch x (catch y z) = catch (catch x y) z
  exceptPure      ::: ∀ (x :: a) (y :: m a), catch (return x) y = return x
```

Fig. 9. The *MonadFail* and *MonadExcept* classes

because OpTT directly encodes the (finite) sequences of reductions that are required to establish equivalence of terms, the presence of nonterminating computations does not compromise the soundness of its proving fragment. There is much work on dealing with non-termination and infinite structures in existing theorem proving systems. Coq [1] features support for coinductive types [5], encompassing both coinductive data structures and predicates. A similar design is used for coinduction in Agda. Neither Coq nor Agda, however, supports direct reasoning about undefined computations, as MProver and Guru do. Sparkle [2] supports reasoning about undefined computations, as well as reasoning about infinite structures via structural induction guarded by admissibility. Indirect approaches to dealing with undefined computations include the formalization of domain theory within Coq [8], and extensions to Coq's type theory [9, 10].

MProver is certainly not the first tool designed to support integrated development and verification of functional programs. Particularly closely related to MProver is the Sparkle prover for the Clean programming language [2]. Like MProver, Sparkle has first-class support for reasoning about undefined computations: just as **undefined** is essentially treated as a constructor for all data types in MProver, Sparkle introduces a special expression form, denoted $\bot$, for talking about undefined computations. Sparkle is built on a sophisticated system of tactics and hints, which often results an automatic or near-automatic proving process where typical properties of functional programs are concerned.

One difference between MProver and Sparkle lies in the semantic foundations. Reasoning in Sparkle takes place internally in a simplified version of the Clean language called Core Clean. MProver, by contrast, does not simplify programs to a core language. The semantics of Core Clean is based on lazy graph rewriting, whereas MProver uses what is essentially a call-by-name reduction semantics. Considerable work has also been done on using type classes in Sparkle [11], which should enable a type class-directed style of proving (one of "proving to specifications" rather than proving to structures) very similar to MProver. However, Sparkle does not extend type classes with logical specifications like MProver does. Instead, it relies on a clever scheme of induction over the sets of defined instances; this allows proofs to leverage the semantic relationships among derived typeclass instances (e.g. if the *Eq* instance for type *T* is an equivalence

```
ifredundant ::: Forall (b::Bool) (e::a),
                Assuming b =/= undefined,
                  if b e e = e
productspec ::: Forall (xs::[Int]),
                Assuming 0 ‘elem‘ xs = True,
                  0 = product xs
condlift ::: MonadExcept m =>
             Forall (b::Bool) (m1 m2 m3::m a),
             Assuming b =/= undefined,
                 catch (if b then m1 else m2) m3
               = if b then catch m1 m3 else catch m2 m3

fastprodpure ::: Forall xs::[Int],
                 Assuming 0 ‘elem‘ xs =/= undefined,
                   fastprod xs = return (product xs)
fastprodpure =
 Foralli xs::[Int],
 Assume (zerodecidable:::0 ‘elem‘ xs =/= undefined),
  [ fastprod xs
  = catch (work xs) (return 0)    { join }
  = catch
      (if 0 ‘elem‘ xs
         then fail
         else return (product xs))
      (return 0)                  { subst by (workspec xs) }
  = if 0 ‘elem‘ xs
      then catch fail (return 0)
      else catch
             (return (product xs))
             (return 0)           { condlift
                                      (0 ‘elem‘ xs) fail
                                      (return (product xs)) (return 0)
                                      zerodecidable }
  = if 0 ‘elem‘ xs
      then return 0
      else catch
             (return (product xs))
             (return 0)           { subst by (exceptLeftUnit (return 0)) }
  = if 0 ‘elem‘ xs
      then return 0
      else return (product xs)    { subst by (exceptPure (product xs)) }
  = if 0 ‘elem‘ xs
      then return (product xs)
      else return (product xs)    { subst by productspec }
  = return (product xs)           { ifredundant
                                      (0 ‘elem‘ xs)
                                      (product xs)
                                      zerodecidable }
  ]
```

Fig. 10. MProver formalization of Gibbons and Hinze's “fast product” function

relation, so too is the instance for type $[T]$). The advantage of this approach is that reasoning in Sparkle can be viewed as "external" to an even greater degree than in MProver; reasoning about type class-based programs is available even if the original program is completely innocent to logical specifications. Finally, it is worth noting that Sparkle has support for explicit strictness annotations [12]. This provides a very clean and reasonable alternative to the kind of "definedness stipulations" that permeate the proof of Figure 10. In Section 5 we will briefly sketch a similar approach to this problem that we are considering for MProver.

Type classes originated in Haskell [13] as a means of enabling ad-hoc polymorphism, and the idea has been reimplemented in Coq [14] and Isabelle/HOLCF [15]. The Haskell community has developed a number of theories and tools [16] for formal (and semi-formal) reasoning about Haskell programs. Recent work on static contract checking [17, 18] focuses on the automatic verification of pre- and post-conditions for Haskell functions. SmallCheck [19] is a type-directed framework for automated testing of program properties, similar to QuickCheck [20] but exhaustively testing all values of a type up to a certain "depth", rather than randomly generating test cases. Other tools include the Haskell Equational Reasoning Assistant [21], an AJAX-based tool for rewriting Haskell expressions while preserving semantic equality. The HasCASL project [22] has developed an extension of the algebraic specification language CASL with Haskell-like language constructs. Further investigations of logical aspects of Haskell-like languages, especially with regard to laziness, may be found in [23, 24].

5 Summary and Future Work

This paper has outlined the design of a proof-checking system for a lazy functional language, called MProver. The work described here is still at an early stage, but we expect that MProver will be a very useful tool for mechanized equational reasoning. As we develop our implementation further, we are interested in adapting more examples of pen-and-paper proofs already in the literature and mechanizing them with MProver.

A preliminary implementation of MProver is available from HackageDB, containing a few examples that may be of interest to the reader. As of this writing, we do not have a full development of MProver's metatheory. However, we believe that a soundness argument may be derived from Harrison and Kieburtz's semantics for Haskell, in a similar fashion to P-logic [23,24]. P-logic takes the view that if P is a predicate over (possibly undefined) values of a type T, then P *refines* T: that is, the denotation of P is a subset of the denotation of T. In adapting this to MProver, we may say that **Forall** formulas quantified over expressions of type T refine the type T. As it happens, the programming fragment of MProver as presented here does not contain such features as *seq*, $\sim$-patterns, and polymorphic recursion, which *are* present in P-logic and add significantly to the complexity of its semantics. Excluding these features means that the soundness argument for MProver will be considerably simpler; at the same time, we believe that building MProver's semantics on top of this work shows that MProver *could* be extended with these features without disturbing its semantic foundations.

We also expect that an operational semantics, and a corresponding soundness argument, can be derived from existing work on Operational Type Theory [7], if it is suitably adapted to handle lazy evaluation and infinite data structures; we intend to consider this in future work.

Extension: Termination Types. Haskell equational reasoning proofs very often make assumptions about the finiteness or definedness of an expression. There are two major reasons for this: (1) sometimes infinite or undefined values are a pathological case that we don't actually care about, and excluding them may simplify reasoning (permitting in particular the use of structural induction); and (2) sometimes the properties we want to prove for defined and/or finite values aren't actually true of undefined or infinite values. The design described here does not have an adequate mechanism for handling this.

A possible solution to this problem is to further augment MProver's type system with a system of *termination types*. In this system, termination annotations would refine types by attaching tags restricting types to the finite case, or to the defined case (or to both). A proof quantified over, say, finite lists, could then use structural induction (as opposed to *co*induction). Standard techniques for checking termination and productivity (e.g. structural/guarded recursion) could be integrated into the type checker. This idea bears a close similarity to Howard's work on pointed types [25] and the termination types of Trellys [26].

Acknowledgments. The authors would like to thank the anonymous reviewers of previous versions of this paper, and of this present version, for their most insightful comments. Marko van Eekelen provided many helpful insights and pointers into the literature, especially relating to Sparkle's handling of type classes and (co-)induction.

References

1. The Coq development team: The Coq Proof Assistant Reference Manual. LogiCal Project, Version 8.3 (2010)
2. de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem Proving for Functional Programmers. In: Arts, T., Mohnen, M. (eds.) IFL 2002. LNCS, vol. 2312, pp. 55–71. Springer, Heidelberg (2002)
3. Gibbons, J., Hinze, R.: Just do it: Simple monadic equational reasoning. In: ICFP (September 2011)
4. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries, the Revised Report. Cambridge University Press (2003)
5. Giménez, C.E.: Un calcul de constructions infinies et son application a la verification de systemes communicants, Ph.D. thesis (1996)
6. Yorgey, B.: Typeclassopedia, `http://www.haskell.org/haskellwiki/Typeclassopedia` (accessed May 31, 2012)
7. Stump, A., Deters, M., Petcher, A., Schiller, T., Simpson, T.: Verified Programming in Guru. In: PLPV 2008 (2008)

8. Benton, N., Kennedy, A., Varming, C.: Some Domain Theory and Denotational Semantics in Coq. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 115–130. Springer, Heidelberg (2009)
9. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: ICFP 2008 (2008)
10. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare Type Theory, Polymorphism and Separation. J. Funct. Program. 18(5-6), 865–911 (2008)
11. van Kesteren, R., van Eekelen, M., de Mol, M.: Proof support for general type classes. In: TFP 2004, pp. 1–16 (2004)
12. van Eekelen, M., de Mol, M.: Proof tool support for explicit strictness. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 37–54. Springer, Heidelberg (2006)
13. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: POPL 1989, pp. 60–76 (1989)
14. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)
15. Huffman, B., Matthews, J., White, P.: Axiomatic constructor classes in Isabelle/HOLCF. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 147–162. Springer, Heidelberg (2005)
16. Hallgren, T.: Haskell Tools from the Programatica Project. In: Haskell 2003, pp. 103–106 (2003)
17. Xu, D.N.: Extended static checking for Haskell. In: Haskell 2006, pp. 48–59 (2006)
18. Xu, D.N., Peyton Jones, S., Claessen, K.: Static contract checking for Haskell. In: POPL 2009, pp. 41–52 (2009)
19. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In: Haskell 2008, pp. 37–48 (2008)
20. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: ICFP 2000, pp. 268–279 (2000)
21. Gill, A.: Introducing the Haskell Equational Reasoning Assistant. In: Haskell 2006, pp. 108–109 (2006)
22. Schröder, L., Mossakowski, T.: HasCasl: Integrated higher-order specification and program development. Theor. Comput. Sci. 410, 1217–1260 (2009)
23. Kieburtz, R.B.: P-logic: property verification for Haskell programs (2002)
24. Harrison, W.L., Kieburtz, R.B.: The Logic of Demand in Haskell. J. Funct. Program. 15, 837–891 (2005)
25. Howard, B.T.: Inductive, coinductive, and pointed types. In: ICFP 1996: Proceedings of the First ACM SIGPLAN International Conference on Functional Programming, pp. 102–109. ACM, New York (1996)
26. Casinghino III, C., Eades, H.D., Kimmell, G., Sjoberg, V., Sheard, T., Stump, A., Weirich, S.: The preliminary design of the Trellys core language Talk and discussion session at PLPV 2011 (2011)
27. Norell, U.: Towards a practical programming language based on dependent type theory. Department of Computer Science and Engineering, Chalmers University of Technology (September 2007)

Towards a Framework for Building Formally Verified Supercompilers in Coq

Dimitur Nikolaev Krustev

IGE+XAO Balkan, Bulgaria
dkrustev@ige-xao.com

Abstract. We present an on-going project for the development – in Coq – of a language-agnostic framework for building verified supercompilers. While existing supercompilers are not very big in size, they combine many different program transformations in intricate ways, so checking the correctness of their implementation presents challenges. We propose to define the main supercompilation algorithm in terms of abstract operations, which hide the details of the object language. The verification of the generic supercompiler relies then on properties of these operations, which are easier to establish in isolation, for each specific language. While we still need to try the approach on more supercompilers for specific languages, the framework in its current state appears a promising technique for simplifying the formal verification of supercompilers.

Keywords: program transformation, supercompilation, partial evaluation, deforestation, formal verification, Coq, dependent types.

1 Introduction

Supercompilation is a powerful program-transformation technique [19,18], related to, and generalizing transformations like partial evaluation and deforestation [21]. Supercompilers have many interesting applications: program optimization [1,14]; checking program equivalence [9]; showing productivity of co-recursive definitions [13] (to name just a few). Supercompilation combines some local transformations of program sources, global techniques, such as function call unfolding/folding, and specific heuristic for ensuring termination (on-line history-based termination checks, generalization of recursive call arguments, etc.). The validity of the local transformations usually follows easily from the semantics of the object language, but the correctness and termination of the overall process are more subtle to verify. General techniques for such verification have been developed [16,17], but their manual application is still hard and error-prone. The recent advances (e.g. [11]) in techniques for formal software verification, especially those based on automated proof assistants like Coq [12], suggest that it may be useful to automate this effort in the context of supercompilation. This can be particularly important if the supercompiler is itself used for proving program properties, or is a part of an industrial-strength optimizing compiler. Such verifications have already been carried out (to different extent) for particular supercompiler implementations [10,13].

Performing similar formal proofs from scratch usually takes a lot of effort and is costly. Besides, the quality of supercompilation results often relies crucially on selecting a right combination of ingredients (termination check, folding and generalization strategy). Repeating the full verification for many variations of the supercompiler components is clearly not a scalable approach. As an alternative, we propose an experimental framework, in Coq, whose goal is to verify the correctness of a whole family of supercompilers, even ones processing different programming languages. This framework is intended as a "supercompiler construction kit", where the user can:

– give definitions of a handful of basic supercompilation components for the object language of choice;
– plug these components into the existing generic supercompilation algorithm, to obtain a working supercompiler for the selected language;
– optionally, define the semantics of the object language in Coq, then prove the semantics-preservation conditions for the basic components and obtain automatically a semantics-preservation result for the whole supercompiler;
– optionally, prove the termination condition for one of the basic components and obtain a guarantee that the whole supercompiler is terminating.

This paper describes the current state of the framework:

– a key contribution is the identification of a small number of correctness conditions for a typical set of components of a classically organized supercompiler in Sect. 2 (extended to cover completeness in Sect. 4.3). These are the only conditions that must be verified for each particular object language and component implementation. We conjecture that the sum of these verifications will be simpler than the monolithic verification of the full supercompiler. We illustrate the basic components by defining them for a very small functional language;
– we give a dependently-typed inductive representation of well-formed process graphs in Sect. 3, and a formal semantics of process graphs – relative to the abstract semantics of the object language – in Sect. 3.1;
– a generic, language-agnostic formulation of the supercompilation algorithm in Coq is shown in Sect. 4.1. We prove that this algorithm is terminating under the assumption of an almost-full relation used as "whistle" (dynamic termination check). The proof is facilitated by the recent constructive formalization in Coq of the theory of almost-full relations [20]. We employ a slight modification in the classical supercompilation algorithm, which simplifies greatly our termination proof, compared to Sørensen's technique [17];
– we give a high-level overview of the formal Coq proofs of preservation of semantics for the generic supercompilation algorithm in Sect. 4.2 (soundness) and Sect. 4.3 (completeness).

```
Inductive DriveStepResult (Cfg CH Cnt: Set) : Set :=
  | DSTransient: Cfg → DriveStepResult Cfg CH Cnt
  | DSBranch: CH → list1Set (Cnt × Cfg) → DriveStepResult Cfg CH Cnt
  | DSDecompose: (list Cfg → Cfg) → list Cfg → DriveStepResult Cfg CH Cnt.
Module Type ScpSig.
Parameters Prog Data: Set. Parameter Eval: Prog → relation Data.
Parameters Configuration Contraction ContractionHead: Set.
Parameter driveStep: Configuration →
  option (DriveStepResult Configuration ContractionHead Contraction).
Parameters EvalEnv EvalEnvTransf: Set.
Parameter contrHeadValid: EvalEnv → ContractionHead → bool.
Parameter evalContr: EvalEnv → ContractionHead → Contraction → option EvalEnv.
Parameter foldConfs: Configuration → Configuration → option EvalEnvTransf.
Parameter HistEntry: Set. Parameter conf2histEntry: Configuration → HistEntry.
Parameter whistleRel: relation HistEntry.
Parameter whistleRel_dec: ∀ x y, {whistleRel x y} + {¬whistleRel x y}.
Parameter generalize: Configuration → Configuration →
  (list Configuration → Configuration) × list Configuration.
End ScpSig.
```

Fig. 1. Supercompiler basic building blocks

```
Inductive Val : Set := VNil | VCons (hd tl: Val).
Inductive Exp: Set := V: nat → Exp | F: FN.T → list Exp → Exp
  | G: FN.T → Exp → list Exp → Exp | Nil: Exp | Cons: Exp → Exp → Exp.
Record Prog': Set := MkProg {fdefs: list (FN.T × Exp); gdefs: list (FN.T × (Exp ×
Exp))}.
Definition Data := Val. Definition Prog := Prog'.
... Definition EvalExp (p: Prog) (e: Exp) (v: Val) : Prop :=
  ∃ e1 , RedExp p e e1 ∧ expIsVal e1 = Some v.
Definition Eval prg v1 v2 := match fdefs prg with
  | (_, e)::_ ⇒ EvalExp prg (substF (fun n ⇒ val2exp (valNth n v1)) e) v2
  | _ ⇒ False
  end.
```

Fig. 2. Example: a small functional language

2 Object Language Assumptions

2.1 Basic Supercompiler Building Blocks

The described framework is based on a small list of basic components, which cover the object-language-specific parts of the final supercompiler. Their signatures are collected in a module type[1] (Fig. 1). We proceed to briefly explain each of these components, and in parallel illustrate how they can be defined for a tiny

[1] Some hints about Coq notation: function, algebraic-data-type (`Inductive` ... `Set` := ...), record, and module definitions are similar to those in functional languages like Haskell and ML. Property statements use standard logic syntax. Inductively-defined relations can be introduced by `Inductive` ... `Prop` := ... (using linear syntax).

sample language. We also introduce along the way some supercompiler-specific terminology[2].

The language we consider as an example is a first-order functional one, with call-by-name semantics (Fig. 2). We assume a set of function names *FN.T* with decidable equality *FN.eq_dec*. The language operates on finite binary trees. Instead of case-expressions it has function definitions, which can pattern-match on their first argument. Functions with pattern-matching definitions are dubbed *g-functions* as in [18], while the rest are called *f-functions*. We have different constructors for f- and g-calls in the abstract syntax, as the latter always have at least 1 argument. Another small twist in order to keep things simple: variable references V n are overloaded with 2 meanings: inside function bodies they reference parameters by position, while in free-standing expressions they are treated just as free variables, with an easy way to create fresh names. We have, correspondingly, a generic substitution function substF (f: **nat** $\rightarrow$ **Exp**) (e: **Exp**) : **Exp**, with 2 specific instances: bvSubst (es: **list Exp**) (e: **Exp**) : **Exp** for substituting a list of arguments inside a function body, and fvSubst (env: **list** (**nat** $\times$ **Exp**)) (e: **Exp**) : **Exp** for substituting for free variables inside an open expression. We omit the full formal definition of the language semantics. It can be found in the Coq sources accompanying the paper[3]. We assume the first f-definition in the program is the main one, with the program input being a list of values for the main-function arguments. For example, we can define in our language a program appending its 2 arguments as lists (notice than in the expression for the Cons x y pattern argument positions are shifted by 2):
appPrg := `let` *eNil* := V 0 `in` `let` *eCons* := Cons (V 0) (G "append" (V 1) [V 2]) `in` MkProg [("main", G "append" (V 0) [V 1])] [("append", (*eNil*, *eCons*))]

and it will hold that:
`let` *v1* := VCons VNil (VCons VNil VNil) `in` `let` *v2* := VCons VNil VNil `in`
`Eval` *appPrg* (VCons *v1* (VCons *v2* VNil))
(VCons VNil (VCons VNil (VCons VNil VNil)))

The first key ingredient of supercompilation is *driving* – a form of symbolic execution of programs with free variables. The driving process typically operates not directly on programs, but on "configurations" – representations of the current state of the driven program. In our example (Fig. 3) each configuration consists of a copy of the original program, plus the expression currently being processed, decomposed into a *reduction context* + *redex*. Because in our language reduction is forced only at the top or at the first argument of a g-call, we represent the (key part of the) redex with a type **Exp'**, which is just a copy of **Exp** lacking a top-level G constructor, and the reduction context stores (in reverse order) all top-level g-calls enclosing the topmost leftmost subexpression of type **Exp'**. We have a function toRedex decomposing an expression into a context + redex, and an inverse function expOfCtx.

In addition, when dealing with branching constructs, which cannot be decided statically during driving, we pursue each alternative in parallel. The undecided

[2] More complete introductions to supercompilation can be found, for example, in [18,7]
[3] `https://sites.google.com/site/dkrustev/Home/publications`

```
Record Conf: Set := MkConf {prog: Prog; redCtx: list (FN.T × list Exp); subexp: Exp'}.
Definition Configuration := Conf.
Definition ContractionHead := nat. Definition Contraction := option (nat × nat).
Definition expOfConf cfg := expOfCtx (redCtx cfg) (subexp cfg).
Definition driveStep' (prg: Prog) (ctx: list (FN.T × list Exp))
  (e: Exp') : option (DriveStepResult Conf ContractionHead Contraction) :=
  match e, ctx with
  | V' x, nil ⇒ None
  | V' x, _::_ ⇒ let nilCtx := map (fun p ⇒ (fst p,
                map (fvSubst ((x, Nil)::nil)) (snd p))) ctx in
    let nilCfg := MkConf prg nilCtx Nil' in
    let cnt := expSuccMaxV (expOfCtx ctx e) in let x1 := cnt in
    let x2 := (S cnt) in let consExp := Cons' (V x1) (V x2) in
    let consCtx := map (fun p ⇒ (fst p,
        map (fvSubst ((x, exp'2exp consExp)::nil)) (snd p))) ctx in
    let consCfg := MkConf prg consCtx consExp in
    Some (DSBranch x ((Some (x1, x2), consCfg)::nil, (None, nilCfg)))
  | Nil', nil ⇒ None
  | Nil', (f, es)::ctx1 ⇒ optionBind (lookup FN.eq_dec f (gdefs prg)) (fun gdef ⇒
    let c_e := toRedex (bvSubst es (fst gdef)) in
    Some (DSTransient (MkConf prg (fst c_e ++ ctx1) (snd c_e))))
  | Cons' e1 e2, nil ⇒ let f cfgs := match cfgs with
      | cfg1::cfg2::_ ⇒ MkConf prg nil (Cons' (expOfConf cfg1) (expOfConf cfg2))
      | _ ⇒ MkConf prg nil e end in
    let c_e1 := toRedex e1 in let c_e2 := toRedex e2 in
    Some (DSDecompose f (MkConf prg (fst c_e1) (snd c_e1)
        :: MkConf prg (fst c_e2) (snd c_e2) :: nil))
  | Cons' e1 e2, (f, es)::ctx1 ⇒
    optionBind (lookup FN.eq_dec f (gdefs prg)) (fun gdef ⇒
    let c_e := toRedex (bvSubst (e1::e2::es) (snd gdef)) in
    Some (DSTransient (MkConf prg (fst c_e ++ ctx1) (snd c_e))))
  | F' f es, _ ⇒ optionBind (lookup FN.eq_dec f (fdefs prg)) (fun fdef ⇒
    let c_e := toRedex (bvSubst es fdef) in
    Some (DSTransient (MkConf prg (fst c_e ++ ctx) (snd c_e))))
  end.
Definition driveStep conf := driveStep' (prog conf) (redCtx conf) (subexp conf).
```

Fig. 3. Example language: configurations, driving

conditions in such cases are represented by "contractions". It is convenient to factor out the common part of all contractions of a given branching construct – represented by ContractionHead in Fig 1. In the case of the sample language, the only kind of driving-time-undecidable branching construct is a g-call of the form (G f (V n) ...). So contractions can be of 2 kinds: V n = Nil and V n = Cons (V m) (V k), where m and k are fresh, and the common part is (V n =). This explains the definitions for ContractionHead and Contraction in Fig 3.

```
Definition EvalEnv := list (nat × Data).
Definition EvalEnvTransf := list (nat × nat).
Definition foldConfs cfg1 cfg2 := if Prog_eq_dec (prog cfg1) (prog cfg2)
 then renamingExists (expOfConf cfg1) (expOfConf cfg2) else None.
Definition HistEntry := Exp. Definition conf2histEntry := expOfConf.
Parameter maxExpCode: nat.
Definition whistleRel (e1 e2: Exp) : Prop :=
  let fin1 := nat2Finite maxExpCode (nat_of_N (expCode e1)) in
  let fin2 := nat2Finite maxExpCode (nat_of_N (expCode e2)) in eq_fin fin1 fin2.
Definition whistleRel_dec: ∀ e1 e2, {whistleRel e1 e2} + {¬whistleRel e1 e2}.
... Defined.
Definition generalize (cfg1 cfg2: Configuration)
  : (list Configuration → Configuration) × list Configuration := (fun _ ⇒ cfg2, nil).
```

Fig. 4. Example language: remaining supercompiler components

In traditional formulations of supercompilation, driving produces a potentially infinite *process tree*, which encodes each possible path that the execution of the input program may take. In our framework we assume, that driving is performed in steps, where each driving step takes a configuration as input and produces one of four kinds of results (Fig. 1, Fig. 3):

– None: driving cannot proceed further at this configuration (for our example, it can happen if we have Nil or a free variable in an empty context, and also at a call of an undefined function);
– Some (DSTransient *c*): driving can perform some deterministic static computation and produce a new configuration (in the case of the example: any f-call, or a g-call, where the shape of the first argument (Nil/Cons) is known);
– Some (DSBranch *ch alts*): driving is stuck at a branching construct with a statically-undecidable condition; in this case it produces a list of *contraction* + *configuration* pairs for each of the branches (as discussed in the previous paragraph, in our example this can only happen when driving reaches a g-call with a free variable as the first argument). In this case – as seen in the example implementation – information about the test outcome is propagated inside each branch. This feature is crucial for achieving more powerful optimizations (compared to deforestation, for example), but orthogonal to our treatment of correctness.
– Some (DSDecompose *f cs*): driving cannot proceed directly with the current configuration, but it can decompose it to a number of sub-configurations with which to continue (for the sample language – when we reach a Cons in an empty reduction context).

This formalization of driving is close to many existing formulations [18,7]. As a technical detail, we simplify the treatment of the branching case by assuming that there always is at least one branch, and that the last branch is taken by default if the preceding ones fail.

One way to produce a finite *process graph* from the infinite *process tree* is through *folding* a leaf of the process tree into a sufficiently similar node on the path to the root. For a functional language, this corresponds to introducing a new recursive function definition. The formalization of folding (Fig. 1) relies on the fact, that we introduce later an evaluation relation *EvalConf* directly for configurations (Fig. 5). As configurations typically contain free variables, the input for configuration evaluation takes the form of an evaluation environment EvalEnv meant to assign values to these free variables. Thus the concrete definition of EvalEnv for our example comes as no surprise (Fig. 4). We further assume an abstract set of transformations EvalEnvTransf, which can be applied to evaluation environments (Fig. 1, Fig. 5). In the case of the sample language, these transformations are just renamings of free variables. Thus, folding is possible given a renaming exists, transforming the first configuration into the second (Fig. 4).

In general, opportunities for folding are not guaranteed to always arise. Supercompilers typically ensure totality by performing on-line termination checks during driving. These checks – traditionally called *whistles* – consider the complete history of already driven program states along the path from the current node to the root of the process tree. Well-quasi orders are usually used as whistles. We assume instead that our whistle is an almost-full relation, following a recent paper by Vytiniotis et al., which promotes almost-full relations as a constructively-defined alternative to well-quasi orders, and provides a Coq library for working with those [20]. To keep the framework flexible, we assume that the whistle operates on some history entries, which can be computed from configurations. For our specific example, the history entry is just the expression corresponding to the given configuration. For the whistle itself, we take some Goedel numbering of expressions (expCode), truncate it up to a threshold (supplied as a parameter), and compare the 2 truncated codes for equality. The truncated codes are always in {0, ..., *maxExpCode*-1}, which in the library from [20] is represented by a type **Finite** *maxExpCode*. We can then directly establish, that our whistle is an almost-full relation, by using a library lemma about equality at type **Finite** n. (Of course, in this simple case we could directly appeal to the pigeonhole principle, but the Coq library from [20] supplies some combinators for building more complicated and more interesting whistles.)

Finally, a supercompiler needs to decide what to do if a configuration is not foldable to a previous one, but the whistle has blown that further driving risks non-termination. The simplest strategy is to leave the configuration as it is, and we indeed use this as a fall-back approach in the supercompiler definition below. More interesting results can be achieved if the supercompiler is able to perform a *generalization*, which produces a set of new configurations on which driving can continue. Perhaps unsurprisingly, our formalization of generalization is very similar to the DSDecompose case of driving. We do not pursue interesting definitions of generalization for our sample language, though, showing that we can still obtain a working supercompiler even with a trivial definition like the one in Fig. 4.

```
Module Type ScpPropsSig.
Axiom decomposeConf_correct: ∀ conf f confs,
  driveStep conf = Some (DSDecompose f confs) → f confs = conf.
Parameter applyEET: EvalEnvTransf → EvalEnv → EvalEnv.
Parameter EvalConf: EvalEnv → Configuration → Data → Prop.
Axiom EvalConf_deterministic: ∀ env conf d1,
  EvalConf env conf d1 → ∀ d2, EvalConf env conf d2 → d1 = d2.
Parameter initConf: Prog → Configuration.
Parameter initEvalEnv: Prog → Data → EvalEnv.
Axiom EvalConf_correct: ∀ prog d1 d2,
  EvalConf (initEvalEnv prog d1) (initConf prog) d2 ↔ Eval prog d1 d2.
Axiom driveStep_DSTransient_correct: ∀ conf conf1,
  driveStep conf = Some (DSTransient conf1) → ∀ env d,
  EvalConf env conf d ↔ EvalConf env conf1 d.
Axiom driveStep_DSBranch_correct: ∀ conf ch alts,
  driveStep conf = Some (DSBranch ch alts) → ∀ env d,
  EvalConf env conf d ↔ contrHeadValid env ch = true ∧
  let env_conf := findAlt env ch alts in EvalConf (fst env_conf) (snd env_conf) d.
Parameter data2conf: Data → Configuration.
Axiom data2confEval: ∀ env d, EvalConf env (data2conf d) d.
Axiom driveStep_DSDecompose_correct: ∀ conf f confs,
  driveStep conf = Some (DSDecompose f confs) → ∀ env d,
  EvalConf env conf d ↔ ∃ ds, mapRelXsYs (EvalConf env) confs ds
    ∧ EvalConf env (f (map data2conf ds)) d.
Axiom whistleRel_almostFull: almost_full whistleRel.
Axiom foldConfs_correct: ∀ conf1 conf2 eet, foldConfs conf1 conf2 = Some eet →
  ∀ env d, EvalConf env conf2 d ↔ EvalConf (applyEET eet env) conf1 d.
Axiom generalize_correct: ∀ conf1 conf2 f confs, generalize conf1 conf2
  = (f, confs) → ∀ env d, EvalConf env conf2 d ↔ ∃ ds, mapRelXsYs
  (EvalConf env) confs ds ∧ EvalConf env (f (map data2conf ds)) d.
End ScpPropsSig.
```

Fig. 5. Supercompiler component properties

There are some auxiliary components in Fig. 1 – contrHeadValid and evalContr – used only internally in defining the correctness conditions coming next, so we omit their definitions.

2.2 Correctness Properties of Supercompiler Components

We have described all the external components our framework requires in order to produce a working supercompiler. In order to also prove it correct, we need the components to respect some properties, listed in Fig. 5. We have already mentioned the introduction of an evaluation relation for configurations – *EvalConf* – in terms of which most properties are formulated. Of course, evaluation of configurations must be compatible with the original semantics of the language: if we produce an initial configuration and evaluation environment from a program and some input data, the evaluation of the configuration with this environment must give the same result as the evaluation of the program (*EvalConf_correct*). We further require 3 separate conditions for 3 of the possible driveStep results:

– *driveStep_DSTransient_correct*: evaluating the initial and the new configuration must give the same result;
– *driveStep_DSBranch_correct*: evaluating the initial configuration must give the same result as the evaluation of the first alternative, whose contraction matches (findAlt finds this alternative using evalContr). Notice that the matching contraction may produce a new evaluation environment to be used for the corresponding selected configuration;
– *driveStep_DSDecompose_correct*: here we assume that the result of evaluating each sub-configuration can be converted itself to a configuration (with *data2conf*). If we then plug these new sub-configurations in place of the original ones, we must obtain the same evaluation result. (We use mapRelXsYs, which is a binary-relation analog of the familiar map.)

Of the remaining correctness conditions, we have already discussed the one concerning the whistle: *whistleRel_almostFull*. This is the only condition needed to prove the supercompiler terminating. Conversely, it has nothing to do with preservation of semantics. When we fold to a configuration above in the process tree, we require (*foldConfs_correct*) that evaluating the upper configuration with a transformed environment (akin to making a function call) is equivalent to the evaluation of the lower configuration with the current environment. Finally the condition for generalization – *generalize_correct* – is completely analogous to the condition for DSDecompose. Proofs of all these properties – for the components defined for our example language (Fig. 3, 4) – can be found in the accompanying Coq sources.

3 Process Graphs

From now on we assume present an arbitrary definition of a module with supercompiler components (Fig. 1), together with proofs of their properties (Fig. 5). As mentioned, driving produces potentially infinite process trees, which are converted – with the help of folding and generalization – into finite process graphs. The process graph resulting from supercompilation can be seen as an alternative representation of the transformed input program. In practical supercompilers, a post-processing phase converts the process graph into a new program in the corresponding language. As this last phase is too language-specific, we do not cover it, and consider instead that the final result of supercompilation is the process graph. These graphs have – besides the tree edges coming from the process tree – also backward edges corresponding to folded configurations. We can use this almost-tree nature of process graphs to make a simple inductive-type representation. To rule out "dangling" backward edges, however, we use a dependently-typed encoding, similar to the now standard (de-Bruijn-index based) approach for encoding only well-formed terms of languages with bindings (Fig. 6). A type **PGraph** n in the inductive family **PGraph** corresponds thus to a subgraph at

```
Inductive PGraph : nat → Set :=
  | PGLeaf: ∀ {n}, Configuration → PGraph n
  | PGTransient: ∀ {n}, Configuration → PGraph (S n) → PGraph n
  | PGBranch: ∀ {n}, ContractionHead →
      list1Set (Contraction × Configuration × PGraph (S n)) → PGraph n
  | PGDecompDrv: ∀ {n}, (list Configuration → Configuration) →
      list (Configuration × PGraph (S n)) → PGraph n
  | PGDecompGen: ∀ {n}, (list Configuration → Configuration) →
      list (Configuration × PGraph n) → PGraph n
  | PGFold: ∀ {n}, Fin (S n) → EvalEnvTransf → PGraph (S n).
```

Fig. 6. Process graphs

depth n, and complete process graphs are represented by the type **PGraph** 0. Notice that we treat slightly differently decomposition nodes arising from driving as compared to ones arising from generalization. As we shall see, the latter are not recorded in the history of visited configurations during supercompilation, and thus are not considered as candidates for folding. For this reason they do not count as separate levels in the graph.

3.1 Semantics of Process Graphs

To establish some semantics-preservation results, we need first to define the semantics of process graphs. To give a semantics of folding nodes, we must keep track of all nodes on the path from the root to the fold node itself. This list contains graphs of strictly decreasing depth, represented as **IndexedVect PGraph** n. The semantics of process graphs can be defined inductively (Fig. 7). Most cases are straightforward, following our assumptions about the semantics of driving step results. In the case of PGFold i *fl*, we move back i steps up the graph (using a function ivNthWithTail, which returns both the i-th element and the corresponding tail of an **IndexedVect**), assuming that the evaluation of the upper node – in an environment modified by the fold label (*fl*) – gives the same result as the evaluation of the fold node. The evaluation of PGBranch simply continues with the selected sub-graph. For PGDecompDrv/PGDecompGen we first evaluate each sub-graph. We can then reconstruct – using *data2conf* – a configuration corresponding to the decomposition node itself, and use its evaluation result to define the result of evaluating the decomposition node. The PGDecompGen case differs from PGDecompDrv in that it does not push the current node onto the stack of ancestor nodes before proceeding to evaluate sub-nodes.

4 Supercompilation

4.1 Provably-Terminating Supercompiler Definition

As already mentioned, we use the recent Coq formalization [20] of almost-full relations to simplify the termination proof of our abstract supercompilation algorithm. This Coq library provides some predefined almost-full relations plus

```
Inductive PGEval: EvalEnv → ∀ n, IndexedVect PGraph n
  → PGraph n → Data → Prop :=
  | PGEvalLeaf: ∀ env n pgs conf d,
      EvalConf env conf d → PGEval env n pgs (PGLeaf conf) d
  | PGEvalTransient: ∀ env n pgs conf pg d,
      PGEval env _ (IVCons (PGTransient conf pg) pgs) pg d
      → PGEval env n pgs (PGTransient conf pg) d
  | PGEvalFold: ∀ env n (pgs: IndexedVect PGraph (S n)) (i: Fin (S n)) fl d,
      PGEval (applyEET fl env) _ (snd (ivNthWithTail pgs i))
        (fst (ivNthWithTail pgs i)) d → PGEval env (S n) pgs (PGFold i fl) d
  | PGEvalBranch: ∀ env n pgs ch alts env_pg d, contrHeadValid env ch = true →
      env_pg = findAlt env ch (list1map (fun cct ⇒ (fst (fst cct), snd cct)) alts)
      → PGEval (fst env_pg) _ (IVCons (PGBranch ch alts) pgs) (snd env_pg) d
      → PGEval env n pgs (PGBranch ch alts) d
  | PGEvalDecompDrv: ∀ env n pgs f cgs d ds,
      mapRelXsYs (PGEval env _ (IVCons (PGDecompDrv f cgs) pgs))
        (map (@snd _ _) cgs) ds → EvalConf env (f (map data2conf ds)) d
      → PGEval env n pgs (PGDecompDrv f cgs) d
  | PGEvalDecompGen: ∀ env n pgs f cgs d ds,
      mapRelXsYs (PGEval env _ pgs) (map (@snd _ _) cgs) ds
      → EvalConf env (f (map data2conf ds)) d
      → PGEval env n pgs (PGDecompGen f cgs) d.
```

Fig. 7. Process graph semantics

combinators for building new out of existing ones. Perhaps more importantly, it provides a way to build a well-founded relation out of an almost-full one. We start by defining a transition relation scpTransRel: relation ScpArg (where ScpArg := (**list** (Configuration × HistEntry) × Configuration)%*type*), which relates the recursive-call arguments to the arguments of the original call in the supercompiler definition below; we can then prove that this transition relation is well-founded, based on our assumption that the whistle is almost-full. With a suitable well-founded relation in place, we can proceed with the definition of the main supercompilation algorithm (Fig. 8). It uses 2 (omitted) auxiliary definitions: optionMatchWithEq matches **option** values with propagation of the match result inside the branches (needed in the proofs that recursive calls to supercompile respect the transition relation mentioned above); scpTryFold (*hist*: **list** (Configuration × HistEntry)) (*cfg*: Configuration) : **PGraph** (length *hist*) → **PGraph** (length *hist*) checks for possibilities to perform folding. The main supercompilation function is then defined using the build-in Coq facilities for recursion by a well-founded relation (the `Fix` combinator). The main definition is slightly less readable than its Haskell or ML equivalent (which we can obtain through extraction), because of the need to carry around extra hypothesis for the termination proof [4]. We can note only one important variation with respect to

[4] The termination proof is elided here, only the computationally-relevant part of the definition is shown.

```
Definition whistle he1 he2 := if whistleRel_dec he1 he2 then true else false.
Definition supercompile (arg: ScpArg) : PGraph (length (fst arg)).
  revert arg. apply (Fix ( scpTransRel_wf) (fun arg ⇒ PGraph (length (fst arg)))).
  refine (fun arg supercompile ⇒
  let hist := fst arg in let cfg := snd arg in
  let drive cfg1 (Heq: None =
        find (fun h ⇒ whistle (snd h) (conf2histEntry cfg1)) hist) :=
    let newHist := (cfg1, conf2histEntry cfg1) :: hist in
    match driveStep cfg1 with
    | None ⇒ PGLeaf cfg1
    | Some (DSTransient cfg2) ⇒ PGTransient cfg1 (supercompile (newHist, cfg2) _)
    | Some (DSBranch ch ccs) ⇒ let f cc := let pg1: PGraph (S (length hist)) :=
        supercompile (newHist, snd cc) _ in (fst cc, snd cc, pg1) in
      PGBranch ch (list1map f ccs)
    | Some (DSDecompose f cfgs) ⇒ let g cfg2 := let pg1: PGraph (S (length hist))
        := supercompile (newHist, cfg2) _ in (cfg2, pg1) in
      PGDecompDrv f (map g cfgs)
    end in
  match driveStep cfg with
  | None ⇒ PGLeaf cfg
  | Some _ ⇒ let pg := optionMatchWithEq
    (find (fun h ⇒ whistle (snd h) (conf2histEntry cfg)) hist)
    (fun p Heq0 ⇒ let oldCfg := fst p in
        let f_cs := generalize oldCfg cfg in
        let f := fst f_cs in let cs := snd f_cs in
        let scpSubCfg cfg1 := match driveStep cfg1 with
          | None ⇒ PGLeaf cfg1
          | Some _ ⇒ let pg: PGraph (length hist) := optionMatchWithEq
              (find (fun h ⇒ whistle (snd h) (conf2histEntry cfg1)) hist)
              (fun p Heq1 ⇒ PGLeaf cfg1)
              (fun Heq1 ⇒ drive cfg1 Heq1) in
            scpTryFold hist cfg1 pg
          end in
        let cgs: list (Configuration × PGraph ((length hist))) :=
          map (fun cfg1 ⇒ (cfg1, scpSubCfg cfg1)) cs in PGDecompGen f cgs)
    (fun Heq0 ⇒ drive cfg Heq0)
    in scpTryFold hist cfg pg
  end).
... Defined.
```

Fig. 8. Generic supercompiler definition

classical formulations of supercompilation [18,17,7], concerning the treatment of generalization. When we need to generalize, instead of immediately proceeding to supercompile the new configurations generated by `generalize`, we check for each of them if we can perform folding, and if a similar configuration is not already in

```
Variable idEET: EvalEnvTransf.
Hypothesis idEET_eq: ∀ env, applyEET idEET env = env.
Hypothesis driveStep_def: ∀ conf, driveStep conf = Some (DSTransient conf).
Hypothesis foldConfs_eq: ∀ conf, foldConfs conf conf = Some idEET.
Lemma scpProg_eq: ∀ c, supercompile (nil, c) = PGTransient c (PGFold Fz idEET).
Lemma scpProgLoops: ∀ c env d, ¬PGEval env _ IVNil (supercompile (nil, c)) d.
```

Fig. 9. Example of a degenerate driving definition

the history. If both tests fail, we do not drive this particular sub-configuration any further, but keep it as a leaf in the process graph. This modification greatly simplifies our termination proof (compared to Sørensen [17] for example), and avoids the need to impose any termination-related conditions on `generalize`.

4.2 Soundness of Supercompilation

Another important property – besides termination – is soundness of supercompilation [6]: whenever the evaluation of a process graph (obtained through supercompilation) produces some result, the evaluation of the input configuration gives the same result. We can prove soundness by induction on the derivation of **PGEval**, and by using the properties from Fig. 5. The technical details – including choosing a strong-enough induction hypothesis – can be found in the Coq sources.

```
Theorem scp_sound: ∀ conf env d, PGEval env _ IVNil
  (supercompile (nil, conf)) d → EvalConf env conf d.
```

4.3 Completeness of Supercompilation

The other direction of semantics preservation is completeness: whenever the input program produces some result, the evaluation of the supercompiled process graph should terminate with the same result. Unfortunately completeness turns out harder to establish. Our current assumptions about supercompiler ingredients do not rule out some degenerate definitions of driving, which do not preserve termination in the supercompilation result (not to be confused with termination of the supercompiler itself). The example in Fig. 9 shows an instance of the problem (with supercompilation always producing an infinite loop).

Obviously we need some stronger assumptions about supercompiler components in order to establish completeness. One possibility is to consider explicitly the evaluation semantics induced by our generic supercompiler definition – in the form of an inductive relation **ScpEval**. If we make the extra assumption, that **ScpEval** is equivalent to *EvalConf*, we can prove a generic completeness result by induction on the derivation of **ScpEval**.

```
Theorem scp_complete: ∀ conf env d, EvalConf env conf d →
  PGEval env _ IVNil (supercompile (nil, conf)) d.
```

Details about this proof can be found in the Coq sources. Further studies are needed to see if this new proof obligation – needed only for proving completeness

– is easy enough to establish in each instance, or it can be replaced by a weaker condition. An evaluation relation for infinite process trees, coupled with an assumption of equivalence to *EvalConf*, appears a good candidate for this purpose.

5 Related Work

Supercompilation has been developed by Turchin and his students, starting from the early 1970s, originally only in the context of the language Refal [19]. Since the 1990s a number of other supercompilers have been developed, transferring the technique to many different (mostly functional) languages. A non-exhaustive(!) list includes [2,3,14,5,9,1,15,13]. Supercompilation is closely related to other program-transformation and optimization techniques, such as partial evaluation and deforestation [21].

A general approach for establishing semantics preservation in supercompilers and similar transformers has been proposed by Sands [16]. While it works for a variety of transformations, it makes strong assumptions about the object language (a higher-order functional language with CBN semantics). In contrast, we make fewer assumptions about the object language, but only treat a specific kind of program transformation. Another general technique, often cited by supercompiler authors, concerns termination of the transformation process [17]. It establishes sufficient conditions for transformer termination in a more general form than the one considered here. This approach has not been formalized, however, and its application in an automated verification system such as Coq would likely require additional work.

In contrast to the number of supercompiler implementations for specific languages, there are only a few proposed approaches for general treatment of supercompilation and similar program transformations [4,6,8]. Jones presents an abstract formulation of driving [4], with only a small number of assumptions about the object language – work similar in spirit to our current approach. Still, some of the assumptions in Jones' paper seem geared towards simple imperative or first-order tail-recursive functional languages. Also, termination and generalization are not treated there.

The work of Klimov [6] covers the complete supercompilation process, and proves a number of interesting high-level properties. To achieve these results, Klimov assumes a specific object language (first-order functional) and data domain. It would be interesting to generalize this approach by abstracting from the details of the object language.

The recent work on the MRSC toolkit [8] is probably closest in spirit to ours. It also considers a language-agnostic formulation of supercompilation based on a small set of basic operations. The focus of the MRSC toolkit is different, however, with the main goal being a generalization of the traditional process of supercompilation, which covers new techniques like non-deterministic and multi-result supercompilation. On the other hand, the MRSC toolkit does not currently cover the process of formally verifying the correctness of the different supercompilers that can be built with it.

6 Conclusions and Future Work

We have presented in some detail – including the key fragments of the Coq source – the current state of a project to create a generic framework for building supercompilers for different programming languages. An important achievement so far has been the identification of a small set of independent properties, which must be satisfied by the basic components of each supercompiler in order to ensure and formally verify the soundness of supercompiled programs and the termination of the supercompiler itself. The underlying hypothesis is, that the independent proofs of these smaller properties will be much simpler than a monolithic proof for the whole supercompiler. The proof of completeness of supercompilation requires, however, some stronger assumptions about the supercompiler components. We plan to study in the future if the conditions required for proving completeness can be simplified and unified with those for soundness.

Another interesting point is, that we have managed to formulate the generic supercompiler algorithm as a total function in Coq. This definition is facilitated by the recent Coq formalization of almost-full relations [20]. Our approach can be adapted to other algorithms, whose termination relies on similar invariants (extending some computation history in each recursive call, while ensuring that this history cannot grow unlimitedly).

Applying the framework to build more supercompilers – for sufficiently different languages – will likely suggest possibilities to refine the framework and further simplify its use. Already the first supercompiler built with this framework suggests an interesting improvement to study: move from call-by-value to call-by-name semantics for decomposition nodes.

While the proposed framework is not tied to any specific programming language, it only covers a specific organization of the supercompilation process. We would not be able to apply it directly on some recent sophisticated modifications of the supercompilation method, such as distillation [3] or multi-result supercompilation [8]. Still the basic idea remains applicable: abstract the details of the specific language into a set of basic operations, equipped with a set of specific correctness properties. Consequently, we envision using modified versions of our framework to cover similar extensions of the basic supercompilation algorithm.

Acknowledgments. The author would like to thank the three anonymous referees for the helpful suggestions on improving the presentation of this article.

References

1. Bolingbroke, M., Peyton Jones, S.: Supercompilation by evaluation. In: Proceedings of the Third ACM Haskell Symposium on Haskell, pp. 135–146. ACM (2010)
2. Glück, R., Klimov, A.V.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Cousot, P., Filé, G., Falaschi, M., Rauzy, A. (eds.) WSA 1993. LNCS, vol. 724, pp. 112–123. Springer, Heidelberg (1993)
3. Hamilton, G.: Distillation: extracting the essence of programs. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 61–70. ACM (2007)

4. Jones, N.D.: The essence of program transformation by partial evaluation and driving. In: Bjorner, D., Broy, M., Zamulin, A.V. (eds.) PSI 1999. LNCS, vol. 1755, pp. 62–79. Springer, Heidelberg (2000)
5. Jonsson, P.A., Nordlander, J.: Positive supercompilation for a higher order call-by-value language. SIGPLAN Not. 44(1), 277–288 (2009)
6. Klimov, A.V.: A program specialization relation based on supercompilation and its properties. In: Turchin, V. (ed.) International Workshop on Metacomputation in Russia, META 2008 (2008)
7. Klyuchnikov, I.: The ideas and methods of supercompilation. Practice of Functional Programming (7) (2011) (in Russian)
8. Klyuchnikov, I.G., Romanenko, S.A.: MRSC: a toolkit for building multi-result supercompilers. preprint 77, Keldysh Institute (2011)
9. Klyuchnikov, I., Romanenko, S.: Proving the equivalence of higher-order terms by means of supercompilation. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 193–205. Springer, Heidelberg (2010)
10. Krustev, D.: A simple supercompiler formally verified in Coq. In: Nemytykh, A.P. (ed.) Proceedings of the Second International Workshop on Metacomputation in Russia, META 2010, pp. 102–127 (2010)
11. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
12. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2011), `http://coq.inria.fr`, version 8.3pl3
13. Mendel-Gleason, G.: Types and verification for infinite state systems. PhD thesis, Dublin City University, Dublin, Ireland (2011)
14. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 147–164. Springer, Heidelberg (2008)
15. Reich, J.S., Naylor, M., Runciman, C.: Supercompilation and the Reduceron. In: Nemytykh, A.P. (ed.) Proceedings of the Second International Workshop on Metacomputation in Russia, META 2010, pp. 159–172 (2010)
16. Sands, D.: Proving the correctness of recursion-based automatic program transformations. Theor. Comput. Sci. 167(1-2), 193–233 (1996)
17. Sørensen, M.H.: Convergence of program transformers in the metric space of trees. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 315–337. Springer, Heidelberg (1998)
18. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Hatcliff, J., Thiemann, P. (eds.) DIKU 1998. LNCS, vol. 1706, pp. 246–270. Springer, Heidelberg (1999)
19. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (1986)
20. Vytiniotis, D., Coquand, T., Wahlstedt, D.: Stop when you are almost-full: Adventures in constructive termination. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 250–265. Springer, Heidelberg (2012)
21. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. 73(2), 231–248 (1990)

Matching Problem for Regular Expressions with Variables

Vladimir Komendantsky

Parallel Scientific, Inc.
vladimir.komendantsky@parsci.com

Abstract. The notion of a *backreference* in practical regular expressions is formalised giving rise to a novel notion of a *regular expression with variables.* So far the state-of-the-art in formal languages theory of practical regular expressions with backreferences was represented by an operational matching semantics of trees of valid matches. Since explicit tree data structures are required only in procedural languages where functional list combinators are not available, this provides an opportunity for functional programming to step in with a computationally and proof-theoretically more tangible definition of backreferences and their matching semantics. An operational notion of (NP-complete) exhaustive pattern matching is provided that relies on a fundamental construction of partial derivatives of regular expressions. Matching is proved sound and complete.

1 Introduction

Regular expressions [22,13] are a formalism ideally suited to specification and implementation with formal methods. They are essential for text processing and form the basis of most markup schema languages. Regular expressions are useful in the production of syntax highlighting systems, data validation, speech processing, optical character recognition, and in many other situations when we attempt to recognise patterns in data.

Extended versions of regular expressions are used in search engines such as Google Code Search. In fact, there is a difference between what is understood by the term *regular expression* in programming and in theoretical computer science. Different software based on regular expressions has in each case its own "RegEx flavour": ECMAScript, Perl-style, GNU RegEx, Microsoft Word, POSIX Basic/Extended RegEx (with extensions), Vim, and many others.

In contrast, theoretical computer science uses a single formal definition for a regular expression which defines it as consisting of constants and operators that denote sets of strings and operations over these sets respectively. For example, assuming a and b are symbols, $\underline{a}+\underline{b}^*$ denotes the set $\{\epsilon, a, b, bb, bbb, \ldots\}$, where ϵ denotes the empty string; and $(\underline{a}+\underline{b})^*$ denotes the set of all strings composed of a and b (including the empty string ϵ): $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. The formal definition is purposely minimalist in that it avoids redundant operators that can be expressed through application of existing ones. Regular expressions in the precise sense express the class of *regular languages*, which is exactly the class of languages accepted by *finite state automata*. This definition has the maximum degree of independence from any implementation.

In functional programming, the usual procedure is to enumerate words of the languages denoted by regular expressions using finite automata [18]. Less common but more efficient are partial derivatives of regular expressions [19,3]. They are purely functional [12].

Patterns. A *pattern* [7] is a description of a word at a meta-scale: instead of considering the word as a sequence of individual symbols, one looks at the word as a sequence of certain blocks. More formally, a pattern p is a word that contains special symbols, called *pattern variables*; p is a pattern of a word w if w is obtained from p by uniformly replacing the variables with words (which may be empty depending on the convention). The *pattern language* denoted by a pattern p is the set of all words that match p. For example, the pattern xyx denotes the set of all words in a language each of which has a prefix and a suffix that are the same and have a word in between.

Many features found in modern pattern matching libraries, and in languages such as Perl or Python, provide expressive power that far exceeds the capacities of regular languages and requires pattern languages. In our opinion, the most interesting of such extra capacities is *backreferencing* – the ability to group subexpressions and recall the value they match later in the same expression. Such a pattern can match strings of repeated words like “papa”, called squares in formal language theory. The obvious Perl pattern for such strings is `(.*)\1`. However, the language of squares is not regular, nor is it context-free. Regular expression matching with an unbounded number of backreferences, as supported by numerous modern tools, is NP-complete [1], and therefore requires practical algorithms that maximally approximate non-deterministic complexity bounds.

Backreferences may be seen acting as variables in patterns. The language denoted by a pattern is obtained by substituting variables with arbitrary terminal strings in the case on erasing patterns and non-empty terminal strings in the case of non-erasing patterns.

Pattern languages are often accepted as the theoretical meaning of extended, also called *practical*, regular expressions [6,20]. However, patterns are defined with no recursion or choice and therefore require pre-processing of regular expressions in order to form a set of patterns from an expression with iteration or non-deterministic choice. For that reason, in this particular paper, I do not choose patterns as the meaning of backreferences.

Matching problem. The matching problem for regular expressions is the problem to decide, for a given word over a finite alphabet of symbols and a given regular expression, whether or not the word belongs to the language denoted by the regular expression. There are quite efficient polynomial algorithms for deciding this basic problem [19,3,8,12]. These algorithms are based on a simple fact that the non-deterministic finite automaton recognising the language of the regular expression is guaranteed to terminate on finite input words.

Alternative existing definitions of regular expressions with backreferences, for example, by means of match-trees [5] or ordered trees [6], are better than patterns in terms of correspondence to the implementation of practical regular expressions – since they refer to the operational matching semantics – but, on the other hand, the tree semantics is quite specific to their matching problem and, moreover, to the paradigm of intended implementation languages.

The solution to these odds that I propose in this paper is to adopt an approach of mechanised meta-theory, namely, to view a matching problem as existence of a mapping from variables to words in a sequent

$$x_0 : e_0, \ldots, x_{k-1} : e_{k-1} \vdash w : e \tag{1}$$

where w is a word to be matched, e is a regular expression whose variables occur in the list of context variables $x_0, \ldots, x_{k-1}$ such that each variable x_i is bound to match only those words that are matched by e_i, and such that the variables of e_i occur in the list $x_0, \ldots, x_{i-1}$. Such a context $x_0 : e_0, \ldots, x_{k-1} : e_{k-1}$ is said to be *telescopic.* The consequent $w : e$ does not introduce new variables, and so, all the variables in e are bound. A valid solution for the matching problem (1) is a simultaneous substitution, for $0 \leq i < k$, of a word w_i for x_i in $e_{i+1}, \ldots, e_{k-1}, e$ such that e_i matches w_i and e matches w. There may be not one but many different valid solutions. The algorithm introduced in this paper is exhaustive, that is, it enumerates all the possible solutions. Using lazy evaluation, one can extract *a* solution from the (lazy) list of all solutions, for example, by taking the head of this list, as in the *list of successes* method ([4], §6.5).

Contributed programs and proofs. The Coq function definitions and proofs of soundness and completeness theorems are available on my webpage [11]. The computational content has been translated to Haskell essentially by hand. The Coq source relies on ssreflect libraries [9], which is made because of the high level of support that is offered to functional list combinators by the mentioned libraries.

Outline. The main body of the paper starts, in Section 2, with an intuitive example showing the use of the internals of the Haskell pattern matching library. In Section 3 I recall a historic approach of match-trees for regular expressions with backreferences. For the purposes of efficient formalisation, I introduce a more transparent notion of regular expressions with variables in Section 4. Sections 5–7 contain the decision procedure for the matching problem programmed in Haskell. In Section 8 there are proofs of soundness of vmatch. Completeness is only stated, with the proofs available in the contributed Coq script.

2 A Haskell Example

The symbolic approach of this paper to matching with variables can be illustrated on the following example. There is a finite alphabet over which we define regular expressions. From the finite structure, we also have a decidable equality and a countable structure. The finite alphabet contains four symbols that might, for instance, denote the nucleotides of messenger RNA. In Haskell, we can define the alphabet as a data type like this:

```
data RNA = A | C | G | U deriving (Eq, Enum, Show)
nMolecules = fromEnum U − fromEnum A + 1
```

where nMolecules is the number of different nucleotides computed by utilising the countable **Enum** structure. Next we give names to regular expressions denoting single symbols in our RNA alphabet and, moreover, introduce two symbolic variables (details follow in Section 6):

```
a = Atom A ; c = Atom C ; g = Atom G ; u = Atom U ; x = Var 0 ; y = Var 1
```

A nucleotide is certainly not an atom in terms of physics, however, it is an Atom of a formal language. A nucleotide is either of A, C, G or U. Rather than keeping this information implicit in the data type, we devise a piece of abstract syntax to capture it. The meaning of “or” will be attached to the constructor Alt (from “alternate”). Moreover, we will also introduce the constructor Star (also known as the Kleene star, or “iterate”) to denote any number of repetitions of a given abstract expression. We can give Haskell definitions now:

```
nucleotide = Alt a (Alt c (Alt g u))
nucleotides = Star nucleotide
```

Coming to the main part of the example, suppose we are given the following RNA sequence:

```
rnaSequence = [U, A, G, C, G, U, A, G, C, G, U, U, U]
```

against which we intend to match the following symbolic expression:

```
rnaExpression =
  Conc nucleotides (Conc y (Conc x (Conc y (Conc nucleotides (Conc x nucleotides)))))
```

where the constructor Conc denotes the concatenation of words matched by its arguments. When matching rnaExpression against rnaSequence, we are looking for any possible matches in rnaSequence for symbolic variables x and y that follow the pattern outlined in rnaExpression, namely, an arbitrary sequence followed by x, y and x which are followed by another arbitrary sequence which is followed by x and yet another sequence. Suppose also that the only matches for y that we accept are the sequences UA or AG. This will reduce the search space dramatically. Hence we require a mapping of variables to their acceptable matches. Since variables are indexed with non-negative numbers, we will use a list of expressions. The element with index 1 will denote the set of acceptable matches for Var 1, that is, y, while Var 0 will be unconstrained:

```
rnaContext = [nucleotides, Alt (Conc u a) (Conc a g)]
```

At the last step, we apply the matching function vmatch from Section 6 to compute all the matches under the context assumptions rnaContext:

```
rnaValuations = vmatch nMolecules rnaContext rnaExpression rnaSequence
```

The result is a list of mappings of variables to sequences which, in this example, contains duplicate matches:

```
[[[G,C,G],[U,A]], [[G,C,G],[U,A]], [[C,G,U],[A,G]], [[C,G,U],[A,G]],
 [[G,C,G],[U,A]], [[G,C,G],[U,A]], [[C,G,U],[A,G]], [[C,G,U],[A,G]]]
```

Duplication occurs because the internal function that generates all possible candidate mappings of variables to sequences does not ensure that each mapping occurs only once. Hence the computational effort is multiplied! However, on such a small scale it turns out to be slower to find and remove duplicate candidates compared to using efficient Haskell memoisation for recalling results of previously performed computations. The method of `vmatch` is especially efficient when computing the first possible match, in which case the tail of the list will not be computed.

3 Historic Preliminaries: Language-Theoretic Definition of Backreferences

This definition of *regular expressions with backreferences* (*rewbs*, for short) originates from [5]. A formal language theorist assumes that the opening parentheses in a given formal expression e (with nullary constructors 0 and 1, unary constructors _ (for alphabetic symbols) and * (iteration), and binary constructors $+$ and $\times$) are numbered from the left to the right, and the closing parentheses are numbered in the correspondence with the opening parentheses. For example,

$$\underset{1}{(}\ldots\underset{2}{(}\ldots\underset{3}{(}\ldots\underset{3}{)}\ldots\underset{2}{)}\ldots\underset{4}{(}\ldots\underset{4}{)}\underset{1}{)}$$

Definition 1 (Backreference). *A* backreference $\backslash m$ *where* $m \geq 1$, *matches the contents of the m-th pair of numbered parentheses on the left of it.*

For example, the regular expression $(\underline{a}^*)\times\underline{b}\times\backslash 1$ defines the language

$$\{a^n \cdot b \cdot a^n \mid n \geq 0\}$$

Lemma 1 (Campeanu–Salomaa–Yu pumping lemma, [5]). *Let e be a rewb. Then there is a natural number n such that, for $w \in L(e)$ and $|w| > n$, there is an $m \geq 1$ and a decomposition $w = x_0 \cdot y \cdot x_1 \cdot y \cdot \ldots \cdot x_m$ such that*

1. $|x_0 \cdot y| \leq n$,
2. $|y| \geq 1$,
3. $x_0 \cdot y^j \cdot x_1 \cdot y^j \cdot \ldots \cdot x_m \in L(e)$, *for all* $j > 0$.

From [5] it is known that rewb languages are context-sensitive and incomparable with the family of context-free languages. The latter is due to the observation that the language $\{a^n \cdot b^n \mid n \geq 0\}$ is context-free but cannot be expressed by a rewb (as a corollary of Lemma 1), and the language $\{a^n \cdot b \cdot a^n \cdot b \cdot a^n \mid n \geq 1\}$ is a rewb language but not a context-free one.

In [6], it was proved that rewb languages are not closed under intersection and their emptiness of intersection problem is undecidable. Moreover, since [1] we know that *the matching problem for rewbs is NP-complete* for arbitrary alphabets, and the paper [6] proves that NP-completeness holds even when the target string is over a unary alphabet.

More specifically, to define a matching problem for rewbs we quote the definition of *matching of a string with an rewb* from [6]. An ordered tree t is a valid match-tree for w and e if and only if the root of t is labelled by (w, e) and the following conditions hold for every node u in the domain of t:

1. If $t(u) = (w, \underline{a})$ for some $a \in A$ then u is a leaf node and $w = a$.
2. If $t(u) = (w, F_1 \times F_2)$ then u has two children labelled, respectively, by (w_1, F_1) and (w_2, F_2), with $w_1 \cdot w_2 = w$.
3. If $t(u) = (w, F_1 + F_2)$ then u has one child labelled either by (w, F_1) or by (w, F_2).
4. If $t(u) = (w, F^*)$ then either u is a leaf node and $w = \epsilon$ or u has $k \geq 1$ children labelled by $(w_1, F), \ldots, (w_k, F)$, with $w_1 \cdot w_k = w$.
5. If $t(u) = (w, (_i F)_i)$ then it has one child labelled by (w, F).
6. If $t(u) = (w, \backslash m)$ then u is a leaf node, $(_m\ F\ )_m$ is a subexpression of e, and there is a node v to the left of u such that $t(v) = (w, (_m\ F\ )_m)$ and no node between v and u has $(_m\ F\ )_m$ in its label. In other words, w is the string previously matched by $(_m\ F\ )_m$ in the left-to-right pre-order of nodes.

The paper [5] features a slightly different definition where unassigned back-references are set to match the empty string by default.

The language $L(e)$ denoted by a rewb e over an alphabet Σ is the set of all $w \in \Sigma^*$ such that (w, e) is the root label of a valid match-tree. The matching problem has an operational definition in [6]:

Definition 2 (Matching problem for rewbs). *For some w and e, is (w, e) the root label of a valid match tree?*

The straightforward *deterministic* exponential time algorithm suggested in [1] uses backtracking and enumerates all the possible subwords of w that can be assigned to the variables in e. The worst-case running time of that algorithm is therefore $O(|w|^{2k})$ where k is the number of variables in e. Once an assignment of subwords to variables is defined, the matching problem reduces to variable-free regular expression matching.

Definition 1 is not satisfactory already because it requires a notion of matching, while matching is only defined in Definition 2 which in turn relies on Definition 1. The next section presents a way out of this awkward situation.

4 Regular Expressions with Variables

A suitable definition has to be only slightly more general. In the grammar construction there need not be any reference to matching.

Definition 3 (Revs). *We define* regular expressions with variables (revs) *over an alphabet Σ with variables from a set X by the following grammar:*

$$e ::= 0 \mid 1 \mid \underline{a} \mid x \mid e + e \mid e{\times}e \mid e^*$$

for any $a \in \Sigma$ and any $x \in X$.

In practical regular expressions, the variables (i.e., backreferences) are assigned values in regular expressions. Therefore we have to express this kind of assignments. We can do that for a rev e by introducing a *telescopic context* Δ consisting of pairs $x_0 : e_0,\ \ldots,\ x_{k-1} : e_{k-1}$, where $x_0, \ldots, x_{k-1}$ are distinct variable names and the variables of e_i are contained in the set $x_0, \ldots, x_{i-1}$. The latter is required to exclude circularity. We say that e is *well-defined by Δ* if Δ is telescopic and variables of e are contained in the variables of Δ. (The standard for backreferences left-to-right ordering of the first occurrences of variables in e is not important in this case). We will write $e(x_0, \ldots, x_n)$ for a regular expression e whose variables occur among $x_0, \ldots, x_n$. Let the telescopic context Δ be

$$x_0 : e_0, \quad x_1 : e_1(x_0), \quad \ldots, \quad x_{k-1} : e_{k-1}(x_0, \ldots, x_{k-2})$$

An improvised definition of the matching problem for revs can be the following.

Definition 4 (Matching problem for revs). *The truth value of the below statement is the answer to the question whether e which is well-defined by $x_0 : e_0, \ldots, x_{k-1} : e_{k-1}$ matches w:*
There is a subword u_0 of w such that e_0 matches u_0 and
...
there is a subword u_i of w such that $e_i(x_0, \ldots, x_{i-1})$ matches u_i and
...
there is a subword u_{k-1} of w such that $e_{k-1}(x_0, \ldots, x_i, \ldots, x_{k-2})$ matches u_{k-1} and $e(u_0, \ldots, u_i, \ldots, u_{k-1})$ matches w.

Now it is the right time to mention a decision procedure for matching. Due to the NP-completeness of matching with backreferences (and hence, variables) the standard *partial derivative* construction ([19], [3]) does not apply here. But we can specify it to a given valuation, that is, a mapping from variables to words. For a given valuation v and e whose variables are contained in the domain of v, the set of *non-trivial partial derivatives* $\pi_v(e)$ of e is defined by induction as follows (in the style of [2]):

$$\begin{aligned}
\pi_v(0) &= \emptyset & \pi_v(F + G) &= \pi_v(F) \cup \pi_v(G)\\
\pi_v(1) &= \emptyset & \pi_v(F{\times}G) &= (\pi_v(F){\times}G) \cup \pi_v(G)\\
\pi_v(\underline{a}) &= \{\epsilon\} & \pi_v(F^*) &= \pi_v(F){\times}(F^*)\\
\pi_v(x) &= \pi_v(\underline{v(x)})
\end{aligned}$$

where ϵ is the empty word, concatenation $\times$ is appropriately extended to sets of revs, and $\underline{v(x)}$ denotes a lifting of the word $v(x)$ to a regular expression, that is, a concatenation of successive symbols in $v(x)$. From [19], we know that, for a variable-free regular expression e, the size of the set $\pi_v(e)$ equals the number of distinct alphabetic symbols in e. For e with variables, the size of this set equals the number of distinct symbols in e and v (assuming no redundant variables in v) because the derivatives of a variable are essentially the derivatives of a concatenation.

The formulation of the partial derivative transition matrix follows in Section 7. That matrix and the set of all partial derivatives $\{e\} \cup \pi_v(e)$ (called a *prebase vector* of e) yield a non-deterministic finite automaton accepting the language of e under v. The matching problem for revs is decided by the prebase construction.

5 Generation of Valuations

There are several functions that are essential for combinatorial generation of a list of all subwords of a given word, as it is required by the exhaustive matching algorithm that tries to check all the possible valuations for variables. The folklore generator takes all suffixes of all prefixes of a given list. Dually, it can take all prefixes of all suffixes. Left and right scans are used to construct prefixes and suffixes respectively. There is a difference however with scans in the Haskell prelude: the scans here do not append the seed value x to the list of results, which is due to the fact that definitions by structural recursion are more straightforward in that case. Below is the definition for the left scan:

```
scanlN :: (b → a → b) → b → [a] → [b]
scanlN f z (x : s) = y : scanlN f y s where y = f z x
scanlN f z [] = []
```

and the right scan:

```
rcons s x = s ++ [x]

scanrNr :: (a → b → b) → b → [a] → [b]
scanrNr f z (x : s) = scanrNr f y s `rcons` y where y = f x z
scanrNr f z [] = []

scanrN f z s = scanrNr f z (reverse s)
```

Non-empty prefixes, non-empty suffixes and the list of all non-empty subwords are defined as follows:

```
initsN :: [a] → [[a]]
initsN = scanlN rcons []
tailsN :: [a] → [[a]]
tailsN = scanrN (:) []
segsN :: [a] → [[a]]
segsN = concat . map tailsN . initsN
```

Finally, the most important technical definition of this paper is the following. In order to match in telescopic contexts, a function is required that generates all k-element lists of values of type a where each i-th element satisfies a condition that depends on the previous $i-1$ elements, that is, on the history of matches of the previous context variables. We call k-element lists of that kind *k-sequences*. The computation is performed by kseqs:

```
kseqs :: [a] → ([a] → a → Bool) → Int → [a] → [[a]]
kseqs grnd px k z
  | k == 0 = [z]
  | k > 0 =
      concat (map (r (k − 1) . rcons z) (filter (px z) grnd))
  where r = kseqs grnd px
```

where z is a prefix to be either returned straight away in the trivial case where $k = 0$, or appended to each of the k-sequences on the left, grnd is a given list of ground values from which we take those that satisfy the boolean predicate px z, append them at the right of z and construct all $k-1$-sequences for all these new prefixes. The "for all" functionality is delegated to **map**, the list of satisfying values is constructed by **filter**, and **concat** performs the role of a "reduce" operator that takes the output from **map**. Some more hints on how and why this definition of k-sequences works are coming up in technical Lemmas 2 and 4.

6 Representation of Regular Expressions with Variables

Here are the design choices that are behind the definition of regular expressions with variables:

– The type of regular expressions is polymorphic in the type of alphabet symbols.
– Variables are indexed by natural numbers.
– In computations, the user is supposed to provide the size of the alphabet and the number of variables in a regular expression, together with constraints on these variables.

Revs form the type REV:

```
data REV a = Void | Eps | Atom a | Var Int
           | Alt (REV a) (REV a) | Conc (REV a) (REV a) | Star (REV a)
           deriving Eq
```

The constructors correspond, respectively, to 0, 1, $\underline{a}$, x, $+$, $\times$ and * from Section 4. Since symbols are represented by numbers, words are represented by lists of numbers, and valuations are represented by lists of words. The intended usage of a valuation is to provide a one-to-one correspondence with a list of regular expression constraints on context variables. Also, a suitable notion of a matrix is required to represent the transition function of the NFA produced by the construction detailed further in Section 7. Hence we have

```
type Valuation a = [[a]]
type Mat a = [[a]]
```

Given a valuation v for free variables in e, a *ground match* is a deterministic test whether e matches w when every free variable is substituted with its value according to the valuation. This test is defined as follows:

```
gmatches :: (Eq a, Enum a) ⇒ Int → Valuation a → REV a → [a] → Bool
gmatches n v e w = g w 0 where
  g = λw i →
    case w of
      [] → o !! i
      a : u → any (λk → g u (1 + k)) (m !! i !! fromEnum a)
  t = ptup v e
  o = [has_eps v e' | e' ∈ t]
  m = pmat n v e
```

Here, t is the list of states of the NFA, o is the characteristic list of the output states of the NFA, and m is the matrix of transitions of the NFA. The application of **any** (an iterated disjunction) computes whether the ground match holds for some partial derivative reachable by the transition a from the current state i.

The list of all successful matches for context variables can be constructed by kseqs. It is a matter of using gmatches to filter the appropriate matches:

```
telematch :: (Eq a, Enum a) ⇒ Int → [REV a] → [a] → [Valuation a]
telematch n ctxt w =
  kseqs ([] : segsN w)
    (λv u → gmatches n v (ctxt !! (length v)) u)
    (length ctxt)
    []
```

Lastly, the list of all valid matches for context variables is computed by our main function of interest:

```
vmatch :: (Eq a, Enum a) ⇒ Int → [REV a] → REV a → [a] → [Valuation a]
vmatch n ctxt e w = filter (λv → gmatches n v e w) (telematch n ctxt w)
```

7 Non-deterministic Finite Automaton Construction

In this section I extend the construction of [12] to account for variables. Correctness of this construction is a separate issue and is a purpose of a theorem of Mirkin [19] that was checked using Coq in [12].

A prebase of a regular expression consists of a *prebase vector* which corresponds to the set of states of an NFA, and a *prebase matrix* which corresponds to the transition function of that NFA. Moreover, the NFA defined by a prebase of a regular expression e accepts the language denoted by e. Graphically, we can picture a vector of states and a transition matrix of e as follows (the indexing pattern is chosen on purpose):

$$\begin{bmatrix} e \\ e_0 \\ \vdots \\ e_{m-1} \end{bmatrix} \qquad \begin{bmatrix} c'_0 & \cdots & c'_{n-1} \\ c_{0,0} & \cdots & c_{0,n-1} \\ \vdots & \ldots & \vdots \\ c_{m-1,0} & \cdots & c_{m-1,n-1} \end{bmatrix}$$

In terms of automata, e is the initial state, and each regular expression in this vector that matches the empty word is a final state. The elements of the transition matrix are sets of numbers i such that $0 \leq i < m$. Each number i in a cell refers to the regular expression e_i and its corresponding row i in this matrix. The width of the matrix is n which is the number of symbols in the alphabet of the regular expression (which is always finite). Therefore columns are enumerated by (the indices of) symbols. Each cell c'_j, for $0 \leq j < n$, contains (references to) the partial derivatives of e with respect to the symbol with number k. Likewise, each cell $c_{i,j}$, for $0 \leq i < m$ and $0 \leq j < n$, contains (references to) the partial derivatives of e_i with respect to the symbol with number j. I mentioned that the indexing is not arbitrary: The topmost row stands out because, with the definitions that are given below in this section, there are no references to that row anywhere in the matrix. Thus numbering of partial derivatives in fact starts from the row below it.

The base cases of the recursive computation of the prebase vector and matrix involve constructors Void, Eps and Atom. The first two cases are similar because, except for the empty word, whatever word we choose to residuate these regular expressions with, we obtain Void, with no derivatives. Therefore we still have to provide one row of the prebase matrix in pmat_void and pmat_eps just to express the fact that these regular expressions have no derivatives with respect to any symbol, see Figure 1. The third base case is only slightly bigger. There is a new

```
ptup_void :: [REV a]                    ptup_eps :: [REV a]
ptup_void = [Void]                      ptup_eps = [Eps]
pmat_void :: Int → Mat [Int]            pmat_eps :: Int → Mat [Int]
pmat_void n = [replicate n []]          pmat_eps n = [replicate n []]
```

Fig. 1. Prebases of Void and Eps

derivative with respect to the given symbol with number i. The prebase matrix must have two rows, with the first row containing the reference to the second row in the column of the given symbol. This reference is denoted by 0. The numbering schema in the cells does not refer to the matrix row 0, which means that, when 0 occurs as an element of a cell in the matrix, it in fact refers to the matrix row 1. Other columns are empty, and all the columns in the second row are empty as well since Eps has no derivatives with respect to symbols:

```
ptup_atom :: a → [REV a]
ptup_atom i = [Atom i, Eps]
pmat_atom :: Enum a ⇒ Int → a → Mat [Int]
pmat_atom n a =
  [if j == fromEnum a then [0] else [] | j ∈ [0 .. n − 1]] :
  [replicate n []]
```

In the recursive cases that follow, the vectors are obtained from vectors of the component regular expressions. Consider the case of Alt first. We construct a vector of length 1 + (size t1 − 1) + (size t2 − 1). The vector of derivatives of Alt e1 e2 consist of this expression itself (trivially), the non-trivial derivatives of e1, and the non-trivial derivatives of e2:

```
ptup_alt :: [REV a] → [REV a] → [REV a]
ptup_alt t1 t2 = Alt (head t1) (head t2) : tail t1 ++ tail t2
```

With the dimensions of the matrix known from the vector construction, we have to fill in the first row of the matrix with disjoint unions of references to the corresponding rows in the submatrices of non-trivial derivatives of the component regular expressions. The disjoint union is implemented here (too simply though) by list concatenation. The rows below the first one contain copies of the corresponding rows from the component matrices, however, the rows from the second matrix have been shifted by the size of the first matrix less the first row. Therefore all the references to rows originating from the second matrix should be shifted accordingly. Note, similar to the atomic case, the function $(-\ 1)$ on integers has natural numbers as its image provided that correct matrices are given as input, see Figure 2. The case of Conc is similar to the previous one. However,

```
pmat_alt :: Int → Mat [Int] → Mat [Int] → Mat [Int]
pmat_conc :: Int → Mat [Int] → Mat [Int] → [Bool] → Mat [Int]
```

```
pmat_alt n m1 m2 =
  [m1 !! 0 !! j ++
   [k + nrows_m1 − 1
   | k ∈ m2 !! 0 !! j]
  | j ∈ symbols] :
  tail m1 ++
  [[[k + nrows_m1 − 1
    | k ∈ m2 !! i !! j]
   | j ∈ symbols]
  | i ∈ [1 .. nrows_m2 − 1]]
  where
    symbols = [0 .. n − 1]
    nrows_m1 = length m1
    nrows_m2 = length m2
```

```
pmat_conc n m1 m2 output1 =
  [[m1 !! i !! j ++
    if output1 !! i
    then [k + nrows_m1 − 1 | k ∈ m2 !! 0 !! j]
    else []
   | j ∈ symbols]
  | i ∈ [0 .. nrows_m1 − 1]] ++
  [[[k + nrows_m1 − 1
    | k ∈ m2 !! i !! j]
   | j ∈ symbols]
  | i ∈ [1 .. nrows_m2 − 1]]
  where
    symbols = [0 .. n − 1]
    nrows_m1 = length m1
    nrows_m2 = length m2
```

Fig. 2. Prebase matrices for constructors Alt and Conc

the first **length** t1 elements are concatenations of a derivative of the first component regular expression with the second component expression. The remaining elements represent derivatives for the case when the first expression matches the empty word. Concatenation behaves differently depending on whether or not the first component regular expression matches []. So, we add an extra argument of type [**Bool**] to the function constructing the concatenation matrix to represent this characteristic of each of the derivatives of the first expression. In the matrix of the compound expression, we shift the values of the references to the rows belonging to the second component matrix, see Figure 2.

```
ptup_conc :: [REV a] → [REV a] → [REV a]
ptup_conc t1 t2 = [Conc e (head t2) | e ∈ t1] ++ tail t2
```

The case of Star is similar to the case of Conc, which is intuitively clear and even simpler because there is only one component regular expression that we concatenate zero or more times with itself:

```
ptup_star :: [REV a] → [REV a]
ptup_star t = estar : [Conc e estar | e ∈ tail t]
  where estar = Star (head t)

pmat_star :: Int → Mat [Int] → [Bool] → Mat [Int]
pmat_star n m output =
  head m :
  [[m !! i !! j ++ if output !! i then m !! 0 !! j else []
   | j ∈ [0 .. n − 1]]
  | i ∈ [1 .. nrows_m − 1]]
  where nrows_m = length m
```

A novelty due to the appearance of variables is the dedicated functions computing the prebase vector and the prebase matrix for a word. These functions are used to compute automata for valuations:

```
ptup_word :: [a] → [REV a]
ptup_word = foldr (ptup_conc . ptup_atom) ptup_eps

pmat_word :: Enum a ⇒ Int → [a] → Mat [Int]
pmat_word n = foldr (λa m → pmat_conc n (pmat_atom n a) m [False, True]) (pmat_eps n)
```

Putting it all together, the vector of partial derivatives and the transition matrix are computed by the following functions. First, the prebase tuple:

```
ptup :: Valuation a → REV a → [REV a]
ptup _ Void = ptup_void
ptup _ Eps = ptup_eps
ptup _ (Atom a) = ptup_atom a
ptup v (Alt e1 e2) = ptup_alt (ptup v e1) (ptup v e2)
ptup v (Conc e1 e2) = ptup_conc (ptup v e1) (ptup v e2)
ptup v (Star e) = ptup_star (ptup v e)
ptup v (Var i) = ptup_word (v !! i)
```

The “matches []” characteristic needed for concatenation and iteration:

```
has_eps :: Eq a ⇒ Valuation a → REV a → Bool
has_eps _ Void = False
has_eps _ (Atom _) = False
has_eps _ Eps = True
has_eps _ (Star _) = True
has_eps v (Alt e1 e2) = has_eps v e1 || has_eps v e2
has_eps v (Conc e1 e2) = has_eps v e1 && has_eps v e2
has_eps v (Var i) = v !! i == []
```

At last, the prebase matrix:

```
pmat :: (Eq a, Enum a) ⇒ Int → Valuation a → REV a → Mat [Int]
pmat n _ Void = pmat_void n
pmat n _ Eps = pmat_eps n
pmat n _ (Atom a) = pmat_atom n a
pmat n v (Alt e1 e2) = pmat_alt n (pmat n v e1) (pmat n v e2)
pmat n v (Conc e1 e2) = pmat_conc n (pmat n v e1) (pmat n v e2)
                                [has_eps v e' | e' ∈ ptup v e1]
pmat n v (Star e) = pmat_star n (pmat n v e)
                                [has_eps v e' | e' ∈ ptup v e]
pmat n v (Var i) = pmat_word n (v !! i)
```

Thus we introduced everything required to complete the definition of vmatch that we started in Section 6.

8 Soundness and Completeness of Exhaustive Matching

In this section I prove, with a reference to the contributed Coq script, that regular expression matching with variables as defined by vmatch is sound, that is, the list of results produced by vmatch contains only valid matches. For that, an informal notion of soundness has to be agreed upon. Suppose we construct a list of matches by applying vmatch to some arguments n (the number of symbols in the alphabet), ctxt (regular expressions typing the free variables), e (the regular expression) and w (the word to be matched). As a result, we expect to obtain a list of valuations, v, for the free variables in e. It is sensible for the soundness statement to be a property of the arguments of vmatch and a valuation. The way we can express that an arbitrary variable v belongs in the list of results of vmatch is to use the standard list-based containment predicate. So, we can assume (v ∈ vmatch n ctxt e w) = true. Note that containment is decidable and therefore the clause is convertible with a boolean value. These are all the premisses to the soundness statement. To capture the intended behaviour, the conclusion should say that e does indeed match w under the valuation v, that the size of v is equal to the size of ctxt, and that the value of each variable according to v is matched by the regular expression type of this variable. The whole statement can be written in Coq as follows:

```
Theorem vmatch_sound n ctxt e w v :
  v ∈ vmatch n ctxt e w = true →
  gmatches n v e w ∧
  size v = size ctxt ∧
  ∀i d, i < size ctxt → gmatches n (take i v) (nth Void ctxt i) (nth d v i).
```

In terms of basic functions, the differences with Haskell are essentially that **length** is replaced by size, and there is a different function, nth, to access the elements of a list that has a default value, unlike !! in Haskell. This theorem is stated below more descriptively and less technically. (We will fall back to Haskell notation from now onward.)

Theorem 1 (Soundness of exhaustive matching). *Let* n *be a natural number denoting the alphabet size,* ctxt *be a list of revs denoting the typing context,* e *be a rev, and* v *be a list of words denoting the valuation such that* v *is contained in* vmatch n ctxt e w*. Then the following conditions are satisfied:*

1. e *matches* w *under the valuation* v *(if* w *contains symbol codes greater than* n*, those codes are ignored).*
2. *The size of the valuation is the same as the size of the typing context.*
3. *For each index* i *within the bounds of the typing context, the* i*-th element of the typing context matches the* i*-th value in* v *under the valuation defined by the first* i *values in* v*.*

The proof of Theorem 1 (deferred until the end of this section to get background results first) essentially relies on the corresponding soundness property of kseqs. Its Coq statement is parametric in the choice of the default value d, in the same way as the Coq statement of Theorem 1 above is. Due to this parametricity, we can suppress d altogether from our paper proofs. Default values can be suppressed from Coq definitions and proofs as well at the expense of lifting plain lists ctxt and v to dependently typed tuples.

Lemma 2 (Soundness of generation of k-sequences). *Assume a type* a *whose equality is decidable. Assume also a* **Bool***-valued function (a decidable binary relation)* px *of arguments of types* [a] *and* a*, a natural number* k*, lists* grnd*,* s *and* z *of values of type* a*, such that* s *is contained in* kseqs grnd px k z*. Then, for all indices* i *less then* k*, the relation* px *holds between the prefix of* s *of size* i + **length** z *and the (*i + **length** z*)-th element of* s*.*

Proof. We reason by induction on k. The base case k = 0 is proved vacuously since there is no i less then 0. To understand the inductive step, consider Figure 3. The actual computation involves **map** and is tree-structured, which might

$$\begin{array}{lll}
\mathsf{kseqs} & k & z \\
\mathsf{kseqs} & k-1 & z \text{ `rcons` } \__0 \\
 & \vdots & \\
\mathsf{kseqs} & 1 & z \text{ `rcons` } \__0 \text{ `rcons` } \ldots \text{ `rcons` } \__{k-2} \\
\mathsf{kseqs} & 0 & z \text{ `rcons` } \__0 \text{ `rcons` } \ldots \text{ `rcons` } \__{k-2} \text{ `rcons` } \__{k-1}
\end{array}$$

Fig. 3. Calculation of k-sequences, top-down

suggest reasoning on an explicit tree, however, here we focus on the logical property of the values represented by the triangle of placeholders: The rightmost placeholders in Figure 3 denote sets of filtered values that are appended at the end of the sequences passed over by the corresponding recursive calls on the line above. Thus, calculating top-down, as in Figure 3, we have the information that the filtered values are related by px to the sequence calculated in the line above. Induction on the parameter k, as opposed to calculation, proceeds bottom-up. So, we can use this logical information in the proof of the inductive step when reasoning on k. Hence it suffices to proceed by induction over the structure of the filtered list, **filter** (px z) grnd.

If the filtered list is empty, then so is the list of k+1-sequences, by the usual properties of **map**, **filter** and function composition, leading to a contradiction with containment of s in the list of k+1-sequences.

If the filtered list is y : ys then, unfolding the definition of **concat** in kseqs, we have that either is s contained in kseqs grnd px (z 'rcons' y) or in

concat (**map** (λx $\rightarrow$ kseqs m (z 'rcons' x)) ys)}

The latter case is immediate from the inductive hypothesis on the filtered list. To prove the former case, we start by assuming i = 0. Since the prefix of size z is the list z itself, we have to prove that z and y are related by px, which follows from the logical property of the filtered list. Assume 0 < i. Here we apply the inductive hypothesis for k substituting z 'rcons' y for the prefix and i − 1 for the index. By doing so, we have successfully used the assumption 0 < i to shift the indices by +1 in the inductive hypothesis to apply it on the inductive step. □

Lemma 3 (Size of a *k*-sequence). *As in Lemma 2, we assume a, px, k, grnd, s and z. If s is contained in kseqs grnd px k z then the size of s equals the size of z plus k.*

Proof. Similar to the proof of Lemma 2, we perform induction on k, with the proof of base case trivial because s = z when k = 0. The inductive step is also proved by induction on the structure of **filter** (px z) grnd but simpler: there is no need to refer to the logical property of the filtered values. We unfold the definition of **concat** and prove the two resulting cases by the inductive hypothesis on the list of filtered values and the inductive hypothesis on k respectively. □

Now we can prove all the three properties in Theorem 1.

Proof (of Theorem 1). Let p denote the predicate λv $\rightarrow$ gmatches n v e w and let s denote telematch n ctxt w. (1) Follows by equational reasoning since v $\in$ **filter** p s if and only if p x and v $\in$ s. (2) Follows from Lemma 3. (3) Follows from Lemma 2. □

The statements below testify that the matching procedure of vmatch is indeed exhaustive, that is, complete. The proofs are found in the contributed code [11].

Lemma 4 (Completeness of generation of *k*-sequences). *Assume a, grnd, px, k, s and z as in Lemma 2. Assume that z is a prefix of s. For all indices i less then k, if the relation px holds between the prefix of s of size i +* **length** *z and the (i +* **length** *z)-th element of s, then s is contained in kseqs grnd px k z.*

Theorem 2 (Completeness of matching with variables). *Assume n, ctxt, e, w and v as in Theorem 1. If the conclusions of the Theorem 1 are simultaneously satisfied then v is contained in vmatch n ctxt e w.*

9 Related Work

There is a diversity of regular expression libraries in Haskell, each with its own features. The closest currently available regular expression Haskell library to ours is XHaskell [21,16], released on Hackage under the name of `regex-pderiv` [17]. The library computes partial derivatives of regular expressions. It does not implement backreferences yet. Matching is purely functional and symbolic. The library allows to decide equivalence of regular expressions thanks to the symbolic approach. According to the experimental data in [16], this library performed better on a few tests such as the US address, compared to the C wrapper library `regex-posix` [14].

The author of [14] has also implemented Haskell backends for other C regular expression libraries such as Tagged DFA Regular Expressions (TRE) [15] and Perl-Compatible Regular Expressions (PCRE). These libraries are very efficient but not yet stable. The author also re-implemented TRE in Haskell. Representative tests in [16] show that partial derivative matching is in most cases faster than Haskell TRE matching, however, the former requires more space for computations.

10 Conclusions

I introduced a notion of regular expression with variables (revs) as a solution to unnecessary constraints in the historic definition of regular expressions with backreferences (rewbs). Revs enjoy a simple yet efficient (and sound and complete) matching decision procedure which I also outlined in the paper. The decision procedure yields a list of successful matches for context variables in a regular expression, which allows to extend [17] with backreferences, as a possible direction. Lazy evaluation allows to compute a single element from that list thus alleviating the need for explicit backtracking. Nevertheless, there is a considerable interest in a backtracking matching algorithm, e.g., for strict evaluation or to be used in a monadic computation, which is a current work in progress.

Acknowledgements. I am grateful to the TFP referees whose judgement and advice helped to make the paper better. The research was carried out in the University of St Andrews and was supported by the EU FP7 Marie Curie fellowship 253162 “SImPL”.

References

1. Aho, A.V.: Algorithms for finding patterns in strings. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. A: Algorithms and Complexity, pp. 255–300. The MIT Press (1990)
2. Almeida, J.B., Moreira, N., Pereira, D., de Sousa, S.M.: Partial derivative automata formalized in Coq. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 59–68. Springer, Heidelberg (2011)

3. Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. Theor. Comput. Sci. 155(2), 291–319 (1996)
4. Bird, R., Wadler, P.: Introduction to Functional Programming. Prentice Hall (1988)
5. Campeanu, C., Salomaa, K., Yu, S.: A formal study of practical regular expressions. International Journal of Foundations of Computer Science 14, 1007–1018 (2003)
6. Carle, B., Narendran, P.: On extended regular expressions. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 279–289. Springer, Heidelberg (2009)
7. Castiglione, G., Restivo, A., Salemi, S.: Patterns in words and languages. Discrete Appl. Math. 144(3), 237–246 (2004)
8. Champarnaud, J.-M., Ziadi, D.: Canonical derivatives, partial derivatives and finite automaton constructions. Theor. Comput. Sci. 289(1), 137–163 (2002)
9. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA (2011)
10. Komendantsky, V.: Formal proofs of the prebase theorem of Mirkin (2011), Coq script available at, `http://www.cs.st-andrews.ac.uk/~vk/doc/prebase.v`
11. Komendantsky, V.: Coq and Haskell sources of the decision procedure for regular expression matching with variables (2012), `https://bitbucket.org/vkomenda/vmatch`
12. Komendantsky, V.: Reflexive toolbox for regular expression matching: Verification of functional programs in Coq+Ssreflect. In: The 6th ACM SIGPLAN Workshop Programming Languages meet Program Verification, PLPV 2012, Philadelphia, USA (January 24, 2012), For contributed proofs, see [10]
13. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Infor. and Comput. 110(2), 366–390 (1994)
14. Kuklewicz, C.: The regex-posix package (2012), `http://hackage.haskell.org/package/regex-posix/`
15. Laurikari, V.: NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In: SPIRE 2000, pp. 181–187 (2000)
16. Lu, K.Z.M., Sulzmann, M.: A library implementation of XHaskell (2010), `http://code.google.com/p/xhaskell-library`
17. Lu, K.Z.M., Sulzmann, M.: The regex-pderiv package (2012), `http://hackage.haskell.org/package/regex-pderiv`
18. McIlroy, M.D.: Enumerating the strings of regular languages. Journal of Functional Programming 14, 503–518 (2004)
19. Mirkin, B.G.: New algorithm for construction of base in the language of regular expressions. Tekhnicheskaya Kibernetika 5, 113–119 (1966); English translation in Engineering Cybernetics (5), 110-116 (September-October 1966)
20. Reidenbach, D., Schmid, M.L.: A polynomial time match test for large classes of extended regular expressions. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 241–250. Springer, Heidelberg (2011)
21. Sulzmann, M., Lu, K.Z.M.: XHaskell – adding regular expression type to Haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 75–92. Springer, Heidelberg (2008)
22. Yu, S.: Regular languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 1, Word, language, grammar, pp. 41–110. Springer (1997)

OCaml-Java: OCaml on the JVM

Xavier Clerc

ocamljava@x9c.fr
http://www.ocamljava.org/

Abstract. This article presents the OCaml-Java project whose goal is to allow compilation of OCaml sources into Java bytecodes. The ability to run OCaml code on a Java virtual machine provides the developer with means to leverage the strengths of the Java ecosystem lacking in the OCaml world. Most notably, this includes access to a great number of libraries, and foundations for shared-memory concurrent programming. In order to achieve this, the OCaml-Java project does three contributions: *(i)* an optimized compiler and runtime support to achieve acceptable performance, *(ii)* an extension of the classical OCaml typer to allow manipulation of Java elements from the OCaml world, and *(iii)* a library dedicated to concurrent programming.

Keywords: OCaml, Java, compiler, typer, concurrent programming, shared memory, software transactional memory.

1 Introduction

We will start by examining in this section the necessity, or at least the interest, of an OCaml-to-Java compiler for various programming tasks. Then, Section 2 will present the main characteristics of the OCaml-Java project, and Section 3 will present the compatibility with the official OCaml implementation. Section 4 will introduce some extensions made to the original OCaml typer to allow the manipulation of Java classes from an OCaml program. Section 5 will give a glimpse of the library designed to allow easy multicore programming in OCaml. Finally, Sections 6 and 7 will respectively put forward information about performance and future work.

Why Add a Java Backend to OCaml?

The official OCaml[1] distribution already ships with a compiler to OCaml bytecode, as well as a bunch of native compilers for various architectures (mainly `amd64`, `ia32` and `powerpc`, but also `ia64`, and `arm`). It is thus legitimate to question the need for a new backend: the official compilers seem to cover the spectrum from portability (through dedicated bytecode and virtual machine) to performance (through native compiler).

Our understanding is that a Java backend is nevertheless interesting for at least three reasons:

- access to Java libraries, frameworks, and infrastructure;
- access to a multicore facility;
- access to a *secure* environment (in the sense of the Java security manager).

Java Libraries: The Java language features a huge community that is able to deliver a tremendous manpower, that in turn is able to develop libraries for almost any need. This is perfectly shown by the availability of various GUI toolkits, bindings to almost all database systems, or support for 3D and visualization tasks. Moreover, the Java ecosystem gives rise to genuine infrastructure elements such as, for example, web services or industrial-strength implementations of the map-reduce programming model.

Being able to access these various technology bits is of prime concern for a developer. The best language in the world will be less than useless if its support library is so small that she has to develop in-house every elements of her application. This explains why binding technologies are so important in practice even if of little theoretical interest. Our understanding of the current situation is that by targeting the JVM and giving means to access its resources, we can provide OCaml programmers with libraries that are beyond the OCaml community manpower.

Multicore Programming: The official OCaml implementation relies on the existence of a global runtime lock that is mandatory for two reasons: the garbage collector is neither parallel nor concurrent, and some core libraries are not reentrant. The latter point is quite easy to fix by rewriting some routines, but the former is a much bigger endeavor. Some courageous developers launched a project to tackle the issue, and proposed a prototype for the `x86_64` architecture [2]. This is a first and huge step toward a parallel runtime for OCaml but will need a great amount of work, and will have to be re-done on each supported architecture.

In the time being, due to the general availability of multicore computers, some other developers have proposed libraries allowing to take profit of multiple cores by using multiple processes rather than threads. The popular map-reduce model now has several implementations in OCaml, ranging from simple libraries [3] [4] to complete Hadoop-like implementations [5]. Last but not least, the JoCaml dialect [6] provides language extensions for parallel, and even distributed programming.

All these projects are interesting in their own rights. However, they come short when one really needs to work in a shared-memory model. At best, the current solution will allow to share only some memory parts through memory-file mapping but this will clearly badly interact with the garbage collector (in practice, data shared among several processes cannot be reclaimed, as each processor uses its very own garbage collector that is unaware of the other ones). At the opposite, as soon as the OCaml source is compiled to run on the top of a JVM, it enjoys the presence of a parallel garbage collector, and can rely on Java libraries to implement shared-memory concurrent applications at the lowest development cost.

Secure Execution Environment: While clearly the less appealing reason, it is not infrequent in industrial settings to use the capabilities from the Java security manager to control the execution of an application. This proves useful mainly because the rights given to applications can be far more fine-grained than when relying on classical POSIX permissions for example.

Some Internet services and facilities can also use security managers in order to safely allow the shared hosting of untrusted codes, in computation farms or web services for example.

Why Add an OCaml Compiler to the Java Ecosystem?

If we focus on high-profile languages, we can state that the Java ecosystem already features two languages that bring the benefits of functional programming to the JVM: Clojure [7] and Scala [8].

Clojure is often described as a revival of the venerable LISP language, with a particular emphasis on concurrent programming and efficient data structures. The obvious consequence of the LISP filiation is that Clojure does not impose any form of typing discipline. The question of whether typing discipline is a good thing is a highly controversial issue but the developer should at least be able to choose the tool that suits its need from the toolbox. As a consequence, it seems desirable for the Java ecosystem to provide both dynamically- and statically-typed functional languages.

Scala is perhaps the most used JVM language after Java. Scala can be regarded as quite similar to OCaml itself, by explicitly being a multi-paradigm language with support for functional, imperative, and object-oriented programming. Scala does a great way in blending with the Java typing model. However, we consider that the typing possibilities of OCaml are quite superior:

- type inference, allowing shorter and uncluttered programs;
- pattern exhaustivity and redundancy checks, allowing greater software security;
- functors, allowing powerful abstractions over modules;
- first-class modules (introduced in OCaml 3.12.0), allowing to cross the wall between modules and values;
- GADTs (introduced in OCaml 4.0.0), allowing enhanced expressivity of data type invariants.

As a consequence, even if Scala and OCaml share common principles such as support for multiple paradigms and pragmatic handling of side-effects, we think that the two languages are different enough to prove useful to different communities.

Evolution of the JVM

The evolution of the JVM platform seems to indicate that the time has come for serious functional programming inside the Java ecosystem. Indeed, the last major version of the JVM (namely 1.7) has shown remarkable progress that allows to easily get decent performance for functional code.

The first element is the introduction of the so-called `invokedynamic` facility. The ability to encode custom method dispatch besides the one actually used by the Java language is quite appealing, and also had the consequence of the implementation of method handles. For a compiler implementor, method handles are gems because they provide a simple and very efficient way to encode function pointers, and thus closures.

The second element is the new *G1* garbage-collector that is far more suited to functional languages than the previous ones, because its performance will not plummet under heavy allocation of short-lived objects. Practically, this means that the allocation/collection pattern often observed in functional programs can be handled quite well by modern JVMs.

The third element is the fork/join framework. The newly introduced classes related to concurrent programming are extremely important as they constitute a second and far simpler way to program concurrent applications and drop the classical *mutex*-based programming that is not only error-prone but also lacks good properties such as scalability or composability.

And this is not the end of the story. Among the possible extensions envisioned for Java 1.8, the following are often listed:

– introduction of *lambdas* [9], that would hugely reduce the semantic gap between the Java language and functional languages, thus allowing easier integration and library reuse;
– introduction of more concurrent programming patterns [10] (such as parallel array operations, or streams), thus easing multicore programming;
– introduction of tail call optimizations (either only in the JIT compiler, or through a new bytecode marking a method call as terminal), thus preventing the developer to encounter "porting issues"[1] when using a functional language on top of a JVM.

2 Overview of OCaml-Java

The OCaml-Java project consists of three parts:

– the actual compiler, acting as its central component;
– a runtime support comprising elements needed by compiled code to run (*e. g.* representation of values), as well as a port of the standard library primitives (that were originally written in C);
– a library that is specific to OCaml-Java (*i. e.* that cannot be used with the original implementation), and provides support for concurrent programming.

The runtime support also includes a port of the `ocamlrun` program, that is the virtual machine used to execute programs compiled to OCaml bytecode. It is thus possible to run bytecode-compiled programs on a JVM with absolutely no effort, at the only expense of a greater execution time.

[1] The term may be controversial, but in practice some functions may be unable to run due to memory exhaustion when tail call optimization is not available.

The OCaml-Java compiler supports two modes of linking and three kinds of *applications*. The modes of linking allow the developer to choose between static and shared linking, defaulting to static. In static mode, a standalone jar file is produced with all needed `.class` files, while in shared mode the produced jar file contains references to other jar files (such as those from libraries).
The three kinds of *applications* supported by the compiler are:

– *genuine applications*, that are truly independent (except maybe regarding jar files to be found in the classpath);
– *applets*, that are designed to be run inside a web browser to provide additional interactivity to web pages;
– *servlets*, that are designed to be run in containers serving web pages computed on the server side.

Finally, OCaml-Java also provides support for the Java scripting framework (as specified in the `javax.scripting` package). This means that it is possible to evaluate OCaml code snippets from a Java application by just adding the OCaml-Java scripting jar file to the classpath.

3 Compatibility with Original Implementation

The compatibility of OCaml-Java with respect to the original implementation is high but not perfect. Of course, all language constructs are properly supported by the compiler, and most differences can only be observed in libraries.

Indeed, although all primitives from the libraries shipped with the original distribution have been ported, some are not 100% compatible. Most of the time, this is due to the lack of support provided by the JVM for a given functionality (*e. g.* the support for POSIX routines is only partial). Sometimes, the compatibility may only be partial due to the huge amount of work that would be required for a relatively small pay-off: the support of `Tk` falls in this category. Anyway, a developer deciding to use OCaml-Java would probably also want to switch to a Java graphical toolkit.

Another point that is worth noting about compatibility is that the decision to be highly compatible with the original implementation sometimes imposes a burden regarding execution time. As an example, we have to use the very same representation for closures because some core OCaml libraries are based on this representation. Practically, this means that some optimization opportunities are lost in order to keep compatibility.

Finally, some limitations over OCaml-Java also come from restrictions set on us by the JVM itself. The two most striking examples are the limit to a method size, and the lack of support for tail call optimization. Regarding the limit to a method size (64 Kb), there seems to be no easy way for the compiler to find a workaround. As a consequence, the developer may face a compiler error message stating that a given function is too big and should be split[2].

[2] Sometimes, it is also sufficient to reduce inlining aggressiveness.

Regarding the lack of tail call optimization, we made the same decision that almost every language implementor also made. The reader may be aware of compilation techniques that make it possible to overcome this limitation (*e. g.* by using trampolines, or CPS transformations). However, these workarounds tend to cause other problems (mainly in code size, and execution speed). As a consequence, the consensus among language implementors on the JVM is to support tail call optimization only in the specific case of self recursion. In this very case, a jump to the start of the method is sufficient[3].

4 Typer Extensions

The Case for a Typer Extension

When facing the necessity to provide interoperability between languages, several choices can made such as:

- using an exchange language (*e. g.* XML, JSON, *etc.*);
- using bridge generators (*e. g.* based on an IDL[4]);
- embedding the typing of one language into the other one.

The first solution is quite simple but not always adequate for different reasons: (*i*) the exchange language is often far less expressive than the two languages that wish to communicate, (*ii*) it implies that the two languages do not share entities in memory but only communicate in a message-passing style.

The second solution, which was used (through Nickel[5]) in the previous versions of OCaml-Java, is slightly better because it provides a tighter coupling. However, it implies to be able to represent types and values of one language into the other one. This is not always possible, and when it is, it may entail cumbersome constructs that feel unnatural to the developer. We think that the interoperability between OCaml and Java falls in this category because the object systems are very different (nominal *vs* structural typing, single *vs* multiple inheritance, presence *vs* absence of method overloading, *etc.*).

Moreover, one has to describe the list of elements to build bridges for (implying the use of an IDL), and this adds unnecessary burden on the build process.

The limitations of the two previous options led us to think that embedding the typing of Java elements into the OCaml compiler would indeed provide the simplest interface to the developer. Similar work has already been done inside the Standard ML community [11] [12] and proved quite satisfactory. This approach was not chosen in previous OCaml-Java versions because we felt uncomfortable with two object models in the very same language. However, as said above, constraining one model to fit into the other one was found to be very unpleasant for the developer.

[3] This technique cannot be generalized to arbitrary recursion because Java does not allow to jump into another method.

[4] Interface Description Language

[5] `http://nickel.x9c.fr`

Base Elements

In designing the extensions, we decided to grant the following properties:

– the extensions shall not modify the existing syntax of OCaml;
– the extensions shall compile to *plain* Java bytecode.

The former is important to us because we often rely on tools working at the source level of an OCaml codebase, and do not want these tools to be impacted by our support for Java. The later is important to keep the overhead due to OCaml-Java to a bare minimum; we explicitly do not want to rely on mechanisms such as reflection or, worse, bytecode generation at runtime.

Given those constraints, we first decided to represent Java instances with a polymorphic abstract type that will enjoy special typing rules. This decision is indeed quite important in its own right, because it clearly indicates that we will not rely on the object system of OCaml to encode the Java class hierarchy. This stems from the fact that the object models of the two languages are really different, and attempts to encode one into another (such as Nickel, or O'Jacare[6]) lead to awkward OCaml class definitions and difficult-to-understand error messages.

As a consequence we define a type `'a java_instance` that represent instances of class `'a`. As soon as we do so, we have to introduce a first modification in order to designate, for example, instances of class `java.lang.String`. Indeed, it is not possible to write the type `java.lang.String java_instance` and we decided that Java name classes should be written in OCaml types with simple quotes rather than dots to separate their various parts, leading to `java'lang'String java_instance` which is a perfectly legitimate OCaml type. Of course, this implies that the elements *under* the `java_instance` constructor are not treated as regular OCaml types, but as special elements designating Java classes.

Now that we can designate Java instances, we have to provide means to create and manipulate them. In order to achieve this, we heavily rely on the reuse of a hack already used by the OCaml typer to handle `printf` format strings. In the remainder of this section, we will first briefly describe what the `printf` hack consists in, and then explain how we devised a variant to call Java constructors. The very same principle is applied in the OCaml-Java compiler to access fields, and to call methods.

The `printf` Hack

As already stated, the `printf` hack is used in the OCaml compiler in order to handle the format strings used by all the `printf` variants. The idea is brilliantly simple:

– the format strings are given a special type, namely `('a, 'b, 'c) format`;
– whenever an argument with such a type is waited, the compiler actually looks for a literal string and parses it to determine the values actually waited by the function and encodes it into the type parameters of type `format`;

[6] `http://www.pps.jussieu.fr/~henry/ojacare/`

– at runtime, the `printf` function reparses the format string to determine the number and types of parameters to actually access.

To illustrate this behavior, here is the type of the `Printf.printf` function:

```
val printf : ('a, out_channel, unit) format -> 'a
```

and the type of the expression `Printf.printf "key: %s, value: %d"`, making it clear that is should now be passed a string and an integer:

```
string -> int -> unit
```

Basically, the `printf` hack just short-circuited the normal behavior of the typer in order to bind the `'a` type variable to `string -> int -> unit`.

Example: Construction of Java Instances through OCaml Code

The OCaml-Java support library provides a function `Java.make` with signature `('a, 'b) java_constructor -> 'a -> 'b`. The `java_constructor` type is akin to the aforementioned format string, and its two type parameters have the following semantics:

– `'a` encodes the parameters to be passed to the constructor;
– `'b` encodes the result of the constructor.

As an example, in order to create a button, a developer can write the call `Java.make "javax.swing.JButton(java.lang.String,javax.swing.Icon)"` where:

– `'a` will be bound to `(java'lang'String java_instance * javax'swing'Icon java_instance) java_parameters`;
– `'b` will be bound to `javax'swing'JButton java_instance`.

The type named `java_parameters` is handled specifically by the compiler, and serves two purposes: first, the typer uses it to know that parameters passed underneath are covariant; second, the code generator requires that a literal tuple is passed in order to clearly disallow partial application of Java constructors. The first purpose is of course the most important one from a user's perspective: without it the second parameter of our example should be *exactly* an instance of `javax.swing.Icon`[7], with it the second parameter can be an instance of any class being a child of `javax.swing.Icon`.

Finally, the actual compilation differs from the `printf` hack. The *format string* is erased, as it is not needed at runtime. Moreover, no call is made to the surrogate `Java.make` function. This call is replaced by the following bytecode sequence:

```
new javax.swing.JButton
dup
# loading of first parameter on stack
# loading of second parameter on stack
invokespecial javax.swing.JButton.<init>(java.lang.String,
  javax.swing.Icon):void
```

[7] Which is not even possible as `javax.swing.Icon` is an interface.

Advanced Example: Proxies

Leveraging the full power of Java libraries cannot usually be done by solely constructing instances and accessing their fields and methods. It is often necessary, to program GUI or XML handling for example, to be able to register callbacks responding to given events.

Currently, the OCaml-Java compiler allows the developer to *implement* an interface using OCaml code. Here "*implement*" should be understood in its Java keyword sense, that is the possibility to give the code of a class respecting the signature imposed on it by an interface.

The implementation of an interface is done by using Java proxies, and the typing of OCaml code with a Java interface is once again done by using the `printf` hack. The `Java.proxy` function takes a string that should represent the fully qualified name of a Java interface, and the extended typer imposes the second parameter to be an OCaml object providing the methods required by the interface. As a result, the function returns a Java instance implementing the interface. The following source code shows how an instance implementing an action listener can be programmed in OCaml[8]:

```
let action_handler =
  proxy "java.awt.event.ActionListener"
    (object
      method actionPerformed event =
        let desc =
          Java.call
            "java.lang.Object.toString():java.lang.String"
             event in
        Printf.printf "event: %S\n" desc
    end)
```

This method of embedding Java typing into OCaml through `printf`-like hacks, albeit somewhat close to FFI[9], differs from most FFI mechanisms by being type safe. Indeed, the compiler do not take for granted that passed strings are correct with respect to Java classes, it will actually check that each referenced entity is defined and correctly used.

5 Concurrent Library

As stated in the introduction, one of our major objectives in developing the OCaml-Java project is to allow shared-memory concurrent programming in the OCaml world. By making the OCaml-Java runtime fully reentrant, and by using the garbage collector of the JVM, we lift the first two obstacles on our road to concurrency.

[8] The `event` parameter of the `actionPerformed` method from interface `java.awt.event.ActionListener` has type `java.awt.event.ActionEvent`

[9] Foreign Function Interface.

Now comes what is probably the trickiest part: designing a support library to make concurrency as easy as can be. Hopefully, even if things are not settled yet in the concurrency world, there is quite a lot of expertise on this matter. Developers start to grasp that *traditional* mutex-programming is way too error-prone, and explore alternative ways of structuring concurrent programs. Moreover, there is a raising perception that side-effects are particularly evil in a concurrent setting.

Favoring *Implicit* Concurrency

The official OCaml distribution ships with some means for concurrent programming. The foundations are provided by the well-known thread/mutex/condition triad from POSIX filiation. Furthermore, the `Event` module provides higher abstractions inspired by the Concurrent ML system [13].

The first addition from OCaml-Java is made through a bunch of *atomic* modules (*e. g.* `AtomicInt`, `AtomicFloat`, *etc.*). These modules are the direct counterparts of the classes from the `java.util.concurrent.atomic` Java package. Their goal is to provide atomic accesses without explicitly resorting to a mutex. Moreover, they provide more complex operations such as updates through predefined operations, and compare-and-swap.

The second addition from OCaml-Java is the ability to leverage the power of the fork/join framework. We decided to provide less generality than the Java fork/join library, allowing easier use. As an example, we define a function named `split` that separates the fork/join logic into two functions. The signature of `split` is `('a -> ('a * 'a) option) -> ('b -> 'b -> 'b) -> ('a -> 'b) -> ('a -> 'b)` where:

– the first parameter is the *fork* function, that for a given input value decides whether the computation should be split (returning `Some (x, y)`) or not (returning `None`);
– the second parameter is the *join* function, that combines the output values of two sub-computations;
– the third parameter is the original function whose computations can be done in parallel (it is obviously the developer's duty to ensure such a property).

When applied, `split fork join f` will thus return a new function `f'` with the same signature as `f` but that is able to compute its result using several cores. As an example, the infamous[10] Fibonacci example can be coded this way:

```
let fork x =
  if x <= threshold then
    None
  else
    Some (x - 1, x - 2)
```

[10] Infamous because it is by no mean optimal, and is only used to exhibit the typical skeleton of a parallelized function.

```
let join x y = x + y

let rec fibo x =
  if x <= 1 then
    1
  else
    (fibo (x - 1)) + (fibo (x - 2))

let parallel_fibo = split fork join fibo
```

Atomicity through Transactions

Rather than relying on some mutex discipline, the use of software transactional memory is becoming increasingly popular over the years. In the OCaml world, a successful experiment has been done through AtomCaml [14]. We reused almost the same idea, albeit using a totally different implementation.

What differs from AtomCaml (and other *full* software transactional memory systems) on the surface is that the developer has to explicitly label the elements that should be protected by transactions. This may be seen as unnecessary burden, but is in fact quite coherent with the OCaml language itself: the developer is already required to explicitly state that a variable is actually holding a reference to a value rather than a bare value (by using the `ref` function).

The same applies here: the developer has to indicate that a variable is used as a reference to a value that should be protected by transactions. In order to do so, she has to use the `STM.ref` function. Transactions are then executed by calling the `STM.run` function as in the following example based on the canonical bank example:

```
let account_a = STM.ref 1000
let account_b = STM.ref 2000

let transfer ammount =
  STM.run (fun get set ->
    let a, b = get account_a, get account_b in
    set account_a (a + ammount)
    set account_b (b - ammount))
```

As shown by the code sample above, the higher-order function `STM.run` calls its parameter with two functions (namely `get` and `set`) that act as bare accessors to values created through the `STM.ref` function.

Lists Considered Harmful

Any functional programmer is well-versed into list processing, and consequently tends to favor this data structure for many kinds of computations. However, it is well known that lists are not well-suited to parallel computation. At the opposite, arrays are frequently used for parallel computation because they are easy to scatter and gather (following the terminology set by the MPI library).

For this very reason, we extended the OCaml-Java library with a module named `ParallelArray` providing easy parallel computations. The `ParallelArray` module has a signature that is compatible with the one of the `Array` module from the standard library of the original OCaml distribution. Practically, this means that any code using the `Array` module without making any assumption on the order in which elements are treated can be parallelized by just replacing a module by another one. This may even be done by shadowing the original `Array` module at the beginning of a source file through the following binding:

```
module Array = ParallelArray.
```

We want to stress the fact that the functions from `ParallelArray` are guaranteed to have a signature that is compatible with their counterpart from `Array`, but not the exact same signature. Indeed, to allow a proper use of parallel operations, we have added some optional parameters to some functions, in order to be able to specify the number of *chunks* in the array as well as how worker threads should be created. If one really needs to have the very same signature as the `Array` module, this can be achieved by applying the `ParallelArray.Make` functor to a module that barely provides fixed values for the optional parameters.

6 Performance

Settings

We present in this section the benchmarks conducted to assess the ability of the OCaml-Java compiler to be used in performance-critical applications. As of today, we applied only several benchmarks from the Computer Language Benchmarks Game[11], namely:

- *binarytrees*, exercising recursive functions over trees;
- *fannkuch*, exercising integer computations over arrays;
- *mandelbrot*, exercising float computations;
- *meteor*, exercising integer computations over arrays;
- *nbody*, exercising float computations;
- *revcomp*, exercising string computations and i/o;
- *spectralnorm*, exercising heavy float computations over arrays;
- *threadring*, exercising threads and mutexes.

All comparisons have been performed on the `amd64` architecture, using the following compilers:

- the native OCaml native compiler (version 3.12.1), because it will tell how our backend competes;
- the Scala (version 2.9.1-1) and Clojure (version 1.3.0) compilers, because they are comparable languages available on the JVM;
- Java itself, because in some sense it gives the baseline corresponding to maximum performance.

[11] `http://shootout.alioth.debian.org/`

Numbers

All tests are performed using Java version 1.7.0_02 and executed with the `-server`, `-XX:+TieredCompilation`, and `-XX:+AggressiveOpts` options. For each compiled version, five runs are successively launched, the best and worse times are removed, and the mean of the three other values is kept. Table 1 presents both raw numbers, as well as ratios to *reference* compilers.

Table 1. Some benchmarks from the Computer Language Benchmarks Game, using various compilers

benchmark	`javac`	`ocamlopt`	`ocamljava`	`scalac`	`clojure`
binarytrees (absolute)	12.06	29.18	50.54	12.56	27.84
... (ratio to `javac`)	-	2.42	4.19	1.25	2.31
... (ratio to `ocamlopt`)	0.41	-	**1.73**	0.43	0.95
fannkuch (absolute)	24.88	58.42	194.80	25.95	96.44
... (ratio to `javac`)	-	2.35	7.83	1.04	3.88
... (ratio to `ocamlopt`)	0.43	-	**3.33**	0.44	1.65
mandelbrot (absolute)	11.09	40.98	60.67	21.29	53.47
... (ratio to `javac`)	-	3.70	5.47	1.92	4.82
... (ratio to `ocamlopt`)	0.27	-	**1.48**	0.52	1.30
meteor (absolute)	0.31	0.74	5.28	5.10	10.35
... (ratio to `javac`)	-	2.39	17.03	16.45	33.39
... (ratio to `ocamlopt`)	0.42	-	**7.14**	6.89	13.99
nbody (absolute)	11.11	13.43	12.93	11.84	19.96
... (ratio to `javac`)	-	1.21	1.16	1.07	1.80
... (ratio to `ocamlopt`)	0.83	-	**0.96**	0.88	1.49
revcomp (absolute)	2.89	11.11	21.89	5.58	17.90
... (ratio to `javac`)	-	3.84	7.57	1.93	6.19
... (ratio to `ocamlopt`)	0.26	-	**1.97**	0.50	1.61
spectralnorm (absolute)	6.88	10.92	28.98	7.21	14.32
... (ratio to `javac`)	-	1.59	4.21	1.05	2.08
... (ratio to `ocamlopt`)	0.63	-	**2.65**	0.66	1.31
threadring (absolute)	35.85	34.87	38.98	1.90	9.62
... (ratio to `javac`)	-	0.97	1.09	0.05	0.27
... (ratio to `ocamlopt`)	1.03	-	**1.12**	0.05	0.28

Even if we have not yet performed a great number of benchmarks, some preliminary conclusions can already be drawn. The ratio of `ocamljava` to `ocamlopt` varies from 0.96 to 7.14, and is below 3 in six benchmarks among eight. When looking at the benchmarks producing the worse results, it becomes clear that `ocamljava` is less competitive when computation is done over `int` values. This is not surprising, as such values are always unboxed in `ocamlopt` (using a bit to distinguish them from pointer values) while other values are boxed. Due to the lack of support for tagged values in the JVM, `ocamljava` boxes all values. Other benchmarks perform intensive computations on values that are boxed in both `ocamlopt` and `ocamljava`, allowing the latter to be on average less than two times slower than the former.

7 Future Work

In the short term, the most important issue to address is still related to performance. Even if the previous section showed that performance is in fact quite acceptable, there is still room for improvement. As an example, we have not yet conducted any inquiry regarding how different garbage collector parameters could affect performance; given that default parameters for these values are tailored for average Java programs, it seems quite plausible to be able to get better performance by tweaking them.

Then, besides compiler and runtime optimizations, there are mainly four major areas of possible enhancement:

- access to OCaml code from the Java side;
- support for Java generics;
- typing extensions related to parallel programming;
- miscellaneous helpers.

Java-to-OCaml Accesses

The objective is to provide means to call OCaml code from Java. Currently, it is already possible through scripting but, as the name suggests, this implies that the OCaml code will have to be compiled at runtime. It would be a great improvement to be able to compile Java classes against OCaml code produced by the OCaml-Java compiler.

Support for Java Generics

The typing extensions to OCaml presented in this paper do not handle Java generics. This means that the type of a list will always be `java.util.List`, independently of the elements actually stored in the list. Since version 1.5, Java supports so-called generics which are broadly akin to type parameter in ML-like languages. As a consequence, a list of strings can be given the more precise type `java.util.List<String>`. Moreover, it is possible (through `super` and `extends` keywords) to indicate to the Java compiler that the type parameter is covariant or contravariant. The OCaml-Java compiler should be enhanced to support this finer-grained typing of Java instances.

Types for Concurrent Programming

In this paper, we presented some elements of a library designed to allow easy concurrent programming. The library currently recycles well-known concurrent ideas such as fork/join patterns, parallel array primitives, or software transactional memory. Those ideas are barely translated into OCaml. We are interested by the study of typing extensions that could be developed in order to gain better control and/or understanding over the concurrent computations. As an example, we could imagine to devise a typing extension (or a syntax extension) to ensure that the developer uses the accessor functions related to the current transaction, and not those from another one.

Miscellaneous

Although it is possible to fully manipulate Java elements from OCaml programs, some helpers may be provided to make such manipulation less verbose. In the current version, the developer has to use fully qualified names. It should be possible to provide a construct akin to Java `import` in order to allow the use of short class names.

Furthermore, it could be possible to shorten constructor or method signatures when there is no ambiguity. As an example, if there is only one method with the requested name, the developer should not have to give its signature but only its name (or, say, its name and arity if it is sufficient to lift the ambiguity).

Such enhancements entail no change from a theoretical point of view, but may prove of great value from a practical standpoint.

References

1. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: Objective Caml (OCaml) programming language website, `http://caml.inria.fr/`
2. Chailloux, E., Canou, B., Wang, P.: OCaml for Multicore Architectures, `http://www.algo-prog.info/ocmc/web/`
3. Filliâtre, J.-C., Kalyanasundaram, K.: Functory: A Distributed Computing Library for Objective Caml. In: Peña, R., Page, R. (eds.) TFP 2011. LNCS, vol. 7193, pp. 65–81. Springer, Heidelberg (2012)
4. Danelutto, M., Di Cosmo, R.: Parmap: minimalistic library for multicore programming, `https://gitorious.org/parmap`
5. Stolpmann, G.: Plama: Map/Reduce and distributed filesystem, `http://plasma.camlcity.org/`
6. Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: JoCaml: A language for concurrent distributed and mobile programming. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 129–158. Springer, Heidelberg (2003)
7. Hickey, R.: The clojure programming language. In: Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, pp. 1:1. ACM, New York (2008)
8. Odersky, M., et al.: The Scala Language, `http://www.scala-lang.org/`
9. Goetz, B., et al.: JSR 335: Lambda Expressions for the Java Programming Language
10. Lea, D.: Concurrency JSR-166 Interest Site
11. Benton, N., Kennedy, A.: Interlanguage working without tears: blending SML with Java. SIGPLAN Not. 34(9), 126–137 (1999)
12. Benton, N., Kennedy, A., Russell, G.: Compiling standard ML to java bytecodes. SIGPLAN Not. 34(1), 129–140 (1998)
13. Reppy, J.: Concurrent ML: Design, application and semantics. In: Lauer, P.E. (ed.) Functional Programming, Concurrency, Simulation and Automated Reasoning. LNCS, vol. 693, pp. 165–198. Springer, Heidelberg (1993)
14. Ringenburg, M.F., Grossman, D.: AtomCaml: first-class atomicity via rollback. SIGPLAN Not. 40(9), 92–104 (2005)

The Functional Programming Language R and the Paradigm of Dynamic Scientific Programming (Position Paper)

Baltasar Trancón y Widemann[1], Carl Friedrich Bolz[2], and Clemens Grelck[3]

[1] Ecological Modelling and Computer Science, Universität Bayreuth, Germany
Baltasar.Trancon@uni-bayreuth.de
[2] Software Engineering and Programming Languages, Heinrich-Heine-Universität Düsseldorf, Germany
[3] Computer Systems Architecture, Universiteit van Amsterdam, Netherlands

Abstract. R is an environment and functional programming language for statistical data analysis and visualization. Largely unknown to the functional programming community, it is popular and influential in many empirical sciences. Due to its integrated combination of dynamic and reflective scripting on one hand, and array-based numerical computation on the other, R poses unique and challenging implementation problems, at odds with the conservative language technology employed by its developers. We discuss the background of R in historical context, highlight some of the more problematic language features, and discuss the potential for the effective use of state-of-the-art language technology in a future, safe and efficient implementation.

1 Introduction

It seems ironic that the functional programming language that is likely to be the only one of its kind for decades featured in the New York Times [26], to account for the most computing hours in scientific labs all over the world, and to have the largest group of academic users outside computer science proper, has a relationship to the functional programming community at large that is best summarized as mutual ignorance. The R system and language [23] is a dynamic functional environment for data analysis and visualization and is widely considered the de-facto standard platform for development of statistical algorithms.

In this paper, we argue that R is a showcase application where functional programming can really shine and be brought to the awareness of a vast scientific and industrial community. We explain why the use of R has outgrown its design, and why more intense contact with the functional programming community could be beneficial to guide future development (Sections 2 and 3). We outline which issues in the implementation of R defeat traditional compilation and optimization schemes (Section 4), and how modern language technology could be used in novel and interesting ways to tame the beast (Section 5).

2 A Brief History of R

The R system is a free open-source reimplementation of the S system, with minor changes in the language. S was conceived from the late 1970s on by the statistician Chambers [10–13] from Bell Labs and won the 1998 ACM Software System Award [28]. The design of the S system can be understood by analogy to two other systems that were created at approximately the same time: the Emacs text editor and the TeX typesetting system.

The three systems have a variety of properties in common: They provide a framework for automation of complex tasks on top of some basic functionality; text file editing in the case of Emacs, typesetting in the case of TeX, and statistical algorithms given as FORTRAN routines in the case of S. The chosen method of automation is explicit high-level programming rather than pre-packaged scripts collected in menus of a graphical user interface, the dominant theme of rival products such as early WYSIWYG word processors, and statistical packages such as SAS [2] and SPSS [21]. The conscious choice for a programming approach to extensibility has been made in each case because the authors foresaw the active role of users in the development process. Consequently, self-documentation and packaging play major roles in the coding styles.

The respective programming languages S/R, TeX and ELisp are more or less declarative, extremely dynamic and mildly domain-specific, and emphasize user-friendliness in terms of safe memory management and robust error handling. Software engineering concerns, in particular provisions for programming in the large such as formalized interface descriptions and means for encapsulation, are not reflected in the basic designs. The disregard for software engineering by the designers does not imply that the systems are ill-designed. To the contrary, the quickness and ease with which they could be employed for practical tasks has contributed significantly to their broad success. But, because of that success, nowadays all of the systems have by far exceeded the lifetime, variety of platforms, number of contributors and complexity of programming layers their inventors could possibly have foreseen, with literally thousands of contributed packages available from central repositories. It is therefore no surprise that issues of versioning, scalability, packaging and feature interaction have become a continual nuisance to the developer communities.

S ran on the obscure Honeywell GCOS platform and on top of FORTRAN routines at first, but major technologies such as Unix and C had been adopted before development activity ceased around 1990. R is a fairly faithful reimplementation of the C flavour of S, begun in 1993 by Ihaka and Gentleman from the University of Auckland, adding aspects of the GNU philosophy to the system, most notably "copyleft" licensing and an open-source community-driven development process. It is administered by the non-profit R Foundation. From the theoretical point of view, the only significant change in the language is the switch from dynamic to lexical scoping of variables [15], although the concept is somewhat compromised (see Section 4.2 below). While R was the "illegitimate" rival of the officially licensed commercial S offspring S-PLUS at first, it has been endorsed by Chambers by his joining the development core team.

3 Using R

The basic R system supports both interaction via a command-line interpreter loop, and offline processing via numerous batch commands and options. Diverse window-based user interfaces and integrated environments for all major platforms exist. Graphical output in high quality and a variety of formats, both interactively and file-based, is supported by basic libraries. The R interpreter can also be embedded in applications written in many mainstream languages.

The functionality of R comprises a base library of statistical and I/O code, and a huge user-contributed repository [25] of currently about 5 000 packages for virtually all conceivable sorts of data analysis. R packages typically come with extensive documentation of function signatures and usage examples. Documentation quality varies; especially structured data objects returned by complex analyses (containing, for instance, the parameters, fitted values, residuals and goodness-of-fit scores of a statistical model) are often under-specified and require a fair deal of trial-and-error and reverse-engineering of encodings to be practically usable. Not one but two concepts of object-orientation exist in R to address these issues, but they are used inconsistently, and the expressiveness of the type system for object attributes is rather poor: For instance, the only homogeneous aggregate types are arrays, of a fixed small set of base types, with unspecified dimension; lists are always heterogeneous, while both primitive scalar types and proper records are nonexistent.

The current state and interaction history of a running R system can be stored and retrieved, in order to make sessions persistent and accountable. R code units come in two flavours: *scripts* containing arbitrary sequences of commands, and *packages* containing definitions in a certain namespace. Note that there is no special declaration level of the R language: a definition is merely the assignment of an anonymous function expression to a variable. Consequently, local definitions, let-bindings and function literals have the same expressive power as global definitions.

The function abstraction mechanism of R is very powerful indeed, featuring named and positional parameters, lexical scoping, variadic functions, implicit laziness, and mutually recursive default values for every lambda term (see Section 4.3 below). While this mechanism is adequate for complex and highly customizable interface functions of packages, the interpretative overhead for a massively recursive programming style is prohibitively high, easily reaching three orders of magnitude relative to efficient compiled code. Functional programming idioms in R include a limited range of higher-order functions, most notably a family of *map/reduce* operations on multi-dimensional arrays, unhelpfully all called **apply** with an optional initial consonant. Additionally there is a tendency to avoid FORTRAN-style loops in favour of point-free operations (see Section 4.4 below). Besides concerns for elegance, the main reason is that the R interpreter has no special notion of index variables, increment operations and single-point array access. Consequently, the interpretative overhead of treating them as dynamic degenerate cases of high-level operations is enormous with respect to the use of dedicated machine registers and instructions in compiled code.

From the academic viewpoint, R is a valuable addition to any curriculum in empirical sciences. The language is easy and fun to teach and to learn, and particularly suitable for students without a computer science background. Even though advanced functional programming skills are rare among R programmers, the fundamentals are well supported by the language and the corpus of example programs. Everything from ad-hoc data exploration and manipulation to publication-quality statistic analyses and diagrams can be achieved in the form of a short R script, often incrementally derived from freely available code. R scripts are also a useful, executable complement to term papers and theses in data-oriented courses. Program source code and results can be spliced into documents by means of the Knuth-style literate programming tool `SWeave`.

4 Under the Hood of R

The R system is a classical command-line interpreter with a core written in C. A major part of the higher-level functionality is written in the R language itself. Code is represented for interpretation and meta-programming purposes as an abstract syntax tree. An experimental bytecode format is supported since version 2.13, released in 2011. Native code is not generated, but foreign function interfaces for various languages such as FORTRAN, C and C++ exist. Many computationally expensive tasks are currently solved using these, because pure R is known to be quite slow and unsuited for tight loops or deep recursion.

Memory management is automatic by a generational garbage collector with additional reference counting for arrays, enabling the dynamic scheduling of destructive updates that is absolutely necessary for efficient, referentially transparent array programming. The system is extremely reflective: code objects, environments and the call stack can be inspected and manipulated freely at runtime. Persistence is a major topic: array data in various formats, command histories and complete snapshots of system state can be read and written. Graphical output, both online and in diverse file formats, is supported natively as well.

The type system is completely dynamic and rather complicated due to the multitude of historically accumulated, incongruent layers. Some type information, such as syntactic categories and array types, is encoded as magic numbers in memory cell headers. Array base types include three-valued logicals, integers, real and complex floating point numbers and strings (but not single characters). Two kinds of generic lists, with array- and pair-based structure, respectively, exist. Other datatypes are characterized by particular values of predefined metadata attributes, most notably **class**: two generic rivalling object/class mechanisms, domain-specific types such as *factors* (arrays of finite enumerated base type, optionally ordered), *data frames* (rectangular tables with heterogeneous columns) and *time series*, etc. The type system is further confounded by a collection of four dynamic type identification functions, namely **typeof**, **mode**, **storage.mode** and **class** with redundant but subtly different results.

Atomic data such as a single integer or truth value are not supported directly, but only in boxed form as singleton arrays; even in places where nothing but a single value makes sense, such as if-then-else conditions. Despite the ubiquity of function parameters that are expected to contain a single number or truth value switch, neither documentation nor semantics are consistent throughout the R base system: Attempts to pass vectors of length other than unity as such parameters variously cause the additional elements to be ignored silently, or one of a variety of warnings and errors to be raised.

4.1 Fundamental Pragmatism

R inherits from S the fundamental design focus on immediate practical domain-specific usability. Chambers describes the position retrospectively [9]:

> *There was also interest in different approaches or theories of computing, and much more so in later versions. However, there seemed always to be an unquestioned assumption that the essential criterion was a system that people would use and in particular one that provided the techniques considered essential. Much of the early discussion was therefore about which techniques we most needed to supply, and how to do it.* [...]
>
> *From a general computing view, the philosophy tries to combine aspects of functional and object-oriented (i.e., method-centered) approaches. But as in previous stages of the evolution of S, adherence to a formal approach tended to be compromised when it conflicted with what we saw users as needing.*

A formally trained programming language expert might contend that mere interest in theory is not nearly enough to guarantee any benefits from novel language aspects, and that the pragmatic blessing may well become a semantic curse, when interferences of compromised aspects get out of control. As a first hint at the kind of problems to be expected, consider the following quote from the official R language definition [23], emphasis added:

> *Whether attributes **should** be copied when an object is altered is a **complex** area, but there are some **general** rules* [...]*: Scalar functions (those which operate element-by-element on a vector and whose output is **similar** to the input) **should** preserve attributes (except **perhaps class**). Binary operations **normally** copy **most** attributes from the longer argument (and if they are of the same length from both, **preferring** the values on the first). Here 'most' means all except the **names**, **dim** and **dimnames** which are set **appropriately** by the code for the operator.* [...]

We have highlighted words that are unexpected in a language definition, which is the definitive source of semantics after all.

4.2 Variable Scoping and Evaluation Strategy

Somewhat bizarrely, R's data model is *almost* purely functional: even large arrays are persistent, and efficient updates rely on reference-counting to detect singleton references and switch to destructive updates dynamically and transparently. The only mutable data structures, however, are those that have the greatest impact on semantics, namely *environments*. Variable bindings and even references to parent environments can be overwritten ad libitum, and bizarre applications have been found and posted on R mailing lists. Thus variable scoping in R has, possibly uniquely, the properties of being *lexical* and generally *lazy* but *late* and *not referentially transparent*. The following example illustrates the peculiar effects.

```
foo ← function(x = y + 1) {y ← 2; x}
```

This statement binds the variable foo to a function of a formal parameter x with a default expression. The default, to be used whenever no actual parameter is given, is a lazy promise or closure, to be evaluated in the environment of the function body. The function body binds the variable y, then returns x. Evaluating the pair of statements

```
y ← 1; c(foo(), y)
```

in the same environment where foo has been defined, assuming **c** has its predefined meaning as the free array constructor, yields the array $(3, 1)$. The assignment to y in the function body is local and has no effect on the outer binding, but shadows it before the promise bound to x is forced. If the function body is changed to {x; y ← 2; x}, then the result becomes $(2, 1)$, because the promise is forced earlier and the reference to y falls through to the outer binding.

In theory, the mutable environments of R can make static analysis arbitrarily hard. The fact has been realized by R developers, and used to rationalize poor performance under the motto "too flexible to be fast" [22]. But fortunately, and quite understandably, destructive updates to environments are used only in very controlled ways in practice. The apparently dominant usage pattern in typical, reasonable user code is repeated assignment to the same local variable, as in loop counter increments or array updates. We propose that a clever combination of static analysis and dynamic guards should be able to wring enough meaning from variable bindings to improve both performance and safety significantly.

The following two example topics illustrate costly dynamic problems faced by an execution engine for R. Cheaper, wholly or partly static solutions rely on estimates of the call graph, and hence on some knowledge about the bindings of *function* variables, since there are no static function calls in R.

4.3 Function Call Rules

The rules depicted in Fig. 1 are a succinct paraphrase of the specification of function application in the R language definition. Since the details are quite complex, the rules depicted in Fig. 2 summarize the well-formed parametrizations from the caller's perspective.

The parameter-matching steps 2–9 are required for all function calls, except when the function expression evaluates to either of two kinds of builtins: for *primitive* functions the parameters are evaluated eagerly, sidestepping the promise mechanism, and passed in textual order; whereas for *special forms* the parameters are passed unevaluated in textual order.

1. In an expression of the form A(B) the function part A is evaluated first.
   - It is an error if the result is neither a builtin nor a lambda abstraction.
2. From B a list of **actual parameters** is formed: pairs of *name* literal (optional, followed by = if present) and *value* expression (optional).
3. From the function head a list of **formal parameters** is formed: pairs of *name* literal (mandatory) and *default* expression (optional, preceded by = if present).
4. Named actual parameters are matched with their eponymous formal counterparts.
5. Remaining named actual parameters are matched with formal parameters if the name extends uniquely.
   - It is an error if an actual name is not exact and has no unique formal extension.
6. Unnamed actual parameters are matched to yet unmatched formal parameters in textual order.
7. Valueless formal parameters (unmatched or matched by an actual parameter without value) are matched with their defaults.
   - Valueless formal parameters without defaults are matched implicitly with *missing* value expressions.
8. Remaining actual parameters are matched with the formal catch-all parameter ‘ ... ’, which may occur at any position and declares the function variadic.
9. An environment with the formal parameters bound to *promises* of the matching expressions is created.
   - Its parent is the lexical environment of the function definition.
   - Missing values are implemented as promises of an error.
10. The function body is evaluated in the environment.
    - Evaluating a reference to a formal parameters forces the matching promise, except for a few special primitive operations (**substitute**).
    - Assignments within the function body modify the environment.
11. The environment is discarded.
12. The value of the last body statement becomes the function result.

Fig. 1. Function evaluation, operational rules

†1. A function call is well-formed if each formal parameter has a corresponding actual parameter, explicitly named and in the same textual order.

†2. From a call well-formed by †1, an actual parameter name may be omitted, except if its formal counterpart is declared after ‘ ... ’.

†3. From a call well-formed by †1–2, a pair of adjacent actual parameters may be transposed, if one of the pair is named.

†4. From a call well-formed by †1–4,
   (a) an actual parameter name may be shortened to a unique prefix,
   (b) an actual parameter value may be omitted, if the formal parameter has a default, is not used, or explicitly allows omission,
   (c) an actual parameter may be omitted altogether, if the formal parameter has a default and no unnamed actual parameters follow,
   (d) a named actual parameter may be added at any position, if the name does not match any formal parameter, not even as a prefix, and the function is variadic,
   (e) an unnamed actual parameter may be added at any position, if all formal parameters are matched and the function is variadic.

Fig. 2. Function parameter matching, well-formedness rules

With the information readily available to the R interpreter, the function to be called is predictable only in rare cases, namely for function literals (explicitly named primitive or lambda term). Function variables, on the other hand, can be bound to functions of any kind and signature. Hence matching of parameters and even the well-formedness of a call are conceptually dynamic problems.

Formal parameter names in a lambda abstraction double in the roles of local variables and actual parameter keywords. Since the caller is entirely free to choose whether to address a parameter by position or by name, alpha equivalence does not hold for any R function expressions.

Missing parameter values (no default and no matching actual parameter or value omitted) are implemented as promises that raise an error when, and only when, forced. Hence it is not generally an error to call a function with too few parameter values, but only if the current function call attempts to use the parameter in a forcing way. Non-forcing uses such as **substitute** (see below) or the **missing** check do not raise errors.

The operation **substitute** is part of the R reflection toolkit. It is able to extract the actual syntax tree and environment from a promise. It is used in some R analyses to record queries in the result objects, and in visualizations to generate titles and axis labels automatically. Apart from these apparently innocuous uses, however, it has the semantically unpleasant property of violating the Church–Rosser principle: reduction of function arguments does not commute with their substitution into the function body. As a consequence, all program optimization by expansion or reduction, most notably common subexpression elimination and partial evaluation, respectively, in nested expressions (every operator is a function call too) becomes unsound and requires nontrivial safeguards in the presence of reflection.

4.4 High-Level Array Operations

The R language, in marked contrast to low-level languages such as C and FORTRAN, and more stringently than its more imperative competitor MATLAB, supports and encourages a point-free style of array programming. Again, the design is extremely pragmatic. Rather than providing an explicit, semantically well-understood basis of higher-order operations such as *map* and *reduce*, or even APL hieroglyphs, individual standard operations are *vectorized*: their global behaviour on arrays is defined in terms of local rules for individual elements, in an ad-hoc fashion governed by the most frequent usage patterns. Hence the resulting coding style is often operationally imprecise, but it can be read and written very effectively and with little room for nontrivial errors.

The following code fragment, excerpted from a package developed by one of the authors, illustrates some of the typical techniques.

```
x ← x[x != 0L]
f ← function (p) −sum(p * log(p, base = 2))
f(x / sum(x))
```

Here the binary entropy of an integer array x, assumed to contain absolute frequencies of a population of items, is computed.

The first line removes zero entries from the array: x is compared with an array containing a single (integer) zero. Like most binary operations, inequality is vectorized to act simultaneously pointwise on its arguments (*zip* in standard functional programming jargon). The arguments do not have the same length. This causes the shorter, second argument to be *recycled*, resulting effectively in an array of zeroes of the same length as x. The result of the comparison is an array of just as many truth values, with TRUE in positions where x is nonzero. Indexing x with this array is vectorized to act as a *filter*, retaining only the elements flagged as TRUE.

The second line binds the familiar entropy formula to the variable f, except that the probability distribution p need not subscripted with a loop variable, by virtue of vectorization.

The third line scales down from absolute to relative frequencies. The division operator is again vectorized to *zip* its arguments, where the latter is a single integer and recycled accordingly. Ordinary division also implies coercion from integers to floating-point numbers. The function bound to f is applied to the resulting array. It inherits vectorization from its constituent operations: Logarithm acts as *map* on its first argument, multiplication as *zip*. Unary minus acts trivially vectorized, negating the single element of the array resulting from **sum**.

Even though point-free style is used to great effect for the human reader and writer, its potential for optimized execution is not currently leveraged in R. Several intermediate arrays are created by this expression, namely as the results of the operations !=, [ ], **log**, $*$ and /, respectively, as well as degenerate arrays that box a single number each, namely the results of the literal 0L and the operations **sum** (bis) and $-$. Neither loop-fusion techniques that would reduce the amount of intermediate data, nor parallelization of *map*- or *zip*-vectorized operations can be applied in the default R system, because each of the involved operations could be redefined dynamically, altering the algebraic and vectorization properties that underly such optimizations.

5 Technological Suggestions

We suggest a three-pronged strategy to make the execution of R programs more efficient. Firstly, standard optimization techniques require a lot of information that is not declared explicitly in R and hence needs to be inferred, preferably by static analysis. The usual suspects are: type and shape analysis for arrays to elide runtime type and bound checks and coercions, strictness analysis to support the unboxing of function parameters, and static binding analysis to enable algebraic laws and inlining of known functions.

Secondly, we address the strict isolation of low-level, statically compiled core functionality on the one hand, and inefficient, dynamically interpreted scripting on the other hand, at the granularity level of individual function bodies and with no exchange of information except function calls proper. We feel it needs to be lifted in favour of an integrated, flexible, code generator-centric approach. This probably requires a radical departure from the existing implementation, since the distinction is deeply ingrained in the employed methodologies and technologies.

No obvious candidate for a backend platform exists that supports the dynamic and number crunching aspects of R equally. But fortunately, even though their unification in the R *language* is most desirable from the user and software engineering perspectives, the separation of scripting and numerics layer in R *programs* appears feasible. Hence we propose the liberal solution to target not one but two platforms, specializing on either aspect, and to guide the choice and/or combination by user preference and analysis results.

5.1 Static Analysis and Beyond

In a language where programs are short and run typically for a long time on huge amounts of data, static analysis and transformation easily pays off. Unfortunately, virtually all ahead-of-time transformation of R code is either impossible or semantically unsafe, unless strong assumptions about the program can be made and verified: The dynamic, intransparent behaviour of environments makes prediction of variable values theoretically difficult, and foils optimizations such as constant propagation and algebraic simplifications. The powerful reflection mechanism allows the behaviour of code to depend on its literal form, and foils optimizations such as common subexpression elimination, function inlining, lambda lifting and substitutive optimizations in general, most notably specializations relying on "the trick" of partial evaluation. The lack of explicit type and shape information for arrays poses the same problems regarding loop organization and bounds checking as in other dynamic languages.

The last, type-related category of issues can also be handled with local speculative techniques such as *quickening*, even when retaining an interpreter [8], and does not necessarily call for static analysis. But the former two involve long-range dependencies that are not easily dealt with, unless it can be assumed that the problematic features are not actually used. An empirical survey [20] (see Section 6.1 below) of a corpus of R code indicates that the majority of practical program fragments is actually reasonably safe. Hence we expect that a static approximation, even if a little coarse, should be able to leverage many well-understood and effective optimization techniques. It may even be worthwhile to investigate whether potentially unsafe transformations actually break a given package, relying on R regression testing for heuristic validation.

For the sake of modularity, as well as to allow the user to stand in where static analysis fails, a general annotation format for static information in R code would be useful to communicate properties at interfaces. In this aspect, the reflective power of R can be turned to advantage: Program terms are ordinary data structures with an extensive query and manipulation interface, and arbitrary structured metadata can be attached to any data via attributes. Hence annotations can be added simply by establishing a metadata format, with no need for any invasive extensions of the language proper.

It remains to see whether users of R can be convinced to use the mechanism and add declarations to their programs. We expect that ideological arguments from semantic theory or software engineering methodology would not be well-received. But practical annotation-based tools, for instance providing diagnosis of inconsistencies, automatic instrumentation with assertions, automatic documentation of interfaces and/or automatic generation of test cases, might well gain some recognition in the community and increase awareness for the issues.

5.2 Dynamic Compilation

A recent line of techniques that have been successfully used for implementing other complex dynamic languages without too much effort are *tracing* JIT compilers [14]. Those are JIT compilers that take as their compilation unit not a function but a commonly executed code path (trace) within the program. Quite often these traces correspond to loops within the original program, ending with a jump to their beginning.

Tracing JITs have been used successfully for dynamic imperative programming languages [14], but also some first experiments for declarative programming languages have been done, for example for Prolog [7].

The traces in a tracing JIT are formed by observing the execution paths through the program as it executes after some profiling. Thus all the traces correspond to control flow paths that have been taken a few times already.

This approach has many advantages. On the one hand it makes many components of the JIT compiler much simpler to write. Both the optimizers [4] and the backends can be very simple, because they only need to deal with linear pieces of code. This also allows the optimizer to perform aggressive type specialization and optimize away the potential for dynamic behaviour that is not used in the current code path. This can lead to very efficient machine code which removes most of the overhead of dynamic typing.

Dynamic compilation and trace compilation in particular is extremely effective in optimizing the overhead of dynamic typing on the local level. However, due to the limited scope of compilation no global information can be exploited. This is why we think that a combination with a static analysis phase which feeds back some globally established properties of the program into the runtime compiler could improve the efficiency of dynamic compilation even more.

Most tracing JIT compilers have been written for one specific VM and thus for one specific language. A few projects have emerged that tried to write reusable tracing JIT compilers. This approach is called “meta-tracing”, because the tracers do not operate on the level of the program, but on the level of the interpreter for the program. Examples are the SPUR project [3] and the PyPy/RPython project [6].

RPython [1] (“Restricted Python”, there is no relationship to R) was developed for the PyPy project [24]. It is a programming language designed for implementing interpreters for dynamic programming languages. The RPython subset of Python is restricted in such a way as to make compilation of the interpreter to C possible. The interpreter written in RPython can thus be translated into a VM in C. During this process, various aspects of the final VM are woven into the interpreter. Examples for this are garbage collection and a tracing JIT compiler [6].

This weaving of low-level aspects into the final VM means that the language implementation in RPython stays independent of low-level details. The meta-tracing JIT is one such orthogonal aspect. It is woven into the final VM, guided by a few hints that the interpreter author inserts into the interpreter [5].

We plan to utilize the RPython framework for the implementation of our R system, specifically the execution of R code. We feel that a meta-approach to tracing JIT construction is the only sensible way to efficiently implement a language as complex as R efficiently as a whole. We hope to be able to integrate with this jitting implementation the results from our static analyses to make the JIT generate even better code.

5.3 High-Performance Functional Backends

It hardly requires visionary power to understand that fairly small compute-intensive numerical kernels, as shown in Section 4.4, are the kind of R code that dominates the execution times of entire programs while at the same time the performance difference between R and low-level compiled languages, say C or Fortran, is the greatest. Consequently, such definitions also provide the most attractive opportunities to speed up the execution of R code, potentially by orders of magnitude. The on-going trend in commodity hardware towards multi-core designs and the proliferation of many-core graphics accelerators in the mass market are both a blessing and a curse for languages like R: a curse as they will widen the performance gap between R and more low-level approaches; a potential blessing if R could implicitly utilise parallel computing power without the notorious hassle incurred by low-level parallel programming.

Compilation of R array kernels into efficient code, not to mention decent support for a variety of multi- and many-core architectures, is a major research and engineering challenge. Therefore, it is attractive not to go all the way from R down to C or Fortran, but to leverage a language whose design is somewhat half way between R and C as an intermediate compilation target and to leverage existing compilation technology.

Such a language is SaC (Single Assignment C, [17]). SaC is a compiled array programming language with a syntax that, as the name suggests, resembles that of C proper, but that at the same time comes with a purely functional semantics. SaC features stateless multi-dimensional arrays similar to APL or MatLab, call-by-value parameter passing for arrays, automatic memory management, etc. Such features of SaC considerably reduce the semantic gap in compiling R.

We illustrate this in Fig. 3 by means of a SaC code fragment implementing the example introduced in Section 4.4. Apart from the C-style syntax of function definitions and applications, fragments of the code are almost identical to R. Some issues are minor: for example, prior to the division in function `g` we need to explicitly convert integers into floating point numbers. Other issues are more relevant: SaC is monomorphic with respect to scalar types. A big plus on the other side is SaC's support for rank-generic programming: the type `int[*]` refers to integer arrays of any number of dimensions and any extent along these dimensions. One aspect we cannot directly mimic from the R code is the

```
double[*] log2( double[*] p)
{
  return { iv -> p[iv] == 0.0 ? 0.0 : log2(p[iv]) };
}

double f( double[*] p)
{
  return -sum( p * log2(p));
}

double g( int[*] x)
{
  return f( tod(x) / tod(sum(x)));
}
```

Fig. 3. Computing entropy in SaC

elimination of zero elements in the argument array. Apart from reducing the size of the array and, thus, reducing the computational effort of subsequent computations, the main reason is to avoid computing the logarithm of zero. We achieve at least the latter by overloading the `log2` function for arrays such that it yields zero in case, which does not affect the rest of the computation.

Despite a fairly similar programming style, the semantics of SaC code is much more stringently defined than that of R. This supports a highly optimising compiler technology that achieves competitive runtime performance for compute-intensive applications [27] and fully compiler-directed parallelisation for multi-core multi-processor systems [16] and for CUDA-enabled graphics cards [18].

6 Related Work

6.1 Evaluation of the Design of R

In a paper to appear at ECOOP 2012, Morandat et. al. [20] comprehensively analyze the properties of the R language as well, albeit from an object-oriented rather than functional perspective, reaching similar conclusion as this paper. They offer several interesting contributions, among them a formal semantics for some core parts of the language as well as a careful analysis of which features are used *in practice* by R programmers. The latter is done by collecting, running and analyzing a large corpus of R programs. One of the results of running benchmarks of R programs compared to equivalent programs written in Python is that R is significantly slower even than other dynamic programming languages like Python. Another result of the corpus analysis of R programs is that the older of the two competing R object systems is still much more widely used.

6.2 The Ra Extension

Ra [19] is an extension to the R system that interprets a proprietary bytecode format rather than R syntax trees. The bytecode is produced by a companion JIT library that specializes on arithmetic loops, using runtime type and shape information for arrays, and thus eliminating many dynamic type case distinctions, allocation and variable lookup operations. The label “JIT compiler” on the approach is not consistent with the conventional use of the term for dynamic languages, because no machine code is produced at runtime.

Unfortunately, there are severe additional caveats: Firstly, function calls and local control flow other than **for** loops are not processed in any way. Secondly, the transformation is not semantically conservative; more dynamic uses of array variables than supported by the compilation scheme, even such as resizing, will not just revert to the slow original, but raise an error. Lastly, the loop-oriented programming pattern that is sped up by Ra is deprecated in R in favour of vectorized operations anyway. In summary, despite good ideas, Ra cannot be considered a significantly more efficient implementation of R.

7 Conclusion

Scientific computing is nowadays more than just number crunching. The increasing *logical* complexity of data processing methods, as opposed to their *computational* complexity, requires other approaches than the extreme high-performance low-elegance style offered by the classics, most notoriously FORTRAN.

Scientists have reacted by adding high-level scripting layers, written in dynamic languages such as Python, to their software. A unified approach such as the one offered by R has numerous advantages:

Scientists, often without proper formal training in programming skills, are required to master only one language and one environment that apply to all levels of their data processing software. The R language is, semantic peculiarities aside, well-documented and forgiving, and encourages abstract, elegant and reusable functional style. The R environment is highly interactive and comes with extremely powerful tools for data exploration and visualization.

The functional structure of R programs makes them theoretically amenable to automatic optimization and parallelization of array processing code, thus compensating for a significant part of the efficiency loss with respect to low-level high-performance code, without sacrificing the gains in flexibility, both with respect to application parameters and hardware capabilities, and ease of development and maintenance.

On the practical side, the existing R system has deficiencies with respect to diagnostics and performance of pure R code. The problem is currently being side-stepped by R developers by recommending the use of foreign function interfaces for performance-critical code fragments, thus compromising many benefits of the high-level approach, most notably datatype dynamicity, platform independence, memory safety and interactive responsiveness.

The concentration of the dynamic and numeric perspectives in a single language is a unique and challenging situation. We believe that the traditional language technology underlying the current R implementation is fundamentally ill-suited for the problem, but that modern technologies developed for other dynamic languages on the one hand, and for other functional languages on the other hand, can be used to great effect. The seemingly active hostility of R towards optimization should be understood as a motivating challenge to compiler constructors, and may well serve as a realistic test case to evaluate the techniques applied to it. R has already gained considerable recognition from academic users for being *usable* and *flexible*. If *fast* and *safe* could be added to the list, it would make a very convincing case for functional programming.

Acknowledgments. A research proposal for an optimizing implementation of R using the techniques outlined in this article is currently under consideration for funding by the *Deutsche Forschungsgemeinschaft* under application number LE 2377/2-1.

References

[1] Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: RPython: a step towards reconciling dynamically and statically typed OO languages. In: DLS. ACM, Montreal (2007) ISBN: 978-1-59593-868-8, doi:10.1145/1297081.1297091

[2] Barr, A.J., Goodnight, J.H.: SAS, Statistical Analysis System. North Carolina State University (1971)

[3] Bebenita, M., et al.: SPUR: a trace-based JIT compiler for CIL. In: OOPSLA. ACM, Reno/Tahoe (2010) ISBN: 978-1-4503-0203-6, doi:10.1145/1869459.1869517

[4] Bolz, C.F., Cuni, A., Fijalkowski, M., Leuschel, M., Pedroni, S., Rigo, A.: Allocation removal by partial evaluation in a tracing JIT. In: Khoo, S.-C., Siek, J.G. (eds.) Partial Evaluation and Program Manipulation (PEPM 2011). ACM, Austin (2011) ISBN: 978-1-4503-0485-6

[5] Bolz, C.F., Cuni, A., Fijalkowski, M., Leuschel, M., Rigo, A., Pedroni, S.: Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages. In: ICOOOLPS. ACM, Lancaster (2011)

[6] Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: PyPy's tracing JIT compiler. In: ICOOOLPS, pp. 18–25. ACM, Genova (2009) ISBN: 978-1-60558-541-3, doi:10.1145/1565824.1565827

[7] Bolz, C.F., Leuschel, M., Schneider, D.: Towards a Jitting VM for Prolog execution. In: PPDP. ACM, Hagenberg (2010) isbn: 978-1-4503-0132-9, doi:10.1145/1836089.1836102

[8] Brunthaler, S.: Efficient interpretation using quickening. SIGPLAN Not. 45(12), 1–14 (2010) ISSN: 0362-1340, `http://doi.acm.org/10.1145/1899661.1869633`, doi:10.1145/1899661.1869633

[9] Chambers, J.M.: Stages in the Evolution of S. Bell Labs (2000), `http://cm.bell-labs.com/cm/ms/departmen`

[10] Chambers, J.M., Becker, R.A.: An Interactive Environment for Data Analysis and Graphics. Wadsworth & Brooks/Cole (1984) ISBN: 0-534-03313-X

[11] Chambers, J.M., Becker, R.A.: Extending the S System. Wadsworth & Brooks/ Cole (1985) ISBN: 0-534-05016-6

[12] Chambers, J.M., Becker, R.A.: The New S Language: A Programming Environment for Data Analysis and Graphics. Wadsworth & Brooks/Cole (1988) isbn: 0-534-09192-X

[13] Chambers, J.M., Hastie, T.: Statistical Models in S. Wadsworth & Brooks/- Cole (1991) ISBN: 0-412-05291-1

[14] Gal, A., et al.: Trace-based just-in-time type specialization for dynamic languages. In: PLDI 2009. ACM ID: 1542528. ACM, New York (2009) ISBN: 978-1-60558-392-1, doi:10.1145/1542476.1542528

[15] Gentleman, R., Ihaka, R.: Lexical Scope and Statistical Computing. Journal of Computational and Graphical Statistics 9, 491–508 (2000)

[16] Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming 15(3), 353–401 (2005)

[17] Grelck, C., Scholz, S.-B.: SAC: A Functional Array Language for Efficient Multi-threaded Execution. International Journal of Parallel Programming 34(4), 383–427 (2006)

[18] Guo, J., Thiyagalingam, J., Scholz, S.-B.: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: Annual Symposium on Principles of Programming Languages, 6th Workshop on Declarative Aspects of Multicore Programming (DAMP 2011), pp. 15–24. ACM Press, Austin (2011)

[19] Milborrow, S.: The Ra Extension to R (2011), `http://www.milbo.users.sonic.net/ra/`

[20] Morandat, F., Hill, B., Osvald, L., Vitek, J.: Evaluating the Design of the R Language. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 104–131. Springer, Heidelberg (2012)

[21] Nie, N., Bent, D.H., Hull, C.H.: SPSS: statistical package for the social sciences. McGraw–Hill (1970) ISBN: 0-070-46530-4

[22] R-bloggers. You can scrap it and write something better but let me keep R (2011), `http://www.r-bloggers.com/you-can-scrap-it-and-write-something-better-but-let-me-keep-r/` (visited on August 21, 2011).

[23] Language Definition, R.: R Development Core Team (2010) ISBN: 3-900051-13-5

[24] Rigo, A., Pedroni, S.: PyPy's approach to virtual machine construction. In: Portland, D. (ed.) DLS. ACM, Portland (2006) ISBN: 1-59593-491-X, doi:10.1145/1176617.1176753

[25] The Comprehensive R Archive Network, `http://cran.r-project.org`

[26] Vance, A.: Data Analysts Captivated by R's Power. In: New York Times. Business Computing (January 9, 2009)

[27] Wieser, V., Grelck, C., Haslinger, P., Guo, J., et al.: Combining High Productivity and High Performance in Image Processing Using Single Assignment C on Multicore CPUs and Many-core GPUs. J. Electronic Imaging (to appear)

[28] Wilson, A.: ACM honors Dr. John M. Chambers of Bell Labs with the 1998 ACM Software System Award for creating 'S System' software. In: ACM Press Release (March 1999)

Lingua Franca of Functional Programming (FP)

Peter Kourzanov and Henk Sips

NXP Semiconductors Research, TU Delft

Abstract. Scheme is a minimalist language that brings its ancestor, LISP, on par with modern languages such as *ML and Haskell. Although it has very different tools in its repertoire (e.g., *homoiconicity*), many of the same techniques applied in other languages are also quite usable in Scheme. Some of them (e.g., *pattern-matching*, *monadic* and *multi-stage* programming, a notation for *laziness*) tend to be associated with other languages (primarily: Haskell and OCaml), and hence many programmers liking these constructs are drawn away from Scheme.

The concrete reason for this in FP community is the perceived verbosity of Scheme, while others (in particular, from the embedded community) have troubles adopting the more difficult concepts of FP in general. It is clear that there are approaches developed with FP that do help for productivity in hardware and software design. Transfer of this knowledge, however, is made difficult by an intellectual gap between the communities. In this paper we attempt to reconcile these two points of view by showing that (1) Scheme's minimalistic but *flexible* approach can bring concrete benefits to the embedded community, and (2) none of these "advanced" features is particularly hard to replicate in a language such as Scheme. If one is willing to leverage Scheme's *hygienic macros*, all of these features can be implemented separately as libraries and freely used together. No advanced compiler or interpreter modifications to implement such extensions are required for creating embedded DSLs in it.

To really see what Scheme (and homoiconicity) can bring to the domain of automated program construction, and to apply these ideas in practice to Software-Defined Radio (SDR) stacks, we have re-implemented a multi-staged, monadic FFT generator. In this paper we supplement it with a straightforward implementation of the C code-generation and Graphviz visualization back-ends and argue for a homoiconic, pure and total language to combine the best features of Scheme, Haskell and *ML.

1 Introduction

I got an extremely large error ($\sim$ 5000 *lines) when loading "OCamlTutorial.hs". When I've parsed through it, I'll post back.*
Aditya Siram on Haskell-cafe

Functional programming is on the rise. More and more developers coming from C{,++,♯}, Python/Ruby/Perl, as well as Javascript trades are becoming aware of techniques originally developed for languages such as LISP, Scheme, Haskell, and the languages from the *ML family (SML, OCaml and F♯).

The nature of how languages come into being influences their development. For instance, C and C++ were conceived driven by engineering principles, while the whole family of functional languages (with perhaps the exception of Erlang) is the result of a desire to explore the nature of computation & logic. For quite awhile, this has largely contributed to the intellectual gap between the engineering and the academic communities, where on one side is the deep knowledge of the application domain and on the other side is the inner knowledge of how to best structure the development of software and reason about its behaviour. With the complexities of the software-defined systems increasing rapidly, especially for those found in SDR domain, it becomes more and more important to automate its creation process and aim for "correct-by-construction" implementations.

For both Haskell and Meta Language (ML), typing programs statically and understanding Hindley-Milner (HM) inference with *let-polymorphism*, as well as the associated *value* and *monomorphism* restrictions (letting alone all the type- and kind extensions proposed for these languages), presents significant hurdles to developers found in the embedded industry. One of the more advanced features of C++ (*templates*) is also shunned in this community because of the complexities in understanding e.g., the error messages [2]. Although the benefits of both are well-appreciated in some circles, *practical* usage proves otherwise. With no resolution to static-types vs. dynamic-types dichotomy, and no inclusion of *concepts* [26] in C++ in sight [25], one is forced to make compromises.

The Revised^5 Report on the Algorithmic Language Scheme (R5RS) [1] offers a good venue for making compromises. Being a strongly-typed and a high-order language, it enjoys some of the same face values as Haskell and ML: the ability to describe the behaviour of complex systems using rather simple building blocks: *closures*, *tuples*, and *scoping*. Being syntactically malleable, it allows easy embedding of a Domain-Specific Language (DSL) via safe, hygienic `syntax-rules`, as is elucidated by e.g., [15]. Other macro mechanisms such as low-level macros, reader-macros (and, most notably, `syntax-case`) trade-off safety for expressive power. Application of syntactic extensions facilitates separation of concerns, where a domain expert designs the embedded DSL for use by application experts.

Such a DSL must address concerns of both the domain experts (i.e., embedded architects) and of the application experts (i.e., embedded developers) equally well. On one hand, it must provide domain constructs that allow concise specification and implementation of the algorithms, in a way that is understandable and straightforward for application experts. On the other hand, it must be possible to implement both platform-specific and generic optimizations for concrete implementations. Finally, a DSL must ensure "correct-by-construction" way of working by enforcing the boundary between the two layers.

The paper is organised as follows. In Subsection 1.2 (to remind the reader what the distinguishing features of Scheme are) we look at an archetypal technique of functional programming: expressing recursion using simpler basic building blocks. We avoid a pitfall in using fixed-point combinators (the need for currying) by slightly modifying the classical definition of Y and conclude by showing a natural extension towards *memoization*. In Sections 2 and 3 we review the syntactic extensions that we implemented on top of Scheme (and verified to work with the Bigloo system [22] and Stalin [24]) in order to translate the Fast Fourier Transform (FFT) generator to Scheme. In Section 4 we show our implementation of the C code generator and illustrate the Graphviz back-end operating on the result of the FFT generator in Decimation in Time (DIT) and Decimation in Frequency (DIF) modes. We conclude and review future work in Section 5.

1.1 Related Work

The idea of representing computation by monads has been in the wild for quite awhile [18,30]. Especially interesting and rewarding are the uses of monads in automated program construction, e.g., for combinatorial circuits [14], dynamic programming [27], high-performance computing [6] and linear algebra [5]. In [21], a Scala approach to automate construction of e.g., FFT networks has been described. All of this work heavily relies either on unconventional infrastructure (e.g., `traits`) or on extensive modifications to a compiler (e.g., Meta-OCaml).

1.2 Fixed-Point Combinators in Scheme

It is well known that the simply-typed λ-calculus can not express the ω, an essential ingredient of fixed-point combinators (see the first line in the body of Fig. 1).[1] Although statically-typed encodings using iso-recursive types also exist (e.g., see [17]), they are much less clear than the Scheme's version.

```
(define (Y step)
 ((λ (x) (x x)) ; ω combinator
  (λ (g) (λ data ; varargs
    (apply step (g g) data)))
  ))
```

Fig. 1. X/Y combinator in Scheme

How a fixed-point combinator works is also well-known: see e.g., [12]. Effectively, the ω serves as a "reflector" that initializes the binding of `g` to refer to the function of `g` itself. This binding is threaded throughout the recursion via self-application `(g g)`. In a Call-by-Value (CBV) language such as Scheme, this self-application must be η-expanded, so that `(g g)` and `step` functions are applied only when the thunk (result of the previous self-application) receives an argument (the `data` list).

It is lesser known that applying a fixed-point combinator in practice does *not* require the use of *currying* in the `step` function. Because the length of the `data` that the thunk expects is *variable*, and because the application uses Scheme's built-in `apply` procedure, any number of arguments (beyond the required "self" parameter) can be given to the `step` function. The higher-orderness of this fixed-point combinator can be rectified this way as well (see Fig. 2, lines 4-6). For example, the generalised FIB function (from Fig. 3) can be called using Y simply by prefixing the call and supplying the arguments as they would have been supplied normally (i.e., *uncurried*), as in `(Y fib 1 200)`, for example.

[1] keywords/free bindings are typeset in **bold**, top-level bindings in SMALL CAPS, monadic primitives sans-serif, Bigloo primitives underlined, and comments *slanted*.

Note that the η-expansion is floated: `(λ data ...)` contains `(apply step ...)` as a sub-expression, not vice-versa. While being equivalent to the traditional Y combinator, this version does allow the implementor of the combinator to obtain the arguments to the "next" recursive call before calling the user-supplied `step` function. In lines 8-11 of Fig. 2, this is crucially used to implement the "benign" effect of memoization, completely transparently to the user.

```
(define (Y step . args)                                 1
 (let ([tab '()])                                       2
  ((λ (x)                  ; ω combinator               3
    (let ([r (x x)])       ; thunk                      4
     (if [null? args] r    ; if no args                 5
      (apply r args))))    ; args given                 6
   (λ (g) (λ data          ; self => varargs            7
    (cond ([assoc data tab] => cdr)                     8
     (else (let ([res (apply step (g g) data)])         9
      (set! tab `((,data . ,res) . ,tab))               10
      res)))))                                          11
   )))                                                  12
```

Fig. 2. Streamlined X/Y combinator, with memoization

2 Pattern Matching

```
(def FIB (fn _ s [_ < 2] ⇒ s
  | rec s n ⇒ (+ (rec s (− n 2))
                 (rec s (− n 1)))
)) ; a "relational" pattern
```

Fig. 3. Generalised Fibonacci

Scheme's minimalism and the resulting simplicity in the language design of course comes at a price: the user has to bring applications that are written directly using the core language to the same conceptual level as required by Scheme's minimalistic approach. As a result, the examples of the previous section not only show the power of Scheme, but also its verbosity. This is often aggravated by the need to represent complex data-structures as pairs - which in Scheme is a conflation of tuples & lists. Lack of built-in syntactic sugar often leads to abundance of parentheses, which is a frequently cited complaint about languages rooted in List Processing (LISP).

Although this problem has been recognised for quite awhile [31,19], many implementations still include pattern-matching as a *non-standard* extension. And of course, each implementation does it in a different way.[2] Although there were attempts to implement a portable matcher [11,3,23,28], none did address the compatibility to other languages.

This defeats the purpose of having a standard as well as having a large number of different (competing) implementations. In this paper, we propose a particular style of pattern-matching that - on purpose - brings Scheme closer to other

[2] So for example, the Wright/PLT matcher has a non-intuitive predicate & structure case syntax, the Queinnec/Bigloo matcher [20] has verbose variable binder syntax.

languages in this respect. In the examples starting from this section, we shall use the following notational conventions:

def : immutable global variable binding (i.e., a `define` sans `set!` mutator)

cases : backwards-compatible pattern-matching extension of Scheme's `case`.

fn/fn', fun/fun' : anonymous function constructor, desugaring into λ and `cases` for argument list destructuring. `fun` and `fun'` variants return `#F` on match failure. `fn'` and `fun'` implicitly add *quasiquote* around each pattern case.

lets : *destructuring* binding form, also utilising `cases` as the back-end.

$\Rightarrow$: delineates the pattern(s) on the Left-Hand Side (LHS) and the expression(s) on the Right-Hand Side (RHS). The expressions are wrapped by an implicit sequencing form (`begin`), which returns the value of the last expression.

$\rightarrow$: delineates the pattern(s) on the LHS and the **data** (a single expression of static shape) on the RHS. The data can be *quasi-static*, as the RHS is implicitly quasiquoted (see lines 3-4 in Fig. 12 for an example).

In addition to **n+k** patterns (line 1 of Fig. 4), we support **relational** patterns, which compare favorably to **predicate** patterns (cf. line 2 to 3 of Fig. 4) and also make some uses of `when`-guards redundant (e.g., line 1 of Fig. 3). **Active** patterns (as introduced in [7] and applied in F♯) are utilized in our complex number library (which is not shown in this paper because of space limitations).

```
1 (def A (fn 0 [n+1-1] ⇒ n+1 ;"n+k" pattern
2   [m > 0][n > 0]⇒(A (− m 1) (A m (− n 1)))
3   [positive? value −: m] 0 ⇒ (A (− m 1) 1)
4 )) ; "predicate" pattern above
```

Fig. 4. Ackermann's function

The primary novelty of our pattern-matcher is not in its set of features, many of which have appeared in prior art (possibly, in another form), but in the way they are applied together with monadic and lazy computations. In fact, we deliver the matcher as a single portable package based on `syntax-rules` supporting several Scheme implementations, and the following features:

wildcards, literals, pairs/lists, vectors are standard for all pattern-matchers.

quasiquotation patterns are useful for match-by-example programming, where the pattern is quasi-static. We support any number of quasiquotation levels.

static & polymorphic records provide the ability to match records *syntactically*, using the following syntax: `(tag : flds ...)` On the RHS, this syntax expands to code that is compliant to SRFI-9: `(tag flds ...)`.[3]

predicate patterns generalise records to *procedural* interface, where a predicate guards a list of functional sub-patterns using either `fun-:pat` or `pat:= fun` syntax. Active patterns bundle a set of record-types via transformers.

lazy patterns concealing the `force` operator[4] behind the pattern-matching interface. This complements to the {}-notation for `delay` and `lazy` forms.

[3] Scheme Request for Implementation (SRFI) extend R5RS to provide records etc.

[4] A common complaint of lazy programming in Scheme or ML.

ellipsis (...) allows matching against repeating sub-patterns, binding corresponding pattern variables to lists of sub-matches.

catamorphisms & views is a feature derived from matcher found in [11]. It allows the matcher to be recursive (the `[~(pat)]` *catamorphism* syntax) or call out to another matching procedure (the `[~ fun -> pat]` *view* syntax).

non-linear patterns allow to pattern-bind the same name multiple times. The consistency is checked with Scheme's standard `equal?` procedure by default. Other ways of checking equality are supported too via predicates.

conjunctive, disjunctive and negation patterns generalise the **as**-patterns found in Haskell and some extensions of ML's pattern matcher, e.g., [32].

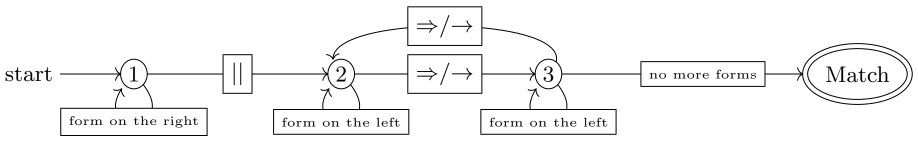


Fig. 5. Finite State Machine (FSM) for the `cases` form

The `cases` form is implemented as a R5RS macro that functions as a Term-Rewriting System (TRS). Although `syntax-rules` macros tend to be entirely straightforward (see for example, Fig. 9), we do not include the full implementation of `cases` because of space limitations. To highlight the principle of operation, we include only a rough FSM depicted in Fig. 5. Matching proceeds in stages: (1) collection of the arguments to be matched and their packaging in a list (left-to-right), (2) collection of expression(s) on the RHS (right-to-left) and (3) collection of sub-patterns and pattern alternatives on the LHS (right-to-left). The final result of this process is given to lower-level `pattern-match` macro that is no longer concerned with high-level syntax of `cases`-expressions, but with the actual handling of all types of sub-patterns. As a bonus, the `cases` form also supports SRFI-87's version of the Scheme's standard `case` form.

Note that we have attempted to design our pattern matcher to be *bidirectional* where possible. That is, we make the syntax of the pattern on the LHS resemble the syntax of the match result expression on the RHS. Because vectors, like lists, are not self-quoting in the Scheme standard [1], we ensure that both the pattern and the expression are quasiquoted. Combining auto-quoting for both pattern (`fun'`, `fn'` and `cases'` forms) and match results (→ delineation), we actually achieve bidirectionality if the pattern only uses literals, pairs/lists, vectors, (both static and polymorphic) records , non-linear, lazy patterns and quasiquotation.[5] See INTR resp. DEINTR functions from Figures 10 resp. 11.

3 Monadic Programming

Monads, as shown in [16] are *promiscuous*, and several sources already confirm their usefulness in languages other than Haskell (see [9,13]). It is easy to use

[5] "n+k" patterns and views are also possible as long as used functionals are bijections.

High-Order Function (HOF) patterns to encode monads.[6] Basic sequencing is enforced via the requirement that the *continuation* at each statement separator in the sequence expects preceding statement(s) to have produced values, and performed all effects. The "semi-colon" is invisible in Scheme, see monadic program in Fig. 10. Moreover, the Input-Output (IO) monad is the implicit default.

A known shortcoming of the *monadic* style of programming in statically typed languages other than Haskell (that is: *ML, Scala) is its verbosity [6]. To address this issue we apply a number of methods facilitated by R5RS Scheme:

```
(def LISTM (instantiate::list-monad
 (∅ '())
 (⊎ append)
 (unit list)
 (bind (fn xs fu ⇒ (apply append
     (filter-map fu xs))))
)) ; derived from MonadPlus
```

Fig. 6. List monad instance

1. Generalised comprehensions, using the Zermelo-Fraenkel (ZF) set-builder-like notation (for which we provide syntactic sugar via a reader-macro)
2. Overloading of standard operators to "lifted" versions (via R5RS macros)
3. Encapsulation of bind and unit as methods inside an object of a certain class, as shown on Fig. 6 (this is specific to Bigloo's object system, but otherwise easily replicable with other Scheme implementations)
4. Provision of an assignment syntax that is familiar to embedded developers

3.1 Comprehensions

```
(def QS (with-access::list-monad
            LISTM (bind unit ∅ ⊎)
(fn _ () ⇒ ∅
 | p (h . tl) ⇒ (⊎
   (QS p [x | x←tl where (p x h)])
   [h]
   (QS p [x | x←tl except(p x h)]))
 )))
```

Fig. 7. Monadic quicksort

Comprehensions are an important syntactic device that helps in expressing some algorithms in their purest form. For example, the QS (Quick Sort) can be implemented directly according to its specification, as seen in Fig. 7. Note that with Scheme we can rely on dynamic return-types for the `fu` functional from the List monad instance from the previous subsection. By using `filter-map` instead of a conventional `map`, predicates can simply return `#F` and computation can proceed without vacuous flattening of empty lists. Hence, we simply rewrite ≪`where (p args ...)`≫ into ≪`#T ← (p args ...)`≫, and similarly ≪`except (p args ...)`≫ into ≪`#F ← (p args ...)`≫; we let the `match-failure` procedure always return `#F`.

In conjunction with the Bigloo reader extension that converts all occurrences of `[forms ...]` into either monadic comprehensions (≪`[expr | generators ... ]`≫ variant as in the figure above) or monadic computations

[6] Just as it is easy to use *delimited continuations* [8] for the same, as they are functional representation of the control stack.

(≪`[forms ... ]`≫ variant as in the FFT figures below), this paves a way to mathematically straightforward programming with monads in Scheme.

4 Applications: FFT

In this section we bring together the techniques highlighted above and apply them to an important component of virtually all SDR stacks. Such systems are characterized by the flexibility in Media Access Control (MAC) protocols, Forward Error Correction (FEC) encodings and physical transmission parameters such as the *modulation*. Because most of the innovation in radio algorithms is based in mathematics of information theory, it is important to maintain a strong link between the SDR implementation and its algebraic specification, which is usually prototyped using Matlab. The inherent complexity of SDR can be thus partially addressed by "correct-by-construction" Software (SW) code-generation.

Pushing the limits of automated program construction is one of the more interesting developments in both SW and Hardware (HW) generation nowadays. Although it has been accepted more in the former, with the advent of high-level synthesis tools such as Forte Cynthesizer [10] and Calypto Catapult [4], the pragmatics of creation of SW-defined systems have been shifted to HW. It is, in fact, apparent that describing hardware using C/C++ (languages that have grown together with the growth of SW) can bring to the level of hardware the same sort of problems that were previously observed only in the software world.

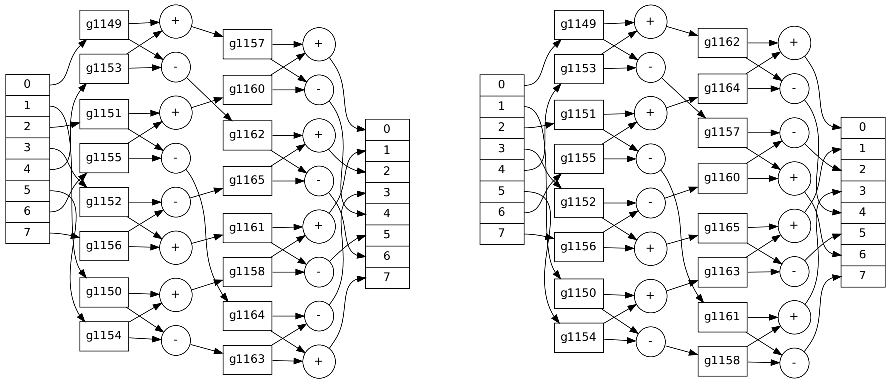


Fig. 8. FFT-4 in complex domain: DIT (left) & DIF (right)

Therefore, it is important to inherit not only problems, but also their partial solutions such as the "correct-by-construction" code-generation techniques for making Intellectual Property (IP) blocks using higher-level synthesis tools.

FFT is one example of such a block, found in most applications of embedded signal processing. We build on the prior-art and contribute insights into the SDR application domain from Scheme perspective by executing the following plan: (1) take the application expert's hat and provide two alternative versions of the

FFT algorithm, mimicking the process of adaptation of IP blocks, (2) show the ease with which such algorithms can be *algebraically* specified, and subsequently algebraically *modified*, without the need to understand the mechanics of monads and monadic interpreters, and (3) automatically process the IP blocks by a separate (fixed) layer in the stack (the monadic interpreter, and then a code generator). This last layer is written and verified only once by domain experts, the rôle which we assume in Subsection 4.3.

Unlike prior art, our work did not require extensions to the language interpreter and compiler itself (cf. implementation of the Meta-OCaml compiler). Leveraging Scheme's syntactic extensions, we present a DSL for describing such algorithms. The DSL hides most of the monadic constructs by utilising the following techniques. First, the [] notation that delineates the block-generating expression(s) is implemented as an unsafe (but small) `set-read-syntax!` handler that desugars into a safe `syntax - rules` back-end macro to handle monadic computations. This is triggered by the `#lang - monads` "pragma" in Fig. 9. Second, the `expensive:` annotation (used in e.g., line 3 of Fig. 10) that triggers sharing of functional unit outputs for optimization, is implemented as an add-on to the same back-end macro. Third, the `algo` form that declares a multi-staged computation and encapsulates the annotation for working in a particular monad is implemented as a `syntax-rule` in Fig. 9.

```
(define-syntax-rule [algo form . body]
    (def form (with-access::monad
                scontM (unit bind)
       . body)
      ))
#lang-monads
(define-syntax + add_u_a)
(define-syntax - sub_u_a)
(define-syntax * mult_u_a)
<<FFT DIT>> ; see Figure 10.
<<FFT DIF>> ; see Figure 11.
#lang-std
```

Fig. 9. FFT network generator top-level

4.1 DIT

The FFT algorithm can be defined by the following recursive equations. An illustration of the computation that the equations specify can be found on the left of Fig. 8, where rectangular blocks represent memory locations or register cells, and circular blocks represent functional units.

$$\mathrm{FFT}^1(l) = l, \forall k \in \mathbb{N} \begin{cases} k \equiv 1 \pmod 2 \\ k \equiv 0 \pmod 2 \end{cases} 0 \le i < \frac{n}{2} \tag{1}$$

$$\mathrm{FFT}_i^n(l) = \mathrm{FFT}_i^{\frac{n}{2}}\{l_k | k \equiv 0\} + e^{i\frac{-2\pi j}{n}}\,\mathrm{FFT}_i^{\frac{n}{2}}\{l_k | k \equiv 1\} \tag{2}$$

$$\mathrm{FFT}_{i+\frac{n}{2}}^n(l) = \mathrm{FFT}_i^{\frac{n}{2}}\{l_k | k \equiv 0\} - e^{i\frac{-2\pi j}{n}}\,\mathrm{FFT}_i^{\frac{n}{2}}\{l_k | k \equiv 1\} \tag{3}$$

The radix-2 DIT FFT algorithm proceeds by *deinterleaving* the list of (complex) samples of length ≥ 2 (singleton lists are returned "as-is"). The two sublists are processed by recursively applying the algorithm to each half, and then merging via application of the *butterfly* in the forward mode. Both results of the butterfly merge (halves of the input list by construction) are flattened using the Scheme's standard **append** procedure. Fig. 10 literally transcribes this algorithm.

```
(algo (FFT_DIT dir) ;Eq.1
 (fn _ ((_) as ls) ⇒ [ls]
| self ls ⇒
 [(e o) := (DEINTR ls)
  y0 :=: (self e)
  y1 :=: (self o)
  (MERGE_DIT dir y0 y1)]))
```

```
(def DEINTR (fn'()→(()())
| (,x ,y .
,[~ DEINTR -> `(,a ,b)])→
   ((,x . ,a) (,y . ,b))))
```

```
(algo (MERGE_DIT d e o)
(letrec((MG(fn j (x . xs)(y . ys)⇒
[(expensive: X := x and Y := y
 z := (* (w_a d (* 2 (length e))j)Y))
 S := (+ X z) ; Eq.2 below
 D := (- X z) ; Eq.3 below
 (a b):=:(MG (+ j 1) xs ys)
 return `((,S . ,a) (,D . ,b))]
| else ⇒ ['(() ())])))
[(a b) :=: (MG 0 e o)
 return (append a b)]
))
```

Fig. 10. DIT FFT

The **forward** butterfly operation (MERGE_DIT on the right of the figure) essentially implements the recursive equations 1, 2 and 3. The algebraic operators are overloaded in the DSL to use the complex arithmetic. Although native complex numbers support is optional in Scheme (and Bigloo does not have complex numbers) we have, nevertheless, managed to implement both *cartesian* (simplifying addition) and *polar* (simplifying multiplication) complex numbers, leveraging *active* patterns and hygienic macros for extending the standard arithmetic operators like `+`, `*` etc. to complex operands.

4.2 DIF

The DIF FFT algorithm proceeds in the reverse direction. Essentially, every operation of the DIT version is reversed: deinterleaving $\mapsto$ *interleaving*, the forward butterfly (i.e., merging) $\mapsto$ backwards butterfly (i.e., splitting), `append` $\mapsto$ SPLIT (deterministic due to halving property of both DEINTR and SPLIT, as well as functoriality of the FFT operation), while the overall sequencing is reversed.

$$\mathrm{FFT}^1(l) = l, 0 \le i < \frac{n}{2}, 0 \le k < \frac{n}{2} \tag{4}$$

$$\mathrm{FFT}^n_{2i}(l) = \mathrm{FFT}^{\frac{n}{2}}_i\{l_k\} + \mathrm{FFT}^{\frac{n}{2}}_i\{l_{k+\frac{n}{2}}\} \tag{5}$$

$$\mathrm{FFT}^n_{2i+1}(l) = e^{i\frac{-2\pi j}{n}}(\mathrm{FFT}^{\frac{n}{2}}_i\{l_k\} - \mathrm{FFT}^{\frac{n}{2}}_i\{l_{k+\frac{n}{2}}\}) \tag{6}$$

This is apparent from Fig. 11, which depicts an algorithm that is structured exactly like the one from the previous subsection (modulo direction). The **backwards** butterfly (SPLIT_DIF on the right of the figure) essentially implements the recursive equations 4, 5 and 6, which are obtained by calculating Eq.2 + Eq.3 and Eq.2 − Eq.3.

Note that some unnecessary sharing (introduced by the first `expensive:` block from the figure) was visible in the DIF network from Fig. 8. It was of course removed by an extra post-processing step, which we can not highlight here, along with the implementation of SPLIT (an inverse of `append`), due to space limitations. Reversibility of both variants of the FFT should be apparent.

```
(algo (FFT_DIF d) ;Eq.4
 (fn _ ((_) as ls) ⇒ [ls]
  | self ls ⇒
 [(h1 h2):=:(SPLIT_DIF d ls)
  y0 :=:(self h1)
  y1 :=:(self h2)
  return (INTR y0 y1)]
 ))
```

```
(def INTR (fn'()()→()
|(,x . ,xs)(,y . ,ys)→(,x ,y
. ,(apply INTR `(,xs ,ys)))
))
```

```
(algo (SPLIT_DIF d ls)
(lets (((h1 h2) (split ls)))
 (letrec((SP(fn j (x . xs)(y . ys)
⇒[(expensive: X := x and Y := y)
    S := (+ X Y) ; Eq.5 above
    D := (- X Y) ; Eq.6 above
    (expensive:
     Z := (* (w_a d (length ls)j)D))
    (a b) :=: (SP (+ j 1) xs ys)
    return `((,S . ,a) (,Z . ,b))]
  | else ⇒ ['(() ())])))
     (SP 0 h1 h2))
 ))
```

Fig. 11. DIF FFT

4.3 Let-coalescing by Inflation of Administrative Redexes

The Scheme code that is generated by the monadic interpreter for FFTs contains a long chain of nested applications of 2-argument functions. Each application corresponds to a complex operation. Therefore, some post-processing of the results is required in order to generate C code that can be efficiently mapped on Digital Signal Processor (DSP) cores and in HW. In addition, it is also useful to generate visualization that is directly recognizable by SDR application experts.

```
(def (COALESCE a u)
    (fn' ; unapplied abstraction
      | (lambda (,var) ,[~(body)]) →
          (lambda (,var) ,body)
       ; applied lambda with one or more args
      | ((lambda (,var . ,vars) ,body) ,ex . ,exps) ⇒
          (if [any (_ occurs? ex) a]          ; _ is right-curry
                `((lambda ,a
                      ,((COALESCE '() '())
                          `((lambda (,var . ,vars) ,body)
                                ,ex . ,exps))) . ,u)
                ((COALESCE (cons var a) (cons ex u))
                 (if [or (null? exps) (null? vars)] body
                      `((lambda ,vars ,body) . ,exps))))
       ; no more abstractions
      | ,body → ((lambda ,a ,body) . ,u)
    ))
```

Fig. 12. Coalescing by inflation of administrative redexes

To accomplish this, we took a domain expert's hat and applied a simple transformation on the structure of the generated Scheme vector code that *coalesces* all such 2-argument functions. We accumulate bindings and their values as we recurse on the structure of the code. As soon as a dependency of a value on any of the already accumulated bindings is detected, a *redex* is emitted. This redex thus contains all independent bindings (and their values) that were accumulated so far. Fig. 12 depicts the full implementation of this post-processing step.

The net effect of this post-processing is that the administrative redexes - corresponding to static scopes in C with no shallow dependencies inside - are *inflated* as much as possible. Because all operations in each stage are independent of each other, they can be performed in *parallel*. In addition, this transformation has allowed Graphviz to align maximal number of independent bindings (register cells) in vertical columns and thereby accentuate the number of stages in a FFT network. One can now clearly see the butterflies from the layout as well as the difference between the DIT and DIF modes of the FFT.

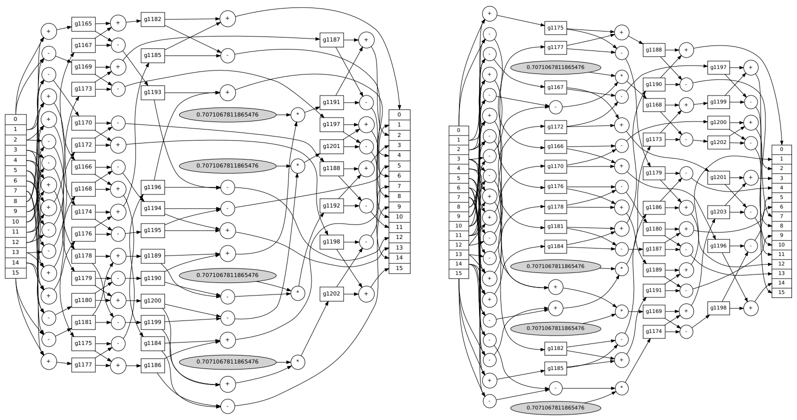


Fig. 13. FFT-8 in complex domain: DIT (left) & DIF (right)

Summarizing, the 2-level η-expansion of the results obtained by running the monadic interpreter, their subsequent post-processing, conversion to Graphviz format and final visualization given a network returned by FFT^4 and FFT^8 is depicted in Figures 8 and 13, respectively. Note that the first-stage register loads have been elided for FFT^8 to simplify the visuals. We have verified that the approach scales to practical FFTs with sizes up-to 8192 (complex samples), and that the generated code correctly operates in both PC and DSP SW environments as well as passed it through the Forte Cynthesizer HW flow. We elide the results on the number of register cells, multipliers and adders for this paper.

The actual C generation back-end was straightforward to implement (once our pattern-match implementation was ready) and is included in Fig. 14. Thanks to Scheme's homoiconicity, simplicity and syntactic flexibility, both tasks turned out to be feasible to perform directly from an interactive Read-Eval-Print-Loop (REPL) session. All of the components presented in this paper were implemented as a regular user-level application/library (no expensive compiler bootstrapping or interpreter hacking was involved), which allowed for rapid experimentation and debugging.

```
(def (CPROG n)
 (letrec
  ((CEXPR (fn ; handle simple expressions
     | ('quote [~(n)]) ⇒ n
     | (symbol? $ -: n) ⇒ n ; $ is symbol->string
     | (number? % -: n) ⇒ n ; % is number->string
     | `(vector-ref ,d ,[~(i)]) ⇒
          (cat ($ d) "[" i "]")
     | `(+ ,[~(x)] ,[~(y)]) ⇒
          (cat "(" x "+" y ")")
     | `(- ,[~(x)] ,[~(y)]) ⇒
          (cat "(" x "-" y ")")
     | `(* ,[~(x)] ,[~(y)]) ⇒
          (cat "(" x "*" y ")")
   ))
   (CSTMT (fn ; handle simple statements
     | `(vector-set! ,d ,('quote i) ,[~ CEXPR -> val]) ⇒
         (cat ($ d) "[" (% i) "]=" val ";")
     | `(begin . ,[~(ss)]) ⇒
         (cat ss)
     | () → ""
     | `(,[~(s1)] . ,[~(ss)]) ⇒
          (cat nl s1 ss)
     | [~ CEXPR -> val] ⇒
          (cat "return " val ";" nl)
   )))
   (fn ; unapplied abstraction
     | `(lambda (,var) ,[~(body)]) ⇒
      (cat "sample_t* fft_" (% n) "("
       "sample_t " ($ var) "[" (% (* 2 n)) "])"
       nl "{" body "}")
    ; applied lambda - redex
     | `((lambda ,vars ,[~(body)]) . ,exps) ⇒
      (cat nl
        "{"
        (fold-left (fn r v e ⇒
            (cat r nl "sample_t " ($ v) "=" [CEXPR e] ";")
               )
            ""
            vars exps)
        body
        "}")
    ; no more abstractions
     | [~ CSTMT -> body] ⇒ body
   )))
```

Fig. 14. Domain-specific Scheme↦C generator

5 Conclusion

Wikipedia: *Lingua franca* is a language systematically used to make communication possible between people not sharing a mother tongue, in particular when it is a 3rd language, distinct from both ...

It is good to see the "semantic" gap between the academic and engineering communities closing. "Syntactic" gap, however, still remains. In this paper, we focus on Scheme as a promising common language (i.e., a "Lingua franca") from both language (academic) and the application domain (engineering) point of view. We contribute a number of syntactic extensions that implement pattern-matching, monadic computations and comprehensions we well as provide a convenient DSL for multi-stage programming.

By complementing Scheme's minimalism with extensible syntax, one can cope with the inevitable verbosity in a manner that suits specific development communities, or conflicting developer preferences. We have applied syntactic extensions to address a specific problem in the embedded community - the need to apply modern "correct-by-construction" IP block generation methods in the context of SDR signal processing. We focused on a common block found in this domain and applied the proposed techniques to a FFT generator, first assuming the rôle of an application expert in modifying the algorithm to work in reverse, and then taking the hat of a domain expert in post-processing the generated vector code to improve the resource usage and provide automated tuning to the generated C code and generated Graphviz output.

We hope to have illustrated in this paper that using such advanced program construction methods does not need advanced compiler work. With Scheme, in fact, one can build the needed blocks incrementally, as we have shown in implementing pattern-matching and monads. We did this by leveraging the hygienic `syntax-rules`, sequencing with HOFs, and basic data-types: the first stage only uses *lists*, while the second stage of our FFT implementation uses *vectors*. Following in the spirit of the [29], we present a number of *libraries* that implement these constructs in a synergistic way.[7] We suggest that Scheme can indeed become the sought *Lingua franca*, usable by both embedded application & domain experts when communicating with the FP community.

5.1 Future Work

Note that some of our FFT block-generating functions, as well as some miscellaneous functions are bidirectional. Most notably, INTR from Fig. 11 and DEINTR

[7] All of the examples shown in the paper are extracted from operational code. Minor modifications to Bigloo included an implementation of the SRFI-10 reader-macro support and `set-read-syntax!` (as well as `set-sharp-read-syntax!`, both non-standard as of now). None of these were made specific to our application. Integration of Bigloo with the SRFI-46 compliant expander for `syntax-rules` (alexpander) was transparent and rather painless, thanks to the ability of adding user-defined passes.

from Fig. 10 are inter-derivable if one swaps the LHS and the RHS of the pattern-match rules, exchanging `[~ fun -> terms]` for `(apply fun terms)`. We consider it future work to extend the matcher to handle ellipsis (...) patterns on the RHS and provide an automated inverse generator.

Particularly interesting is the prospect of addressing the embedded community's language requirements by bringing our proposed extensions of Scheme closer to languages such as Matlab. This is visible, for example, in the seemingly imperative assignment "statement" and destructuring operations on the values returned by block-generating functions in FFT figures.

Acknowledgments. The authors would like to thank the anonymous reviewers for their helpful comments. We are indebted to Manuel Serrano (for Bigloo), Walid Taha (for Meta-OCaml), Chung-chieh Shan (for his contributions to Meta-Scheme), Oleg Kiselyov (for his comprehensive site), Daniel Friedman, Simon Peyton-Jones and Philip Wadler (a.o., for catamorphisms and views), Standard ML (SML), Objective Caml (OCaml) and Haskell communities for inspiration, R7RS mailing list for discussions and Graphviz for providing a fantastic lay-out for the butterflies!

References

1. Adams IV, N.I., Bartley, D.H., Brooks, G., Dybvig, R.K., Friedman, D.P., Halstead, R., Hanson, C., Haynes, C.T., Kohlbecker, E., Oxley, D., Pitman, K.M., Rozas, G.J., Steele Jr., G.L., Sussman, G.J., Wand, M., Abelson, H.: Revised^5 report on the algorithmic language scheme. SIGPLAN Not. 33(9), 26–76 (1998)
2. Alexandrescu, A.: Better template error messages. C/C++ Users J. 17(3), 37–47 (1999)
3. Bender, J.: sxml-match.ss. Internet
4. Calypto. Catapult c. Internet
5. Carette, J., Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. Sci. Comput. Program. 76(5), 349–375 (2011)
6. Cohen, A., Donadio, S., Garzaran, M.-J., Herrmann, C., Kiselyov, O., Padua, D.: In search of a program generator to implement generic transformations for high-performance computing. Sci. Comput. Program. 62, 25–46 (2006)
7. Erwig, M.: Active patterns. In: Kluge, W.E. (ed.) IFL 1996. LNCS, vol. 1268, pp. 21–40. Springer, Heidelberg (1997)
8. Filinski, A.: Representing monads. In: POPL, pp. 446–457 (1994)
9. Foltzer, A.C., Friedman, D.P.: A schemers view of monads. Partial draft (2011)
10. Forte, D.S.: Cynthesizer. Internet
11. Friedman, D., Hilsdale, E., Ganz, S., Dybvig, K.: match.ss. Internet
12. Gabriel, R.P.: The why of y. SIGPLAN Lisp Pointers 2(2), 15–25 (1988)
13. Kiselyov, O.: Monadic programming in scheme. Internet (2005)
14. Kiselyov, O., Swadi, K.N., Taha, W.: A methodology for generating verified combinatorial circuits. In: Proceedings of the 4th ACM International Conference on Embedded Software, EMSOFT 2004, pp. 249–258. ACM, New York (2004)
15. Krishnamurthi, S.: Educational pearl: Automata via macros. J. Funct. Program. 16(3), 253–267 (2006)

16. Lindley, S., Wadler, P., Yallop, J.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electr. Notes Theor. Comput. Sci. 229(5), 97–117 (2011)
17. Meijer, E., Hutton, G.: Bananas in space: Extending fold and unfold to exponential types. In: FPCA, pp. 324–333 (1995)
18. Moggi, E.: Notions of computation and monads. Inf. Comput. 93(1), 55–92 (1991)
19. Queinnec, C.: Compilation of non-linear, second order patterns on s-expressions. In: Deransart, P., Małuszyński, J. (eds.) PLILP 1990. LNCS, vol. 456, pp. 340–357. Springer, Heidelberg (1990)
20. Queinnec, C., Geffroy, P.: Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In: WSA, pp. 109–117 (1992)
21. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In: Visser, E., Järvi, J. (eds.) GPCE, pp. 127–136. ACM (2010)
22. Serrano, M., Weis, P.: Bigloo: A portable and optimizing compiler for strict functional languages. In: Mycroft, A. (ed.) SAS 1995. LNCS, vol. 983, pp. 366–381. Springer, Heidelberg (1995)
23. Shinn, A.: match.scm. Internet
24. Siskind, J.M.: Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, Inc. (1999)
25. Stroustrup, B.: The c++0x “remove concepts” decision. Internet
26. Sutton, A., Stroustrup, B.: Design of concept libraries for c++. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 97–118. Springer, Heidelberg (2012)
27. Swadi, K.N., Taha, W., Kiselyov, O., Pasalic, E.: A monadic approach for avoiding code duplication when staging memoized functions. In: Hatcliff, J., Tip, F. (eds.) PEPM, pp. 160–169. ACM (2006)
28. Tobin-Hochstadt, S.: Extensible pattern matching in an extensible language. CoRR, abs/1106.2578 (2011)
29. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: Hall, M.W., Padua, D.A. (eds.) PLDI, pp. 132–141. ACM (2011)
30. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science 2(4), 461–493 (1992)
31. Wright, A.K., Duba, B.F.: Pattern matching for scheme. Technical Report TX 77251-1892, Rice University (May 1995)
32. Yallop, J.: Ocaml patterns extesions. Internet

A Acronyms

CBV	Call-by-Value
DIF	Decimation in Frequency
DIT	Decimation in Time
DSL	Domain-Specific Language
DSP	Digital Signal Processor
FEC	Forward Error Correction
FFT	Fast Fourier Transform
FP	Functional Programming
FSM	Finite State Machine
HM	Hindley-Milner
HOF	High-Order Function
HW	Hardware
IO	Input-Output
IP	Intellectual Property
LHS	Left-Hand Side
LISP	List Processing
MAC	Media Access Control
ML	Meta Language
OCaml	Objective Caml
PC	Personal Computer
R5RS	Revised^5 Report on the Algorithmic Language Scheme
R7RS	Revised^7 Report on the Algorithmic Language Scheme
REPL	Read-Eval-Print-Loop
RHS	Right-Hand Side
SDR	Software-Defined Radio
SML	Standard ML
SRFI	Scheme Request for Implementation
SW	Software
TRS	Term-Rewriting System
ZF	Zermelo-Fraenkel

Haskell Gets Argumentative

Bas van Gijzel and Henrik Nilsson

Functional Programming Laboratory,
School of Computer Science,
University of Nottingham,
United Kingdom
{bmv,nhn}@cs.nott.ac.uk

Abstract. Argumentation theory is an interdisciplinary field studying how conclusions can be reached through logical reasoning. The notion of argument is completely general, including for example legal arguments, scientific arguments, and political arguments. *Computational* argumentation theory is studied in the context of artificial intelligence, and a number of computational argumentation frameworks have been put forward to date. However, there is a lack of concrete, high level realisations of these frameworks, which hampers research and applications at a number of levels. We hypothesise that the lack of suitable domain-specific languages in which to formalise argumentation frameworks is a contributing factor. In this paper, we present a formalisation of a particular computational argumentation framework, Carneades, as a case study with a view to investigate the extent to which functional languages are useful as a means to realising computational argumentation frameworks and reason about them.

Keywords: computational argumentation theory, domain-specific languages, functional programming.

1 Introduction

Argumentation theory is an interdisciplinary field studying how conclusions can be reached through logical reasoning. Argumentation should here be understood in a general sense, including for example political debates along with more rigorous settings such as a legal or a scientific argument. A central aspect is that there is usually not going to be any clear-cut truth, but rather arguments and counter-arguments, possibly carrying different weights, and possibly relative to to the understanding of a specific audience and what is known at a specific point in time. The question, then, is what it means to systematically evaluate such a set of arguments to reach a rational conclusion, which in turn leads to the notion of *proof standards*, such as "beyond reasonable doubt". Fields that intersect with argumentation theory thus include philosophy (notably epistemology and the philosophy of science and mathematics), logic, rhetoric, and psychology.

Argumentation theory has also been studied from a *computational* perspective in the field of artificial intelligence, with the dual aim of studying argumentation theory as such [1,2] and of more direct applications such as verification of arguments [3] or for programming autonomous agents capable of argumentation [4]. Dung's *abstract* argumentation framework [5] has been particularly influential, as it attempts to capture only the essence of arguments, thus making it generally applicable across different argumentation domains.

Since Dung's seminal work, a number of other computational argumentation frameworks have been proposed, and the study of their relative merits and exact, mathematical relationships is now an active sub-field in its own right [6,7,8,9,10]. However, a problem here is the lack of concrete realisations of many of these frameworks, in particular realisations that are sufficiently close to the mathematical definitions to serve as specifications in their own right. This hampers communication between argumentation theorists, impedes formal verification of frameworks and their relationships as well as investigation of their computational complexity, and raises the barrier of entry for people interested in developing practical applications of computational argumentation.

We believe that a contributing factor to this state of affairs is the lack of a language for expressing such frameworks that on the one hand is sufficiently high-level to be attractive to argumentation theorists, and on the other is rigorous and facilitates formal (preferably machine-checked) reasoning. We further hypothesise that a functional, domain-specific language (DSL) would be a good way to address this problem, in particular if realised in close connection with a proof assistant.

The work presented in this paper is a first step towards such a language. In order to learn how to best capture argumentation theory in a functional setting, we have undertaken a case study of casting a particular computational argumentation framework, Carneades [11,12], into Haskell[1]. We ultimately hope to generalise this into an embedded DSL for argumentation theory, possibly within the dependently typed language Agda with a view to facilitate machine checking of proofs about arguments and the relationships between argumentation frameworks. While we are still a long way away from this goal, the initial experience from our case study has been positive: our formalisation in Haskell was deemed to be intuitive and readable as a specification on its own by Tom Gordon, an argumentation theorist and one of the authors of the Carneades argumentation framework [13]. Furthermore, our case study is a contribution in its own right in that it:

– already constitutes a helpful tool for argumentation theorists;
– demonstrates the usefulness of a language like Haskell itself as a tool for argumentation theorists, albeit assuming a certain proficiency in functional programming;
– is a novel application of Haskell that should be of interest for example to researchers interested in using Haskell for AI research and applications.

This is not to say that there are no implementations of *specific* argumentation theory frameworks around; see Sect. 4 for an overview. However, the goals and structure of those systems are rather different from what we are concerned

[1] Cabal package on Hackage: `http://hackage.haskell.org/package/CarneadesDSL`

with in our case study. In particular, a close and manifest connection between argumentation theory and its realisation in software appears not to be a main objective of existing work. For us, on the other hand, maintaining such a connection is central, as this is the key to our ultimate goal of a successful *generic* DSL suitable for realising *any* argumentation framework.

The rest of this paper is structured as follows. In Sect. 2, we give an intuitive introduction to Carneades, both to provide a concrete and easy to grasp example of what argumentation frameworks are and how they work, and to provide a grounding for the technical account of Carneades and our implementation of it that follows. We then continue in Sect. 3 by giving the formal definition of the central parts of Carneades juxtapositioned with our realisation in Haskell. The section covers central notions such as the argumentation graph that captures the relationships between arguments and counter arguments, the exact characterisation of proof standards (including "beyond reasonable doubt"), and the notion of an audience with respect to which arguments are assigned weights. Related work is discussed in Sect. 4, and we conclude in Sect. 5 with a discussion of what we have learnt from this case study, its relevance to argumentation theorists, and various avenues for future work.

2 Background: The Carneades Argumentation Model

The main of purpose of the Carneades argumentation model is to formalise argumentation problems in a legal context. Carneades contains mathematical structures to represent arguments placed in favour of or against atomic propositions; i.e., an argument in Carneades is a *single* inference step from a set of *premises* and *exceptions* to a *conclusion*, where all propositions in the premises, exceptions and conclusion are literals in the language of propositional logic. For example, Fig. 1 gives an argument in favour of the proposition *murder* mimicking an argument that might be put forward in a court case.

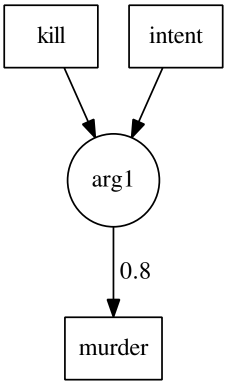
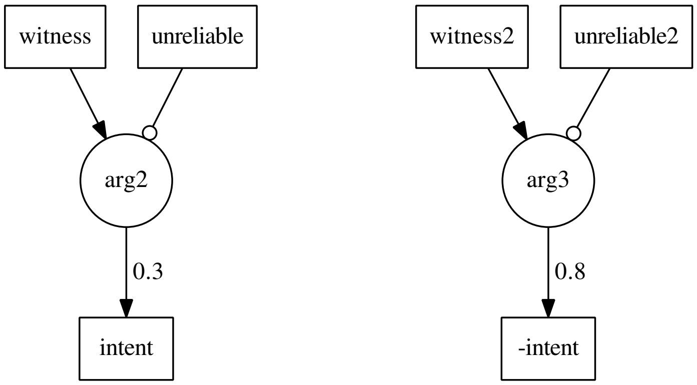


Fig. 1. Carneades argument for murder

For ease of reference, we name the argument (*arg1*). However, arguments are not formally named in Carneades, but instead identified by their logical content. An argument is only to be taken into account if it is *applicable* in a technical sense defined in Carneades. In this case, *arg1* is applicable given that its two premises *kill* and *intent* are *acceptable*, also in a technical sense defined in Carneades. (We will come back to exceptions below.) In other words, we are able to derive that there was a murder, given that we know (with sufficient certainty) that someone was killed and that this was done with intent.

In Carneades, a set of arguments is evaluated relative to a specific *audience* (jury). The audience determines two things: a set of *assumptions*, and the *weight* of each argument, ranging from 0 to 1. The assumptions are the premises and exceptions that are taken for granted by the audience, while the weights reflect the subjective merit of the arguments. In our example, the weight of *arg1* is 0.8, and it is applicable if *kill* and *intent* are either assumptions of the audience, or have been derived by some other arguments, relative to the *same* audience.

Things get more interesting when there are arguments both for and against the same proposition. The conclusion of an argument against an atomic proposition is the propositional negation of that proposition, while an argument against a negated atomic proposition is just the (positive) proposition itself. Depending on the type of proposition, and even the type of case (criminal or civil), there are certain requirements the arguments should fulfil to tip the balance in either direction. These requirements are called *proof standards*. Carneades specifies a range of proof standards, and to model opposing arguments we need to assign a specific proof standard, such as *clear and convincing evidence*, to a proposition.



(a) Pro *intent* (b) Con *intent*

Fig. 2. Arguments pro and con *intent*

Consider the two arguments in Fig. 2, where the arrows with circular heads indicate exceptions. Fig. 2(a) represents an argument in favour of *intent*. It is applicable given that the premise *witness* is acceptable and the exception *unreliable* does not hold. Fig. 2(b) represents an argument against *intent*. It involves a second witness, *witness2*, who claims the opposite of the first witness. Let us assume that the required proof standard for *intent* indeed is *clear and convincing evidence*, which Carneades formally defines as follows:

Definition 1 (Clear and convincing evidence). *Given two globally predefined positive constants α and β; clear and convincing evidence holds for a specific proposition p iff*

- *There is at least one applicable argument for proposition p that has at least a weight of α.*
- *The maximal weight of the applicable arguments in favour of p are at least β stronger than the maximal weight of the applicable arguments against p.*

Taking $\alpha = 0.2$, $\beta = 0.3$, and given an audience that determines the argument weights to be as per the figure and that assumes $\{\mathit{witness}, \mathit{witness2}\}$, we have that $-\mathit{intent}$ is acceptable, because *arg3* and *arg2* are applicable, weight($\mathit{arg3}$) $>$ α, and weight($\mathit{arg3}$) $>$ weight($\mathit{arg2}$) $+ \beta$.

For another example, had *unreliable2* been assumed as well, or found to be acceptable through other (applicable) arguments, that would have made *arg3* inapplicable. That in turn would make *intent* acceptable, as the weight 0.3 of *arg2* satisfies the conditions for clear and convincing evidence given that there now are no applicable counter arguments, and we could then proceed to establish *murder* by *arg1* had it been established that someone indeed was killed.

3 Towards a DSL for Carneades in Haskell

3.1 Arguments

As our ultimate goal is a DSL for argumentation theory, we strive for a realisation in Haskell that mirrors the mathematical model of Carneades argumentation framework as closely as possible. Ideally, there would be little more to a realisation than a transliteration. We will thus proceed by stating the central definitions of Carneades along with our realisation of them in Haskell.

Definition 2 (Arguments). *Let $\mathcal{L}$ be a propositional language. An* argument *is a tuple $\langle P, E, c\rangle$ where $P \subset \mathcal{L}$ are its* premises, *$E \subset \mathcal{L}$ with $P \cap E = \emptyset$ are its* exceptions *and $c \in \mathcal{L}$ is its* conclusion. *For simplicity, all members of $\mathcal{L}$ must be literals, i.e. either an atomic proposition or a negated atomic proposition. An argument is said to be* pro *its conclusion c (which may be a negative atomic proposition) and* con *the negation of c.*

In Carneades all logical formulae are literals in propositional logic; i.e., all propositions are either positive or negative atoms. Taking atoms to be strings suffice in the following, and propositional literals can then be formed by pairing this atom with a Boolean to denote whether it is negated or not:

```
type PropLiteral = (Bool, String)
```

We write $\overline{p}$ for the negation of a literal p. The realisation is immediate:

```
negate :: PropLiteral → PropLiteral
negate (b, x) = (¬ b, x)
```

We chose to realise an *argument* as a newtype (to allow a manual Eq instance) containing a tuple of two lists of propositions, its *premises* and its *exceptions*, and a proposition that denotes the *conclusion*:

```
newtype Argument = Arg ([PropLiteral], [PropLiteral], PropLiteral)
```

Arguments are considered equal if their premises, exceptions and conclusion are equal; thus arguments are identified by their logical content. The equality instance for *Argument* (omitted for brevity) takes this into account by comparing the lists as sets.

A set of arguments determines how propositions depend on each other. Carneades requires that there are no cycles among these dependencies. Following Brewka and Gordon [6], we use a dependency graph to determine acyclicity of a set of arguments.

Definition 3 (Acyclic set of arguments). *A set of* arguments *is* acyclic *iff its corresponding dependency graph is acyclic. The corresponding dependency graph has a node for every literal appearing in the set of arguments. A node p has a link to node q whenever p depends on q in the sense that there is an argument pro or con p that has q or $\overline{q}$ in its set of premises or exceptions.*

Our realisation of a set of arguments is considered abstract for DSL purposes, only providing a check for acyclicity and a function to retrieve arguments pro a proposition. We use FGL [14] to implement the dependency graph, forming nodes for propositions and edges for the dependencies. For simplicity, we opt to keep the graph also as the representation of a set of arguments.

```
type ArgSet = ...
getArgs    :: PropLiteral → ArgSet → [Argument]
checkCycle :: ArgSet → Bool
```

3.2 Carneades Argument Evaluation Structure

The main structure of the argumentation model is called a Carneades Argument Evaluation Structure (CAES):

Definition 4 (Carneades Argument Evaluation Structure (CAES)). *A* Carneades Argument Evaluation Structure *(CAES) is a triple*

$$\langle arguments, audience, standard\rangle$$

where arguments is an acyclic set of arguments, audience is an audience as defined below (Def. 5), and standard *is a total function mapping each proposition to to its specific proof standard.*

Note that propositions may be associated with *different* proof standards. This is considered a particular strength of the Carneades framework. The transliteration into Haskell is almost immediate[2]:

$$\textbf{newtype}\ CAES = CAES\ (ArgSet, Audience, PropStandard)$$

Definition 5 (Audience). *Let $\mathcal{L}$ be a propositional language. An* audience *is a tuple* $\langle$assumptions, weight$\rangle$, *where* assumptions $\subset \mathcal{L}$ *is a propositionally consistent set of literals (i.e., not containing both a literal and its negation) assumed to be acceptable by the audience and* weight *is a function mapping arguments to a real-valued weight in the range* $[0, 1]$.

This definition is captured by the following Haskell definitions:

$$\begin{aligned}&\textbf{type}\ Audience = (Assumptions, ArgWeight)\\&\textbf{type}\ Assumptions = [PropLiteral]\\&\textbf{type}\ ArgWeight = Argument \to Weight\\&\textbf{type}\ Weight = Double\end{aligned}$$

Further, as each proposition is associated with a specific proof standard, we need a mapping from propositions to proof standards:

$$\textbf{type}\ PropStandard = PropLiteral \to ProofStandard$$

A proof standard is a function that given a proposition p, aggregates arguments pro and con p and decides whether it is acceptable or not:

$$\textbf{type}\ ProofStandard = PropLiteral \to CAES \to Bool$$

This aggregation process will be defined in detail in the next section, but note that it is done relative to a specific CAES, and note the cyclic dependence at the type level between *CAES* and *ProofStandard*.

The above definition of proof standard also demonstrates that implementation in a typed language such as Haskell is a useful way of verifying definitions from argumentation theoretic models. Our implementation effort revealed that the original definition as given in [11] could not be realised as stated, because proof standards in general not only depend on a set of arguments and the audience, but may need the whole CAES.

3.3 Evaluation

Two concepts central to the evaluation of a CAES are *applicability of arguments*, which arguments should be taken into account, and *acceptability of propositions*, which conclusions can be reached under the relevant proof standards, given the beliefs of a specific audience.

[2] Note that we use a newtype to prevent a cycle in the type definitions.

Definition 6 (Applicability of arguments). *Given a set of arguments and a set of assumptions (in an audience) in a CAES C, then an argument* $a = \langle P, E, c\rangle$ *is* applicable *iff*

- $p \in P$ *implies* p *is an assumption or [*$\overline{p}$ *is not an assumption and* p *is acceptable in* C *] and*
- $e \in E$ *implies* e *is not an assumption and [*$\overline{e}$ *is an assumption or* e *is not acceptable in* C *].*

Definition 7 (Acceptability of propositions). *Given a CAES* C*, a proposition* p *is* acceptable *in* C *iff* $(s\ p\ C)$ *is true, where* s *is the proof standard for* p*.*

Note that these two definitions in general are mutually dependent because acceptability depends on proof standards, and most sensible proof standards depend on the applicability of arguments. This is the reason that Carneades restricts the set of arguments to be acyclic. (Specific proof standards are considered in the next section.) The realisation of applicability and acceptability in Haskell is straightforward:

```
applicable :: Argument → CAES → Bool
applicable (Arg (prems, excns, _)) caes@(CAES (_, (assumptions, _), _))
  = and $ [(p ∈ assumptions) ∨ (p `acceptable` caes) | p ← prems]
          ++
          [(e ∈ assumptions) ↓ (e `acceptable` caes) | e ← excns ]
    where
      x ↓ y = ¬ (x ∨ y)
acceptable :: PropLiteral → CAES → Bool
acceptable c caes@(CAES (_, _, standard))
  = c `s` caes
  where s = standard c
```

3.4 Proof Standards

Carneades predefines five proof standards, originating from the work of Freeman and Farley [15,16]: *scintilla of evidence*, *preponderance of the evidence*, *clear and convincing evidence*, *beyond reasonable doubt* and *dialectical validity*. Some proof standards depend on constants such as α, β, γ; these are assumed to be defined once and globally. This time, we proceed to give the definitions directly in Haskell, as they really only are translitarations of the original definitions.

For a proposition p to satisfy the weakest proof standard, scintilla of evidence, there should be at least one applicable argument pro p in the CAES:

```
scintilla :: ProofStandard
scintilla p caes@(CAES (g, _, _))
 = any (`applicable`caes) (getArgs p g)
```

Preponderance of the evidence additionally requires the maximum weight of the applicable arguments pro p to be greater than the maximum weight of the applicable arguments con p. The weight of zero arguments is taken to be 0. As the maximal weight of applicable arguments pro and con is a recurring theme in the definitions of several of the proof standards, we start by defining those notions:

```
maxWeightApplicable :: [Argument] → CAES → Weight
maxWeightApplicable as caes@(CAES (_, (_, argWeight), _))
 = foldl max 0 [argWeight a | a ← as, a ‘applicable‘ caes]
maxWeightPro :: PropLiteral → CAES → Weight
maxWeightPro p caes@(CAES (g, _, _))
 = maxWeightApplicable (getArgs p g) caes
maxWeightCon :: PropLiteral → CAES → Weight
maxWeightCon p caes@(CAES (g, _, _))
 = maxWeightApplicable (getArgs (negate p) g) caes
```

We can then define the proof standard preponderance:

```
preponderance :: ProofStandard
preponderance p caes = maxWeightPro p caes > maxWeightCon p caes
```

Clear and convincing evidence strengthen the preponderance constraints by insisting that the difference between the maximal weights of the pro and con arguments must be greater than a given positive constant β, and there should furthermore be at least one applicable argument pro p that is stronger than a given positive constant α:

```
clear_and_convincing :: ProofStandard
clear_and_convincing p caes
 =  (mwp > α) ∧ (mwp − mwc > β)
   where
     mwp = maxWeightPro p caes
     mwc = maxWeightCon p caes
```

Beyond reasonable doubt has one further requirement: the maximal strength of an argument con p must be less than a given positive constant γ; i.e., there must be no reasonable doubt:

```
beyond_reasonable_doubt :: ProofStandard
beyond_reasonable_doubt p caes
 = clear_and_convincing p caes ∧ (maxWeightCon p caes < γ)
```

Finally dialectical validity requires at least one applicable argument pro p and no applicable arguments con p:

```
dialectical_validity :: ProofStandard
dialectical_validity p caes
   = scintilla p caes ∧ ¬ (scintilla (negate p) caes)
```

3.5 Convenience Functions

We provide a set of functions to facilitate construction of propositions, arguments, argument sets and sets of assumptions. Together with the definitions covered so far, this constitute our DSL for constructing Carneades argumentation models.

```
mkProp        :: String → PropLiteral
mkArg         :: [String] → [String] → String → Argument
mkArgSet      :: [Argument] → ArgSet
mkAssumptions :: [String] → [PropLiteral]
```

A string starting with a '-' is taken to denote a negative atomic proposition.

To construct an audience, native Haskell tupling is used to combine a set of assumptions and a weight function, exactly as it would be done in the Carneades model:

```
audience :: Audience
audience = (assumptions, weight)
```

Carneades Argument Evaluation Structures and weight functions are defined in a similar way, as will be shown in the next subsection.

Finally, we provide a function for retrieving the arguments for a specific proposition from an argument set, a couple of functions to retrieve all arguments and propositions respectively from an argument set, and functions to retrieve the (not) applicable arguments or (not) acceptable propositions from a CAES:

```
getArgs              :: PropLiteral → ArgSet → [Argument]
getAllArgs           :: ArgSet                → [Argument]
getProps             :: ArgSet                → [PropLiteral]
applicableArgs       :: CAES                  → [Argument]
nonApplicableArgs    :: CAES                  → [Argument]
acceptableProps      :: CAES                  → [PropLiteral]
nonAcceptableProps   :: CAES                  → [PropLiteral]
```

3.6 Implementing a CAES

This subsection shows how an argumentation theorist given the Carneades DSL developed in this section quickly and at a high level of abstraction can implement a Carneades argument evaluation structure and evaluate it as well. We revisit the arguments from Section 2 and assume the following:

$$\begin{aligned} arguments &= \{arg1, arg2, arg3\}, \\ assumptions &= \{kill, witness, witness2, unreliable2\}, \\ standard(intent) &= beyond\text{-}reasonable\text{-}doubt, \\ standard(x) &= scintilla, \text{ for any other proposition x}, \\ \alpha = 0.4,\ &\beta = 0.3,\ \gamma = 0.2. \end{aligned}$$

Arguments and the argument graph are constructed by calling *mkArg* and *mkArgSet* respectively:

```
arg1, arg2, arg3 :: Argument
arg1 = mkArg ["kill", "intent"] [] "murder"
arg2 = mkArg ["witness"] ["unreliable"] "intent"
arg3 = mkArg ["witness2"] ["unreliable2"] "-intent"

argSet :: ArgSet
argSet = mkArgSet [arg1, arg2, arg3]
```

The audience is implemented by defining the *weight* function and calling *mkAssumptions* on the propositions which are to be assumed. The audience is just a pair of these:

```
weight :: ArgWeight
weight arg | arg ≡ arg1 = 0.8
weight arg | arg ≡ arg2 = 0.3
weight arg | arg ≡ arg3 = 0.8
weight _                = error "no weight assigned"

assumptions :: [PropLiteral]
assumptions = mkAssumptions ["kill", "witness",
                             "witness2", "unreliable2"]

audience :: Audience
audience = (assumptions, weight)
```

Finally, after assigning proof standards in the *standard* function, we form the CAES from the argument graph, audience and function *standard*:

```
standard :: PropStandard
standard (_, "intent") = beyond_reasonable_doubt
standard _             = scintilla

caes :: CAES
caes = CAES (argSet, audience, standard)
```

We can now try out the argumentation structure. Arguments are pretty printed in the format *premises* $\sim$ *exceptions* $\Rightarrow$ *conclusion*:

```
getAllArgs argSet
 > [["witness2"]       ~["unreliable2"] ⇒ "-intent",
    ["witness"]        ~["unreliable"]  ⇒ "intent",
    ["kill", "intent"]~[]               ⇒ "murder"]
```

As expected, there are no applicable arguments for $-$*intent*, since *unreliable2* is an exception, but there is an applicable argument for *intent*, namely *arg2*:

```
filter (`applicable` caes) $ getArgs (mkProp "-intent") argSet
 > []
```

```
filter (`applicable` caes) $ getArgs (mkProp "intent") argSet
 > [["witness"] ⇒ "intent"]
```

However, despite the applicable argument *arg2* for *intent*, *murder* should not be acceptable, because the weight of $arg2 < \alpha$. Interestingly, note that we can't reach the opposite conclusion either:

```
acceptable (mkProp "murder") caes
 > False
acceptable (mkProp "-murder") caes
 > False
```

As a further extension, one could for example imagine giving an argumentation theorist the means to see a trace of the derivation of acceptability. It would be straightforward to add further primitives to the DSL and keeping track of intermediate results for acceptability and applicability to achieve this.

4 Related Work

In this section we consider related work of direct relevance to our interests in DSLs for argumentation theory, specifically efforts in the field of computational argumentation theory to implement argumentation frameworks, and DSLs in closely related areas with similar design goals to ours.

For a general overview of implementations and a discussion of limitations regarding experimental testing, see Bryant and Krause [17]. Most closely related to the work presented in this paper is likely the well-developed implementation [18] of Carneades in Clojure[3]. However, the main aim of that implementation is to provide efficient tools, GUIs, and so on for non-specialists, not to express the implementation in a way that directly relates it to the formal model. Consequently, the connection between the implementation and the model is not immediate. This means that the implementation, while great for argumentation theorists only interested in modelling argumentation problems, is not directly useful to a computational argumentation theorist interested in relating models and implementations, or in verifying definitions. The Clojure implementation is thus in sharp contrast to our work, and reinforces our belief in the value of a high-level, principled approach to formalising argumentation theory frameworks.

One of the main attempts to unify work in argumentation theory, encompassing arguments from the computational, philosophical and the linguistic domains, is the Argument Interchange Format (AIF) [19,20]. The AIF aims to capture arguments stated in the above mentioned domains, while providing a common core ontology for expressing argumentative information and relations. Very recent work has given a logical specification of AIF [21], providing foundations for interrelating argumentation semantics of computational models of argumentation, thereby remedying a previous weaknesses of AIF. Our implementation tackles

[3] http://carneades.berlios.de/

the problem from another direction, starting with a formal and computationally oriented language instead.

Walkingshaw and Erwig [22,23] have developed an EDSL for neuron diagrams [24], a formalism in philosophy that can model complex causal relationships between events, similar to how premises and exceptions determine a conclusion in an argument. Walkingshaw and Erwig extend this model to work on non-Boolean values, while at the same time providing an implementation, thereby unifying formal description and actual implementation. This particular goal is very similar to ours. Furthermore, the actual formalisms of neuron diagrams and the Carneades argumentation model are technically related: while an argument on its own is a simple graph, the dependency graph corresponding to the whole Carneades argument evaluation structure is much more complex and has a structure similar to a full neuron diagram. Arguments in Carneades could thus be seen as an easy notation for a specific kind of complex neuron diagrams for which manual encoding would be unfeasible in practice. However, due to the complexity of the resulting encoding, this also means that for an argumentation theorist, neuron diagrams do not offer directly relevant abstractions. That said, Walkingshaw's and Erwig's EDSL itself could offer valuable input on the design for a DSL for argumentation.

Similarly, causal diagrams are a special case of Bayesian networks [25] with additional constraints on the semantics, restricting the relations between nodes to a causal relation (causal diagrams are a graphic and restricted version of Bayesian networks). Building on the already existing relation between Carneades and Bayesian networks [26], we can view the neuron diagrams generalised to non-Boolean values in Carneades by generalising the negation relation and proof standards to non-Boolean values in the obvious way, and picking scintilla of evidence as the proof standard for all propositions. So, in a way, neuron diagrams are a specific case of arguments, using scintilla of evidence as the proof standard. Finally, to compute an output for every combination of inputs, as is done for neuron diagrams, we can vary the set of assumptions accordingly.

However, formal connections between Bayesian networks and (dialectical) argumentation are still in its infancy; most of the work such as Grabmair [26], Keppens [27] and Vreeswijk [28] are high level relations or comparisons, containing no formal proofs.

5 Conclusions and Future Work

In this paper we have discussed the Carneades argumentation model and an implementation of it in Haskell. This paper should be seen as a case study and a step towards a generic DSL for argumentation theory, providing a unifying framework in which various argumentation models can be implemented and their relationships studied. We have seen that the original mathematical definitions can be captured at a similar level of abstraction by Haskell code, thereby allowing for greater understanding of the implementation. At the same time we obtained a domain specific language for the Carneades argumentation framework, allowing argumentation theorists to realise arguments essentially only using a vocabulary with which they are already familiar.

The initial experience from our work has been largely positive. Comments from Tom Gordon [13], one of the authors and implementers of the Carneades argumentation model, suggest that our implementation is intuitive and would even work as an executable specification, which is an innovative approach in argumentation theory as a field. However, our implementation was not seen as usable to non-argumentation theorists, because of the lack of additional tools. We do not perceive this as a worrying conclusion; our framework's focus is on computational argumentation theorists. We rather envision our implementation being used as a testing framework for computational argumentation theorists and as an intermediate language between implementations, providing a much more formal alternative to the existing Argument Interchange Format [20].

One avenue of future work is the generalisation of our DSL to other related argumentation models. It is relatively common in argumentation theory to define an entirely new model to realise a small extension. However, this hurts the meta-theory as lots of results will have to be re-established from scratch. By reducing such an extension to an existing implementation/DSL such as ours, for instance by providing an implementation of an existing formal translation such as [8,10], we effectively formalise a translation between both models, while gaining an implementation of this generalisation at the same time.

This could be taken even further by transferring the functional definitions of an argumentation model into an interactive theorem prover, such as Agda. First of all, the formalisation of the model itself would be more precise. While the Haskell model might seem exact, note that properties such as the acyclicity of arguments, or that premises and exceptions must not overlap, are not inherently part of this model. Second, this would enable formal, *machine-checked*, reasoning *about* the model, such as establishing desirable properties like consistency of the set of derivable conclusions.

Then, if *multiple* argumentation models were to be realised in a theorem prover, relations *between* those models, such as translations, could be formalised. As mentioned in the introduction, there has recently been much work on formalisation of translations between conceptually very different argumentation models [7,8,10,9]. But such a translation can be very difficult to verify if done by hand. Using a theorem prover, the complex proofs could be machine-checked, guaranteeing that the translations preserve key properties of the models. An argumentation theorist might also make use of this connection by inputting an argumentation case into one model and, through the formal translation, retrieve a specification in another argumentation model, allowing the use of established properties (such as rationality postulates [29]) of the latter model.

Finally, we are interested in the possibility of mechanised argumentation as such; e.g., as a component of autonomous agents. We thus intend to look into realising various argumentation models efficiently by considering suitable ways to implement the underlying graph structure and exploiting sharing to avoid unnecessarily duplicated work. Ultimately we hope this would allow us to establish results regarding the asymptotic time and space complexity inherent in various argumentation models, while providing a framework for empirical evaluations and testing problems sets at the same time. Especially the latter is an area that has only recently received attention [17,7], due to the lack of implementations and automated conversion of problem sets.

Acknowledgments. The authors would like to thank the anonymous reviewer from the initial TFP feedback round whose very constructive comments helped to significantly improve this paper. We would furthermore like to thank Tom Gordon and the anonymous reviewers of the second feedback round for their helpful suggestions.

References

1. Chesñevar, C.I., Maguitman, A.G., Loui, R.P.: Logical models of argument. ACM Comput. Surv. 32(4), 337–383 (2000)
2. Prakken, H., Vreeswijk, G.A.W.: Logics for defeasible argumentation. Handbook of Philosophical Logic 4(5), 219–318 (2002)
3. Reed, C., Rowe, G.: Araucaria: software for argument analysis, diagramming and representation. International Journal of AI Tools 13(4), 961–980 (2004)
4. Rahwan, I., Ramchurn, S.D., Jennings, N.R., Mcburney, P., Parsons, S., Sonenberg, L.: Argumentation-based negotiation. Knowl. Eng. Rev. 18(4), 343–375 (2003)
5. Dung, P.M.: On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. Artificial Intelligence 77(2), 321–357 (1995)
6. Brewka, G., Gordon, T.F.: Carneades and abstract dialectical frameworks: a reconstruction. In: Giacomin, M., Simari, G.R. (eds.) Computational Models of Argument. Proceedings of COMMA 2010, Amsterdam etc., pp. 3–12. IOS Press (2010a)
7. Brewka, G., Dunne, P.E., Woltran, S.: Relating the semantics of abstract dialectical frameworks and standard AFs. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 780–785 (2011)
8. van Gijzel, B., Prakken, H.: Relating Carneades with abstract argumentation via the ASPIC$^+$ framework for structured argumentation. Argument & Computation 3(1), 21–47 (2012)
9. Governatori, G.: On the relationship between carneades and defeasible logic. In: van Engers, T. (ed.) Proceedings of the 13th International Conference on Artificial Intelligence and Law (ICAIL 2011). ACM Press (2011)
10. Prakken, H.: An abstract framework for argumentation with structured arguments. Argument & Computation 1, 93–124 (2010)
11. Gordon, T.F., Walton, D.: Proof burdens and standards. In: Simari, G., Rahwan, I. (eds.) Argumentation in Artificial Intelligence, pp. 239–258. Springer, US (2009)
12. Gordon, T.F., Prakken, H., Walton, D.: The Carneades model of argument and burden of proof. Artificial Intelligence 171(10-15), 875–896 (2007)
13. Gordon, T.F.: Personal communication (2012)
14. Erwig, M.: Inductive graphs and functional graph algorithms. Journal Functional Programming 11(5), 467–492 (2001)
15. Freeman, K., Farley, A.M.: A model of argumentation and its application to legal reasoning. Artificial Intelligence and Law 4, 163–197 (1996), 10.1007/BF00118492

16. Farley, A.M., Freeman, K.: Burden of proof in legal argumentation. In: Proceedings of the 5th International Conference on Artificial Intelligence and Law (ICAIL-05), pp. 156–164. ACM, New York (1995)
17. Bryant, D., Krause, P.: A review of current defeasible reasoning implementations. Knowl. Eng. Rev. 23, 227–260 (2008)
18. Gordon, T.F.: An overview of the Carneades argumentation support system. In: Tindale, C., Reed, C. (eds.) Dialectics, Dialogue and Argumentation. An Examination of Douglas Walton's Theories of Reasoning, pp. 145–156. College Publications (2010)
19. Chesñevar, C., McGinnis, J., Modgil, S., Rahwan, I., Reed, C., Simari, G., South, M., Vreeswijk, G., Willmott, S.: Towards an argument interchange format. The Knowledge Engineering Review 21(4), 293–316 (2006)
20. Rahwan, I., Reed, C.: The argument interchange format. In: Simari, G., Rahwan, I. (eds.) Argumentation in Artificial Intelligence, pp. 383–402. Springer, US (2009)
21. Bex, F., Modgil, S., Prakken, H., Reed, C.: On logical specifications of the Argument Interchange Format. Journal of Logic and Computation (2012)
22. Walkingshaw, E., Erwig, M.: A DSEL for Studying and Explaining Causation. In: IFIP Working Conference on Domain Specific Languages (DSL 2011), pp. 143–167 (2011)
23. Erwig, M., Walkingshaw, E.: Causal Reasoning with Neuron Diagrams. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, pp. 101–108 (2010)
24. Lewis, D.: Postscripts to 'Causation'. In: Philosophical Papers, vol. II, pp. 196–210. Oxford University Press (1986)
25. Pearl, J.: Bayesian networks: A model of self-activated memory for evidential reasoning. In: Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, pp. 329–334 (1985)
26. Grabmair, M., Gordon, T.F., Walton, D.: Probabilistic semantics for the carneades argument model using bayesian networks. In: Proceedings of the 2010 Conference on Computational Models of Argument: Proceedings of COMMA 2010, Amsterdam, The Netherlands, pp. 255–266. IOS Press (2010)
27. Keppens, J.: Argument diagram extraction from evidential bayesian networks. Artificial Intelligence and Law 20, 109–143 (2012)
28. Vreeswijk, G.A.W.: Argumentation in bayesian belief networks. In: Rahwan, I., Moraïtis, P., Reed, C. (eds.) ArgMAS 2004. LNCS (LNAI), vol. 3366, pp. 111–129. Springer, Heidelberg (2005)
29. Caminada, M., Amgoud, L.: On the evaluation of argumentation formalisms. Artificial Intelligence 171, 286–310 (2007)

Repeating History: Execution Replay for Parallel Haskell Programs

Henrique Ferreiro[1], Vladimir Janjic[2], Laura M. Castro[1], and Kevin Hammond[2]

[1] Department of Computer Science, University of A Coruña, Spain
{hferreiro,lcastro}@udc.es
[2] School of Computer Science, University of St Andrews, United Kingdom
{vj32,kh}@cs.st-andrews.ac.uk

Abstract. Parallel profiling tools, such as ThreadScope for Parallel Haskell, allow programmers to obtain information about the performance of their parallel programs. However, the information they provide is not always sufficiently detailed to precisely pinpoint the cause of some performance problems. Often, this is because the cost of obtaining that information would be prohibitive for a complete program execution. In this paper, we adapt the well-known technique of *execution replay* to make it possible to simulate a previous run of a program. We ensure that the non-deterministic parallel *behaviour* of the Parallel Haskell application is properly emulated while its deterministic *functionality* is unmodified. In this way, we can gather additional data about the behaviour of a parallel program by replaying some parts of it with more detailed profiling information. We exploit this ability to identify performance bottlenecks in a *quicksort* implementation, and to derive a version that achieves an 82% speedup over a naive parallelisation.

1 Introduction

Writing *correct* parallel programs in pure functional languages, such as Glasgow Parallel Haskell (GpH [10,15]), is relatively simple. The absence of side-effects means that it is not necessary to worry about some situations such as race conditions or deadlocks that can seriously complicate parallel programs written using more traditional techniques. However, writing *good* parallel programs, which will give good speedups on a wide variety of parallel architectures, is much harder. Understanding why a seemingly "perfect" parallel program does not perform the way the programmer expects can be difficult, especially in a lazy language like Haskell. Profiling can help in understanding the performance of parallel programs. Current profiling tools, such as ThreadScope [6] and cost centre profiling [3], allow the programmer to obtain some information about the behaviour of a parallel program. However, the information that they give is often too low-level to pinpoint performance problems (e.g. in the case of ThreadScope), or their use can even change the runtime behaviour of the original program (e.g. in the case of cost centre profiling).

In this paper, we describe how to adapt the well-known technique of *execution replay* [14] to allow us to use performance debugging for parallel functional programs. Traditionally, execution replay has been used to debug imperative programs, and, as its name suggests, it aims to replay the execution of a program in order to reproduce the same state of the memory and registers as in the original execution. To the best of our knowledge, this paper represents both the first attempt to use this technique in the context of a lazy functional language, and its first adaptation to parallel performance debugging. With our implementation, the repeated execution of a program is *simulated* in a way that allows us to *i*) reproduce the conditions that led to the original poor parallel performance and *ii*) make changes to the program execution in order to collect additional information about its runtime behaviour. In this way, we can dynamically tune the amount and type of profiling that we do during the replay in order to obtain high-level profiling information without changing the runtime behaviour of the original program.

In particular, this paper presents the following novel research contributions:

– We describe the implementation of execution replay for the parallel programs written in the pure lazy functional language Haskell. In particular, we describe the smallest set of events from a program execution that needs to be recorded in order to reproduce the parallel behaviour of these programs.
– We present a novel simulator that was built to replay the program execution using these events.
– We discuss how this technique can be used for performance debugging of Parallel Haskell programs.
– We present a use case, where execution replay is used to discover the performance bottleneck of the list-sorting algorithm quicksort.

2 Why Is Parallel Functional Programming Hard?

The lack of explicit program flow, and the fact that a lot of things happen implicitly during the program execution, is both a blessing and a curse for parallel functional programmers, especially in a lazy language like Haskell. While it is arguably easier to write parallel programs in Haskell than in imperative languages (the programmer "just" needs to insert simple parallel annotations in the appropriate places in his code), discovering performance bottlenecks of such programs can be daunting. There is a large number of things that can go wrong, and for which the programmer does not have explicit control. Consider, for example, a simple parallel implementation of the quicksort algorithm:

```
psort :: Int → [Int] → [Int]
psort _        [] = []
psort parLevel (x : xs)
    | parLevel > 0 = hiSorted `par` loSorted `pseq` (loSorted ++ x : hiSorted)
    | otherwise    = seqSort (x : xs)
  where (lo, hi)   = partition (<x) xs
        loSorted = psort (parLevel − 1) lo
        hiSorted = force (psort (parLevel − 1) hi)
```

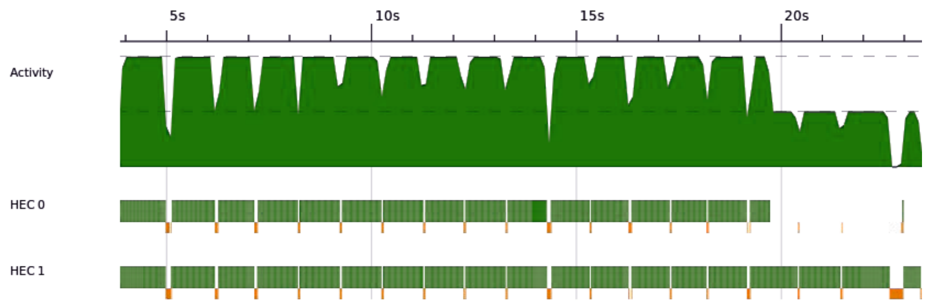


Fig. 1. ThreadScope profile of *psort*

The rationale behind this attempt at parallelising quicksort is simple: after dividing the initial list l into its lower and higher parts (lo and hi) by using x as a pivot, we try to sort these two parts in parallel using the *par* combinator. Because of how laziness works, we use the function *force*[1] to make sure that each parallel thread completely evaluates its sublist. In addition, by using the *parLevel* parameter, we control the amount of parallelism generated, so that after a certain point is reached in the recursion depth, the higher and lower parts of the list are sorted sequentially. In this way, we can tune the parallelism to get a small number of coarse-grained parallel threads. However, no matter what value for *parLevel* we choose, the speedups of *psort* that we obtain are very poor, not even achieving a speedup of 2 in up to 8 cores.

In order to understand why this program gives a bad speedup, we can try to use ThreadScope to visualise what happens during its execution. Figure 1 shows an execution profile of the program. This shows a high-level overview of the thread activity on both cores. The solid rectangles indicate that a thread is running, the little marks in between and the rectangles that produce gaps in the *activity* area indicate garbage collection. Blank space indicates that the core is idle. We can zoom in on specific parts of the execution and obtain low-level information such as individual thread identifiers, some information about thread blockage, or garbage collection requests.

From the profile above, we can observe that the program behaves reasonably well most of the execution. Then, towards the end, there is some serialisation where only one thread at a time is doing evaluation. However, ThreadScope does not provide us any hints about where do these problems come from, e.g. what part of the program is responsible for the final sequential phase. Based on the knowledge of the runtime behaviour of the language, we can speculate that the serialisation comes from the linear behaviour of the $+\!\!+$ operator, which traverses both lists sequentially. However, we cannot know for sure.

As we can see from this example, even though the information that we obtain using ThreadScope is valuable, it is too low-level to allow for proper

[1] $force :: NFData\ a \Rightarrow a \rightarrow a$ returns its argument after forcing its evaluation to normal form.

understanding of the parallel performance of the program. What is really needed is much more detailed, high-level information about the runtime behaviour, ideally linking the parallel events from the ThreadScope profile to the source expresions they are related to. In the example above, knowing which expressions are being evaluated in the final sequential phase would be a first step into fixing the performance problems of this program. We come back to this problem in Section 5.

Obtaining the information required for performance debugging using existing infrastructure and tools would require either recording a huge amount of additional information, which could then be processed offline, or rerunning the program multiple times with different profiling options enabled. In both cases, the runtime behaviour (such as scheduling and communication between threads) of the original program might change, making the profiling data useless and making it very hard to reproduce the problem that is being debugged. Our solution to the problem is to reproduce the original execution of the program without changing its runtime behaviour while, at the same time, dynamically adjust the level and type of profiling information that is gathered during the execution. We achieved this by using execution replay.

3 Execution Replay for Parallel Haskell

Execution replay [14,2] is a debugging technique in which a programmer records the execution trace of a program and then uses that trace to replay it step by step. The trace of the program encapsulates the whole state of the system as it changes throughout the execution. When replaying, the programmer is able to inspect the state of the program (e.g. variables, registers, stack) as it was at each step of the original execution.

Execution replay consists of two distinct phases (see Figure 2):

– a *logging phase* where, during the original program execution, enough information is logged so that the execution can be replayed and
– a *replaying phase* where the original program execution is replayed, using the logged information.

Our main goal is to use this technique to investigate performance bottlenecks of parallel programs, written in purely functional programming languages. This has an important consequence in that we do not have to be concerned with replaying *exactly the same* execution as the original one. It will suffice if the replayed execution is “similar enough” to the original one, such that both have the same parallel behaviour. We can, therefore, see a replay as a simulation of the execution where the threads created and the interactions between them are the same as in the original execution, and where other details of the execution may differ. This flexibility allows us to introduce changes in the program which will enable us to gather data needed for debugging its parallel performance. It also means that the amount of information that we need to record is significantly smaller than if we want to do a full replay. In the next section, we discuss exactly what events we need to record in the logging phase.

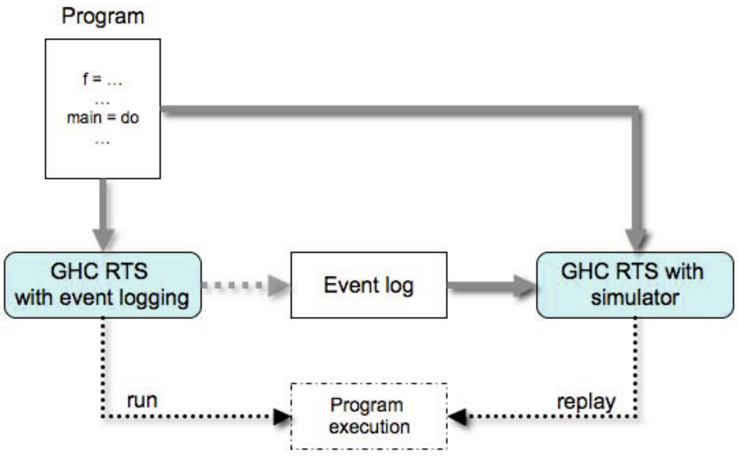


Fig. 2. Execution replay in Parallel Haskell

We have built a prototype implementation of this modified execution replay in the Glasgow Haskell Compiler (GHC) and runtime system for Parallel Haskell [13][2]. Currently, logging of the events works by running a program under GHC with event logging support[3]. For replaying, we use the same compiled executable, with the `--replay` command line flag. This runs the simulator inside the GHC runtime system, which reads the events recorded in the logging phase and sequentially simulates the program execution. In Section 4 we provide additional details of our implementation.

3.1 Events Needed for Replay

Execution replay relies on the amount of recorded information in the logging phase being tractable. Usually, most of the program execution consists in running code with a deterministic runtime behaviour, which can be replayed just by re-running it. With the introduction of mutation, parallelism and non-deterministic data sources (e.g. random numbers, I/O, signals), the execution path (and, hence, the ordering of certain events in the program execution) can change. If the program execution is to be reproduced, all the events that introduce non-determinism in the program execution need to be recorded. In imperative languages, the biggest problem is to track the mutation of data, which may be shared between different threads at any time, and this may require every access to shared memory to be logged. Current mechanisms for doing this efficiently rely on very elaborated protocols of page ownership tracking at the operating system level [7].

[2] Its development can be followed on `http://github.com/hferreiro/ghc`

[3] Using the flag `-eventlog` to compile and the runtime system flag `-ls` when executing the program.

In a pure and lazy functional language, on the other hand, the situation is much simpler. The interactions between threads (which are the main reason for the existence of different execution paths in parallel programs) are greatly simplified. Data dependencies between threads are handled transparently, without the use of locks and other synchronising mechanisms. Once a computation has been evaluated to normal form, then the runtime system enforces read-only access. Furthermore, due to laziness, any unevaluated data will be updated by at most one thread.

For simplicity, we consider only pure computations (i.e. those that do not use any side-effects, such as I/O and concurrent data mutation). Also, as mentioned earlier, we are focusing on parallel profiling, which means that we are only interested in each thread's progress and the coordination between threads. Given this, for Parallel Haskell programs there is only a small number of events that needs to be recorded:

– thread interactions: *thread run*, *thread stop*, *thread block* (when a thread is blocked on some data being evaluated by some other thread);
– scheduling events: *thread migrate* (when threads are migrated between cores), *new spark* (for creation of parallelism), *steal spark* (load balancing event), *run spark* (when a parallel expression is picked by its owner thread);
– task related: *acquire capability*, *release capability* (to track ownership of capabilities).

Besides these, there are also some additional events very specific to the internal details of the GHC runtime system. In Section 4, we give more details about the implementation of the recording of events in GHC.

3.2 Usage of Execution Replay

The key observation for our work is that we are *simulating* the previous execution of the same program, rather than rerunning it. The replay is deterministic in its runtime behaviour, and only depends on the events that were recorded in the original run of the program. This makes it ideal for performance debugging of functional programs, since gathering more profiling data does not have any impact on the ordering of the events in the replay. It might only increase the time the replay may take, which is not of great importance in debugging.

Our ultimate goal is to integrate execution replay with the ThreadScope visualisation tool. In that way, we would have a GUI tool that would enable us to pause the replay at the points where the parallel performance starts to degrade, and then turn on the appropriate kind of profiling that would enable us to get a better insight into the problems encountered. In Section 5, we show a worked example of using execution replay to debug a non-trivial parallel program (quicksort). We now discuss a few hypothetical use cases of such a tool:

– For parallel programs that perform badly due to a large amount of unevaluated data shared between threads, which is reflected in frequent blocking of threads, we can replay the program execution without any profiling data up to the point where blocking starts to occur, then turn on profiling to investigate what data are the threads blocking on. At this point, we could take advantage of cost centre profiling to link the heap data to the expressions in the source code to which they relate, so that we can find out where exactly in the program source do these data hotspots come from. We may then rewrite the original program to avoid sharing at these particular points.

– In large parallel programs, we might be interested in different profiling data during different stages of the execution. In some stages, we might be only interested in the granularity of the threads created from sparks, in others, we might be interested in discovering what data is being garbage collected. The possibility of dynamically adjusting the type and level of profiling detail during replay is one of the main motivations for using execution replay.
– For some parallel programs, there may be very subtle bugs which produce one bad execution out of many. It is not very useful to have to rerun your program many times until you reproduce a pathological behaviour. By using execution replay, the only requirement is to have a trace of the target execution. Then, it can be replayed as many times as needed with the confidence that the same wrong behaviour is being analysed as in the original run.

4 GHC Implementation Details

Although the result of a Parallel Haskell program is deterministic, its runtime behaviour might not be. In this section, we describe some of the internals of the GHC runtime system, focusing on the parts that contribute to the non-deterministic behaviour of application execution[4]. We then describe in more detail how we implemented the logging and replaying phases of execution replay in GHC.

4.1 The GHC Runtime System

The Glasgow Haskell Compiler is a state-of-the-art compiler and parallel runtime system for the pure lazy functional language Haskell [10]. It achieves great flexibility by using a lightweight thread model, where multiple logical Haskell threads are mapped into one single OS thread which runs concurrently with others (see Figure 3). The whole runtime system is organised in three layers of abstraction: capabilities, tasks and threads. A *capability* is a virtual core in which Haskell code is run. Each time a new thread is created at the Haskell level, it will be appended to the run queue of its capability. To run the code of these threads, real OS threads are needed. This is the mission of *tasks*: each task corresponds to an OS thread which tries to become the owner of a capability. Once a capability has been acquired, the task will run a scheduler cycling through the capability's run queue and assigning a time slice to each Haskell thread.

Besides finishing its time slice there are other mechanisms by which a thread can lose control of the CPU: blocking on the evaluation of a shared value, a

[4] A much more complete description of GHC can be found in [11].

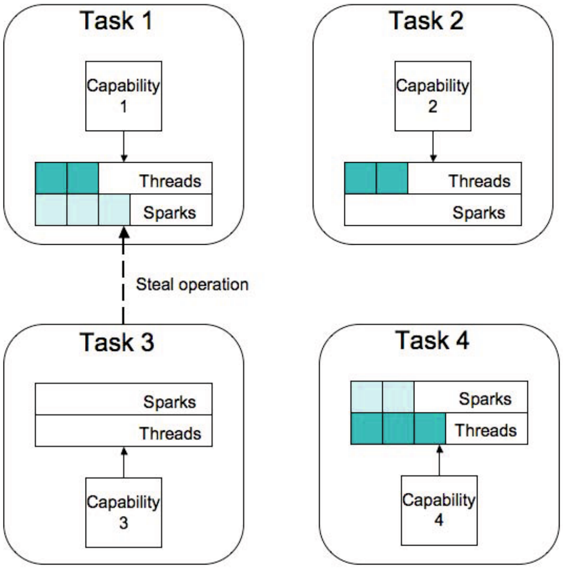


Fig. 3. Overview of the GHC Runtime System

stack or heap overflow, or exceptions. Additionally, threads can migrate to idle capabilities to increase parallelism.

The basic primitives for parallel programming are *par a b* and *pseq a b*[5]. *par* denotes that it would be useful to evaluate its first parameter in parallel to the second which is returned as result [15]. *pseq* makes it possible to order the evaluation of two expressions by ensuring that its first parameter is evaluated to weak head normal form before returning its second parameter [11]. When a thread evaluates the expression *p*‘*par*‘ *q*, a *spark* for *p* is created, and the thread continues with the evaluation of *q*. Sparks are just pointers to the part of the graph that represents the source expression. They are kept in *spark pools*, with one spark pool per capability. Each spark is eventually converted into a thread, or discarded if the expression it points to is already under evaluation, or not needed at all. Load balancing across capabilities is done using a combination of work stealing and work pushing. Idle capabilities attempt to steal sparks from the busy ones, and the capabilites that have enough active threads push them to idle capabilities.

4.2 Logging Phase

From the discussion above, we identified the set of events related to the possible non-determinism in the execution of parallel programs under GHC that we need to record. A mechanism for event logging already exists in GHC [6],

[5] Not to be confused with *seq* which is strict in both of its arguments but does not enforce an ordering in its evaluation [11].

and it supports logging of some basic events that are used for visualisation with ThreadScope. We have significantly enhanced the logging mechanism, adding several new events and changing some of the existing ones. We can group the needed events by the parts of the runtime system they are related to:

Thread scheduling. Logical Haskell threads executed on the same capability are scheduled in a round-robin fashion. Each thread runs in a capability until one of three things occur:

- the thread runs out of heap or stack space (in the first case garbage collection needs to be performed before any thread can continue evaluation);
- the thread blocks on some expression being evaluated; or
- the predefined time slice expires.

In all of the above cases, the thread is preempted and the next thread in the queue is selected for evaluation. If we want to replay how threads are interleaved, we need to be able to tell how much evaluation a thread has done in a given time slice. This amount of work changes in different executions because of how modern CPUs work. Thankfully, GHC preempts threads only when they make a heap check. The already existing *thread run* event already provides us with the data to identify which thread began running in a capability. We then modified the *thread stop* event to additionally store the amount of allocation the thread did in its time slice. For the case in which a thread blocks on an expression being evaluated by another thread, we enhanced the *thread block* event by adding information about the threads involved. A *thread wakeup* event is recorded when its execution can be resumed.

Load balancing. As a consequence of the previous discussion, the number of sparks created in the same time period in different program executions can be different, and also the times at which capabilities become idle (and, therefore, the need to perform spark stealing or thread pushing) can be different. This means that we need to record the events related to spark creation and migration, and also events related to threads being pushed to capabilities. We, therefore, need *new spark* (that occurs when a new spark is created), *spark steal* (that occurs when a capability steals a spark) and *thread migrate* (that occurs when a thread is migrated between capabilities) events to be logged. For these events, we need to record exactly which capability became idle, which spark it stole from which capability, or which thread was pushed to it.

Capability ownership. The task-related events that were described before, *acquire capability* and *release capability*, are new events we added so that we could track which task was responsible for the execution of threads in a capability.

An excerpt from the trace of the quicksort implementation described in Section 2 is shown below:

```
...
4177926000: cap 1: stopping thread 4 (stack overflow) (96 words allocated)
4177940000: cap 1: running thread 4
4180949000: cap 1: stopping thread 4 (heap overflow) (65024 words
 / allocated)
4180979000: cap 0: stopping thread 3 (blocked on blackhole owned by
 / thread 4) (25253 words alloced)
4181027000: cap 0: task 1 releasing
4181146000: cap 1: running thread 4
...
```

4.3 Replay Phase

The replaying phase works by spawning an independent *scheduler thread* at the beginning of the program execution. This thread initialises the runtime system and makes sure that all tasks stop after being created. Then, in a loop, it reads the recorded events ordered by time. If the event is *thread run* or *thread wakeup*, the scheduler thread checks the capability responsible for the event, and allows the corresponding thread to progress until it is stopped (once it has done the same amount of work as in the recorded execution) or blocked. The rest of the events are needed to preserve the ordering between the threads that emit conflicting events (the same spark trying to be stolen by different threads, etc.). Respecting this ordering will allow the execution to be replayed without trouble.

5 Use Case: Why Is Parallel Quicksort so Slow?

In order to show how execution replay can be used for performance debugging of non-trivial parallel programs, we come back to the quicksort example we presented in Section 2. Quicksort has gathered a lot of attention recently in teaching parallel functional programming at several universities [4], since it is an example of a program which "seems" rather trivial to parallelise, yet for which obtaining good speedups (especially using lazy languages) is quite challenging.

A high-level, integrated profiling tool, designed with the use cases detailed in Section 3.2 in mind, is still work in progress, so for this example we show how to use execution replay in conjuction with a custom low-level tool for annotating the source code.

We saw in Section 2 that the obvious method of parallelising this program does not work as expected. In order to come up with a better parallel program, we first made some optimisations to the sequential version. We implemented our own strict version of the *partition* function so that we could avoid the overhead of lazy evaluation caused by computing the sublists on demand. Next, we got rid of the append operator $++$, which requires multiple traversals of the same lists when it is applied left-recursively, as in our case. For this, we used an accumulator in which the resulting list is being constructed. First, we start with the whole list to be sorted and an empty accumulator. Then, at each recursive step, the pivot is accumulated into the sorted higher sublist. When there are no more elements to sort, the accumulator is returned as the fully sorted list.

No. cores	Speedup
2	1.49
4	0.78
8	0.60

Fig. 4. Speedups of *psort1*

```
qsort :: [Int] → [Int]
qsort xs = seqSort xs [ ]
  where seqSort [ ]      zs = zs
        seqSort (x : xs) zs = seqSort lo (x : seqSort hi zs)
          where (lo, hi) = partition x xs
partition :: Int → [Int] → ([Int], [Int])
partition x xs = go xs [ ] [ ]
  where go [ ]      ts fs = (ts, fs)
        go (y : ys) ts fs
          | y < x       = go ys (y : ts) fs
          | otherwise   = go ys ts       (y : fs)
```

Similarly to the first time, we tried to naively parallelise this code in the same way as we did in Section 2. Given the changes mentioned above, we expected to avoid the sequential phase that occurs at the end of the execution.

```
psort1 :: Int → [Int] → [Int]
psort1 n xs = go n xs [ ]
  where go _ [ ]      zs = zs
        go n (x : xs) zs
          | n > 0       = r `par` go (n − 1) lo (x : r)
          | otherwise   = seqSort (x : xs) zs
          where r = force (go (n − 1) hi zs)
                (lo, hi) = partition x xs
```

We measured the speedups of this program on a machine with two Intel Xeon 2.93GHz CPUs, each of them having four cores. Each CPU had 8MB of L2 cache, that was shared between all of its cores. The total amount of RAM was 64GB. In the speedups figure above, we took the mean time over five runs of each program with the same input, a list consisting of 10 million elements.

Figure 4 shows the speedups against the sequential version of the algorithm, measured by using from two up to eight cores. From the figure, we can observe that we are actually getting significant slowdowns as we use more cores.

In order to debug the performance of this program, we used again ThreadScope to get an overview of the thread activity. Figure 5 shows the profile from ThreadScope after running our program using two cores.

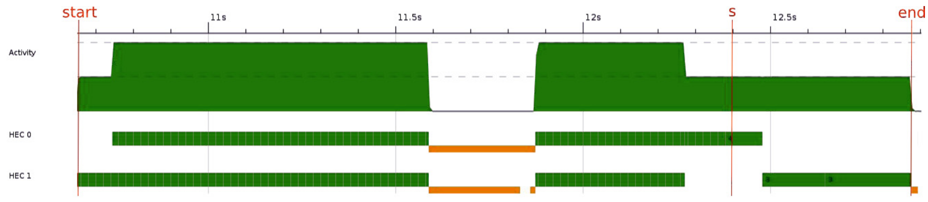


Fig. 5. ThreadScope profile of *psort1*

We can see that we still have the same serialisation problem that we had in the initial parallel version in Section 2, and that getting rid of the ++ operator did not help at all. Additionally, there is a pause in the execution corresponding to a major garbage collection phase. The big amount of input data, coupled with the fact that we set up a large allocation area, is responsible for this behaviour. *psort* does not present this gap because, due to its inefficient sequential implementation, we had to provide a much smaller input list.

We now used execution replay to discover which part of the program is responsible for the sequential phase at the end of the execution. We developed some custom tools to be able to register timestamps when the evaluation of an annotated expression is finished and to analyse the output produced. By using execution replay, we were sure that the same execution was reproduced and so that the output data matched the original ThreadScope profile. To focus on the interesting parts of the program, we added two checkpoints: `start`, which is the point after reading the input list, and `end` which marks the end of the program (see Figure 5). We then replayed the program and processed its output to obtain the following report:

```
  188.020 (   93.914) cap 0: partition [10.837.539]
1.240.278 (1.052.258) cap 0:   seqSort [11.889.797]
1.747.763 (  507.485) cap 0:   seqSort [12.397.282]
1.747.766 (        3) cap 0:     force [12.397.285]
1.828.627 (   80.861) cap 0:     force [12.478.146]

        0 (        0) cap 1:     start [10.649.519]
   94.106 (   94.106) cap 1: partition [10.743.625]
  170.970 (   76.864) cap 1: partition [10.820.489]
  732.638 (  561.668) cap 1:   seqSort [11.382.157]
1.621.573 (  888.935) cap 1:   seqSort [12.271.092]
1.996.394 (  374.821) cap 1:     force [12.645.913]
2.225.849 (  229.455) cap 1:     force [12.875.368]
2.225.852 (        3) cap 1:       end [12.875.371]
```

Each line shows the timestamps for the completion of each annotated function in the program. First, the relative time against `start` is presented. Next, the relative time against the previous function timestamp and the absolute timestamp are shown in brackets. Each event is classified according to its capability.

The relevant aspect of this data is that the sorting process has finished by the time the sequential phase begins. We can see this because the timestamp of the finish time of the last call to the *seqSort* function on capability 0 is 12.397s (checkpoint `s` in Figure 5) and, from the ThreadScope profile, we can observe that the sequential phase starts at a timestamp around 12.3s. After the checkpoint `s`, only the timestamps of the *force* functions are left. So, the sequential phase at the end must correspond to the execution of these functions.

The conclusion is that the program execution is almost perfectly balanced between the two cores while the parallel threads are sorting their parts of the list (the timestamps for the completions of the calls to *seqSort* are similar in each capability). But then, because of the *force* call, each thread needs to traverse the sublist that is passed in the accumulator *zs*. This sublist is sorted by another thread, so the thread evaluating the *force* call becomes blocked immediately, waiting until *zs* has been evaluated. Only then can it finish traversing it. When finished, this thread returns the sorted list and allows its parent thread to also finish evaluating its *force* call. This linear process gets worse as more threads are involved in it. This is the reason why the speedups get worse as we add more cores.

In the end, the same behaviour that we tried to prevent by avoiding the $+\!\!+$ *operator, i.e. sequential traversal of the sorted list, is reproduced by evaluating to normal form each of the sublists!*

This analysis suggests that the way to fix this behaviour is to replace the function *force* with a function that would immediately return when the tail of the list being forced is already in normal form. To this end, we implemented a custom version of quicksort which operates on a datatype *List a* (instead of a regular list) as its input. This new type has the same *Nil*/[ ] and *Cons*/: constructors as regular lists, and also an additional constructor *Done*. The *Done* constructor has a list of elements as argument, and is used to mark the list as fully evaluated. Together with this new type, we introduced a $toList :: List\ a \to [a]$ function which takes a *List a* as input and returns its corresponding regular list in normal form. Its behaviour is similar to our usage of *force*, with the exception that it terminates if a *Done xs* element is found:

```
data List a = Nil | Cons a (List a) | Done [a]
toList :: List a → [a]
toList Nil          = []
toList (Cons x xs) = let xs' = toList xs
                     in x `seq` xs' `seq` x : xs'
toList (Done xs)   = xs
```

Now, by making use of the former definitions, we can implement a version of *psort1* in which the threads evaluating the higher half of the list, *hi*, will mark it as already evaluated, so that the ones sorting the other half will find a *Done xs* value and directly return *xs* instead of traversing it again:

No. cores	*psort*	*psort1*	*psort2*
2	1.69	1.49	1.90
4	1.71	0.78	2.35
8	1.51	0.60	2.75

Fig. 6. Speedups of the different parallel versions of quicksort

```
psort2 :: Int → [Int] → [Int]
psort2 n xs = toList (go n xs Nil)
   where go _ []        zs = zs
         go n (x : xs) zs
            | n > 0      = r `par` go (n − 1) lo (Cons x r)
            | otherwise  = seqSort (x : xs) zs
            where r = Done $! toList (go (n − 1) hi zs)
                  (lo, hi) = partition x xs
```

The speedups for *psort2* are shown in Figure 6. We can observe much better speedups than for *psort1*. For two cores, the speedup is almost linear. When we add more cores, speedup is further increased, but the relative performance is decreased. This can be attributed to the fact that each thread is created only after the list has been partitioned. The same thing will happen to the next threads once the generated sublist are partitioned again. So, if we need to use more threads, it will take longer to create them, increasing the initial sequential phase.

6 Related Work

Previous approaches to performance profiling of Parallel Haskell programs involve the use of simulators such as GranSim [9] or parallel cost centre profiling [3]. GranSim was developed as an instrumentation of the GHC runtime system that allowed the programmer to gather statistics of the program which was simulated to run in a distributed machine with a customizable environment (e.g. network delay). Events could be visualised in a similar way to ThreadScope. The same event log format was used by the parallel profiler for the GUM parallel implementation [16]. Similar techniques are used by the more recent ThreadScope [6] and EdenTV [1] visualisers.

Our approach enhances profiling by using a kind of simulated environment, which, in contrast to GranSim, does not emulate any real hardware but replays a previous run. This technique is known as *execution replay* [14]. So far, it has been used almost exclusively for debugging instead of performance profiling. Most execution replay systems allow any program to be replayed without recompilation [7]. The most difficult problem these systems have to solve is that of shared memory interactions, something we can completely ignore because our source code is purely functional. In addition, some of these systems also try to replay the scheduling of threads (a requirement in our case), but they do so by using hardware counters [5,8], which makes them hardware dependent and subject to inaccurate measurements [12,17].

7 Conclusions and Future Work

In this paper, we described a prototype implementation of the execution replay mechanism in the GHC compiler for Parallel Haskell. We also described how this mechanism can be used to obtain a better insight of the parallel behaviour of functional programs, which makes it very useful for performance debugging of such programs. We have presented a use case of execution replay for parallel debugging, using a parallelisation of the well-known quicksort algorithm as an example. Whe showed that, despite quicksort being a program which seems easy to parallelise, it contains a number of hidden caveats that make obtaining good speedups quite challenging. Hence, being able to obtain better profiling information is vital in order to understand its behaviour and discover the bottlenecks.

This paper presents the first implementation of the execution replay mechanism in the context of a lazy functional language. In addition, this is the first time execution replay is used for performance debugging. Our focus on pure functional languages and on parallel performance debugging significantly relaxed the assumptions that we need to make about the replay. We are not restricted to having to reproduce *exactly the same execution* as the original one. The replayed execution can differ from the original one, as long as they both have the same parallel behaviour. This significantly reduces the amount of logging information that is required for replay, making it much less expensive that when used in imperative languages for replaying the exact state of the program at each point of its execution. It also allows dynamic enabling and disabling of data gathering modules during the replay.

With execution replay as a foundation, we are able to build better profiling tools which will allow functional programmers to better understand and fix many parallel programs for which there were no tools to deal with. In the future, we plan to implement these tools by integrating already existing profiling and visualisation approaches (such as ThreadScope and cost centre profiling) with execution replay. We are also in the process of extending execution replay for programs with side-effects. Finally, we plan to demonstrate the effectiveness of replay-driven performance debugging on a larger set of parallel programs.

Acknowledgments. This work is supported by European Union grant SCIEnce (RII3-CT-026133), EPSRC grant HPC-GAP (EP/G 055181) and MICINN grant TIN2010-20959.

References

1. Berthold, J., Loogen, R.: Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer. In: Parallel Computing: Architectures, Algorithms and Applications. Advances in Parallel Computing, vol. 15, pp. 121–128. IOS Press (2008)

2. Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., De Bosschere, K.: A Taxonomy of Execution Replay Systems. In: Proc. of the Intl. Conf. on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet (2003)
3. Hammond, K., Loidl, H.W., Trinder, P.: Parallel Cost Centre Profiling. In: Proc. of the Glasgow Workshop on Functional Programming, Ullapool, Scotland (1997)
4. Hughes, J., Sheeran, M.: Teaching Parallel Functional Programming at Chalmers. In: Draft Proc. TFPIE 2012 (2012)
5. Itskova, E.: Echo: A Deterministic Record/Replay Framework for Debugging Multithreaded Applications. Master's thesis, Imperial College, London (2006)
6. Jones Jr., D., Marlow, S., Singh, S.: Parallel Performance Tuning for Haskell. In: Proc. Haskell 2009, pp. 81–92. ACM (2009)
7. Laadan, O., Viennot, N., Nieh, J.: Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In: Proc. SIGMETRICS 2010, pp. 155–166. ACM (2010)
8. Lee, D., Said, M., Narayanasamy, S., Yang, Z., Pereira, C.: Offline symbolic analysis for multi-processor execution replay. In: Proc. MICRO, vol. 42, pp. 564–575. ACM (2009)
9. Loidl, H.W.: Granularity in Large-Scale Parallel Functional Programming. Ph.D. thesis, Department of Computing Science, University of Glasgow (1998)
10. Marlow, S.: Haskell 2010. Language Report (2010), `http://www.haskell.org/onlinereport/haskell2010`
11. Marlow, S., Peyton Jones, S., Singh, S.: Runtime Support for Multicore Haskell. In: Proc. ICFP 2009, pp. 65–78. ACM (2009)
12. Mathur, W., Cook, J.: Toward Accurate Performance Evaluation using Hardware Counters. In: Proc. of the Applications for a Changing World, ITEA Modeling & Simulation Workshop (2003)
13. Peyton Jones, S.L., Hall, C.V., Hammond, K., Partain, W., Wadler, P.: The Glasgow Haskell compiler: A Technical Overview. In: Proc. UK Joint Framework for Information Technology (JFIT) Technical Conf. (1993)
14. Ronsse, M., De Bosschere, K., Chassin de Kergommeaux, J.: Execution Replay and Debugging. arXiv:cs/0011006 (2000)
15. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.: Algorithms + Strategy = Parallelism. Journal of Functional Programming 8(1), 23–60 (1998)
16. Trinder, P.W., Hammond, K., Mattson Jr., J.S., Partridge, A.S., Peyton Jones, S.: GUM: A Portable Parallel Implementation of Haskell. In: Proc. PLDI 1996, pp. 79–88. ACM (1996)
17. Zaparanuks, D., Jovic, M., Hauswirth, M.: Accuracy of performance counter measurements. In: Proc. ISPASS 2009, pp. 23–32. IEEE (2009)

Supervised Workpools for Reliable Massively Parallel Computing

Robert Stewart, Phil Trinder, and Patrick Maier

Heriot Watt University
School of Mathematical and Computer Sciences
Edinburgh, UK
{R.Stewart,P.W.Trinder,P.Maier}@hw.ac.uk

Abstract. The manycore revolution is steadily increasing the performance and size of massively parallel systems, to the point where system reliability becomes a pressing concern. Therefore, massively parallel compute jobs must be able to tolerate failures. For example, in the HPC-GAP project we aim to coordinate symbolic computations in architectures with 10^6 cores. At that scale, failures are a real issue. Functional languages are well known for advantages both for parallelism and for reliability, e.g. stateless computations can be scheduled and replicated freely.

This paper presents a software level reliability mechanism, namely supervised fault tolerant workpools implemented in a Haskell DSL for parallel programming on distributed memory architectures. The workpool hides task scheduling, failure detection and task replication from the programmer. To the best of our knowledge, this is a novel construct. We demonstrate how to abstract over supervised workpools by providing fault tolerant instances of existing algorithmic skeletons. We evaluate the runtime performance of these skeletons both in the presence and absence of faults, and report low supervision overheads.

Keywords: Fault tolerance, workpools, parallel computing, Haskell.

1 Introduction

Changes in chip manufacturing technology is leading to architectures where the number of cores grow exponentially, following Moore's law. Many predict the proliferation of massively parallel systems currently exemplified by the large commodity off-the-shelf (COTS) clusters used in commercial data centres, or the high performance computing (HPC) platforms used in scientific computing. For example, over the last 4 years, the performance (measured in FLOPS) of the world's fastest supercomputer has risen 16-fold, according to TOP500[1]. This has been accompanied by a 13-fold increase in the total number of cores, and by power consumption more than tripling (from 2.3 to 7.9 MW). The latter trend in particular points to an ever increasing size of these systems, not only in terms

[1] http://www.top500.org

of total number of cores, but also in the number of networked components (i.e. compute nodes and network switches).

Even with failure rates of individual components appearing negligible, the exponential growth in the number of components makes system reliability a growing concern [21]. Thus, massively parallel compute jobs either must complete quickly, or be able to tolerate component failures.

Depending on the problem domain, there may be several ways to react to failures. For instance, stochastic simulations can trade precision for reliability by simply discarding computations on failed nodes. Similarly, grid-based continuous simulations can recover information lost due to a node failure by "smoothing" information from its neighbours. However, there are problem domains, e.g. optimisation or symbolic computation, where trading precision is impossible because solutions are necessarily exact. In these domains, fault tolerance is more costly, as it can only be achieved by replicating the computations of failed nodes, which incurs overheads and requires book keeping. Nonetheless, replication-based fault tolerance is widely used, e.g. in MapReduce frameworks like Hadoop [2].

Functional languages have long been advocated as particularly suitable for parallel programming because they encourage a stateless coding style which simplifies the scheduling of parallelism. Furthermore statelessness also simplifies task replication, making functional languages even more attractive for massively parallel programming, where replication-based fault tolerance is indispensable.

This paper presents the design and implementation of a fault tolerant workpool in a functional language. A *workpool* is a well-known parallel programming construct that guarantees the parallel execution of independent *tasks*, relieving the programmer of concerns about task scheduling (and sometimes load balancing). A *fault tolerant* workpool additionally guarantees completion of all tasks (under some proviso), thus relieving the programmer of concerns about detecting node failures, replicating tasks, and the associated book keeping. We note that our workpool is able to recover from the failure of any number of nodes bar a single distinguished one, the node hosting the supervised workpool. Therefore, the workpool is not *high-availability*, setting it apart from high-availability behaviours in Erlang. However, our workpool is able to guarantee that system reliability matches the reliability of a single node, which is good enough for most massively parallel applications.

The workpool is implemented on top of *HdpH* [17], a Haskell domain specific language (DSL) for distributed-memory parallel programming. HdpH and the workpool are being developed within the HPC-GAP project[2]. The project aims to solve large computer algebra problems on massively parallel platforms like HECToR, the UK national supercomputing service with currently 90,000 cores, by coupling the *GAP* computer algebra system [11] with the *SymGridParII* coordination middleware [16].

We start by surveying fault tolerant approaches, languages and frameworks (Section 2), before making the following contributions:

[2] http://www-circa.mcs.st-andrews.ac.uk/hpcgap.php

1. We present the design and implementation of a novel fault-tolerant workpool in Haskell (Sections 3.1 and 3.3), hiding task scheduling, failure detection and task replication from the programmer. Moreover, workpools can be nested to form fault-tolerant hierarchies, which is essential for scaling up to massively parallel platforms like HECToR.
2. The implementation of high-level fault tolerant abstractions on top of the workpool: generic fault tolerant skeletons for task parallelism and nested parallelism, respectively (Section 4).
3. We evaluate the fault tolerant skeletons on two benchmarks. These benchmarks demonstrate fault tolerance: computations do complete even in the presence of node failures. We also measure the overheads of the fault tolerant skeletons, both in terms of the cost of book keeping, and in terms of the time to recover from failure (Section 5).

2 Related Work

Most existing fault tolerant approaches in distributed architectures follow a rollback-recovery approach, and new opportunities are being explored as alternative and more scalable possibilities. This section outlines non-language and language based approaches to fault tolerance.

2.1 Non Language-Based Approaches to Fault Tolerance

At the highest level, algorithmic methods have been proposed, with the injection of fault oblivious algorithms and self stabilising algorithms [5]. Various techniques have been used at the application level, such as reflective object-oriented programming [9].

The *Message Passing Interface* [13] is a predominant and efficient communication layer in HPC platforms. Thorough comparisons of fault tolerant MPI approaches and implementations have been made [12]. These include checkpointing the state of computation [4], extending or modifying the semantics of the MPI standard [10], and runtime resilience to overcome node failure [7], though the onus is on the user to handle faults programmatically.

On COTS platforms, computational frameworks such as MapReduce realise fault tolerance through replication, as implementations such as Hadoop [2] have shown. They are optimised for high throughput, but limit the programmer to one parallel pattern. In contrast, our supervised workpool is a fault tolerant construct that can be used for multiple parallel patterns, as described in Section 4.

2.2 Fault Tolerance in Erlang

Erlang [1] is a dynamically typed functional language designed for programming concurrent, real-time, distributed fault tolerant programs. Erlang provides a process-based model of concurrency with asynchronous message passing. Erlang processes do not shared memory, and all interaction is done through message passing.

Erlang has three mechanisms that provide fault tolerance in the face of failures [14]: monitoring the evaluation of expressions; monitoring the behaviour of other processes; and trapping evaluation errors of undefined functions. Additionally, Erlang provides primitives for creating and deleting links between processes. Process linking is symmetrical, and exit signals can be propagated up hierarchical supervision trees.

On top of these primitives, Erlang provides the Open Telecom Platform (OTP) which separates the Erlang framework from the application, and includes a set of fault tolerant *behaviours*. The Erlang/OTP *generic supervisor* behaviour provides fault tolerance for Erlang programs executing on multiple compute nodes. Processes residing in the Erlang VM inherit either the role of *supervisor*, or alternatively, a *child*. Supervisors receive exit signals from children, taking the appropriate action. They are responsible for starting, stopping, and monitoring child processes, and keep child processes alive by restarting them if necessary. Supervisors can be nested, creating hierarchical supervision trees.

In contrast to Erlang behaviours, which are designed for distributed computing, our supervised workpool is designed for distributed-memory *parallel* programming. Additionally, statically typed polymorphic skeletons can be constructed on top of the workpool.

2.3 Fault Tolerance in Distributed Haskell Extensions

Cloud Haskell. Cloud Haskell [8] is a domain specific language for distributed-memory computing platforms. It is implemented as a shallow embedding in Haskell, and provides a message communication model that is inspired by Erlang. It emulates the Erlang approaches (Section 2.2) of isolated process memory and explicit message passing, and provides process linking. Cloud Haskell inherits the language features of Haskell, including purity, types, and monads, as well as the multi-paradigm concurrency models in Haskell. A significant contribution of Cloud Haskell is a mechanism for serialising function closures, enabling higher order functions to be used in distributed computing environments. As Cloud Haskell tightly emulates Erlang, it is once again more designed for distributed rather than parallel computing.

HdpH. Haskell distributed parallel Haskell (HdpH) [17] is a distributed-memory parallel DSL for Haskell that supports high-level semi-explicit parallelism, and is designed for fault tolerance. HdpH is an amalgamation of two recent contributions to the Haskell community. Its closure serialisation and transmission over networks is inspired by Cloud Haskell (Section 2.3), and it uses the `Par` Monad [19], as a shallowly embedded DSL for parallelism. The write-once semantics of the original `Par` Monad are relaxed in HdpH slightly to support fault tolerance: we ignore successive writes rather than failing, which is described in Section 3.3.

HdpH extends the `Par` Monad for *distributed-memory* parallelism, rather than distributed systems as in Erlang or Cloud Haskell. Parallelism in the `Par` Monad is achieved with a `fork` primitive, and an `IVar` is a communication abstraction to communicate results of parallel tasks (referred to in other languages, as futures

[20] or promises [15]). HdpH extends CloudHaskell's closure representation, by supporting polymorphic closure transformations in order to implement high-level coordination abstractions. This extension is crucial for the implementation of generic fault tolerant skeletons as described in Section 4.

3 The Design and Implementation of a Supervised Workpool

3.1 Design

A workpool is an abstract control structure that takes units of work as input, returning values as output. The scheduling of the work units is coordinated by the workpool implementation. Workpools are a very common pattern, and are often used for performance, rather than fault tolerance. For example workpools can be used for limiting concurrent connections to resources, to manage heavy load on compute nodes, and to schedule critical application tasks in favour of non-critical monitoring tasks [26]. The supervised workpool presented in this paper extends the workpool pattern by adding fault tolerance to support the fault tolerant execution of HdpH applications. The fault tolerant design is inspired by the supervision behaviour and node monitoring aspects of Erlang (Section 2.2), and combines this with Haskell's polymorphic, static typing.

Most workpools schedule work units dynamically, e.g. an idle worker selects a task from the pool and executes it. For simplicity our current HdpH workpool uses static scheduling: each worker is given a fixed set of work units. The supervised workpool performs well for applications exhibiting regular parallelism, and also for limited irregular parallel programs, as shown in Section 5. A fault tolerant work stealing scheduler is left to future work (Section 6).

The Haskell implementation is made possible by the loosely coupled design in HdpH. Before describing the workpool in detail, we introduce terminology for HdpH and the workpool in Table 1.

Table 1. HdpH and workpool terminology

IVar	A write-once mutable mutable reference.
GIVar	A global reference to an IVar, which is used to remotely write values to the IVar.
Task	Consists of an *expression* and a *GIVar*. The expression is evaluated, and its value is written to the associated GIVar.
Completed task	When the associated GIVar in a *task* contains the value of the task expression.
Closure	A serializable expression or value. Tasks and values are serialized as closures, allow them to be shipped to other nodes.
Supervisor thread	The Haskell thread that has initialized the workpool.
Process	An OS process executing the GHC runtime system.
Supervising process	The *process* hosting the *supervisor thread.*
Supervising node	The node hosting the supervising process.
Worker node	Every node that has been statically assigned a task from a given workpool.

A fundamental principle in the HdpH supervised workpool is that there is a one-to-one correspondence between a *task* and an `IVar` — each task evaluates an expression to return a result which is written to its associated `IVar`. The *tasks* are distributed as closures to *worker nodes*. The *supervisor thread* is responsible for creating and globalising `IVar`s, in addition to creating the associated tasks and distributing them as closures. Here are the workpool types and the function for using it:

```
type SupervisedTasks a  = [(Closure (IO ()), IVar a)]
supervisedWorkpoolEval :: SupervisedTasks a -> [NodeId] -> IO [a]
```

The `supervisedWorkpoolEval` function takes as input a list of tuples, pairing tasks with their associated `IVar`s, and a list of `NodeId`s. The closured tasks are distributed to worker nodes in a round robin fashion to the specified worker `NodeId`s, and the workpool waits until all tasks are complete i.e. all `IVar`s are full. If a node failure is identified before tasks complete, the unevaluated tasks sent to the failed node are reallocated to the remaining available nodes. Detailed descriptions of the scheduling, node failure detection, and failure recovery is in Section 3.3.

The supervised workpool guarantees that given a list of *tasks*, it will fully evaluate their result provided that:

1. The *supervising node* is alive throughout the evaluation of all tasks in the workpool.
2. All expressions are computable. For example, evaluating an expression should not throw uncaught exceptions, such as a division by 0; all programming exceptions such as non-exhaustive case statements must be handled within the expression; and so on.

Our supervised workpool is non-deterministic, and hence is monadic. This is useful in some cases such as racing the evaluation of the same task on separate nodes, and also for fault tolerance — the write semantics of `IVar`s are described in Section 3.3. To recover determinism in the supervised workpool, expressions must be *idempotent*. An idempotent expression may be executed more than once which entails the same side effect as executing only once. E.g inserting a given key/value pair to a mutable map - consecutive inserts have no effect. Pure computations, because of their lack of side effects, are of course idempotent.

Workpools are functions and may be freely nested and composed. There is no restriction to the number of workpools hosted on a node, and Section 5.2 will present a divide-and-conquer abstraction that uses this flexibility.

3.2 Use Case Scenario

Figure 1 shows a workpool scenario where six closures are created, along with six associated `IVar`s. The closures are allocated to three worker nodes: `Node2,` `Node3` and `Node4` from the supervising node, `Node1`. Whilst these closures are being evaluated, `Node3` fails, having completed only one of its two tasks. As `IVar` `i5` had not been filled, closure `c5` is reallocated to `Node4`. No further node failures

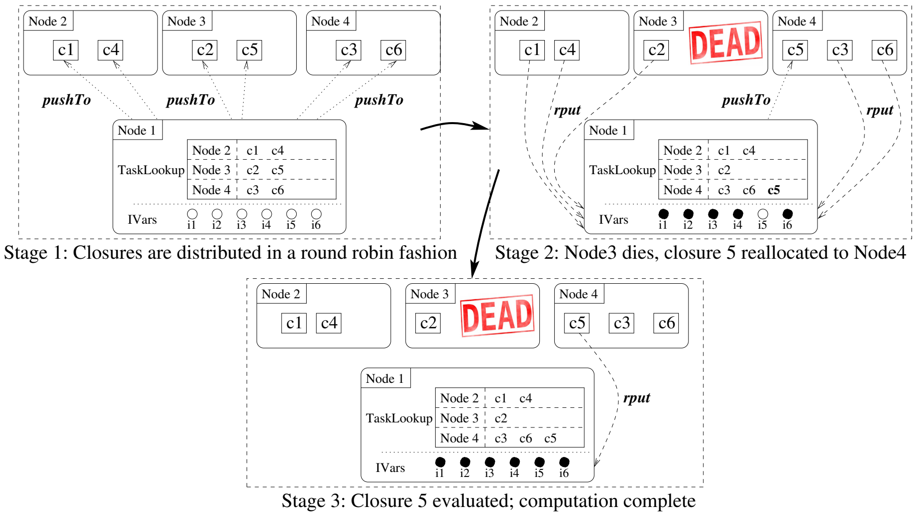

Node 2
c1
c4
Node 3
c2
c5
Node 4
c3
c6
pushTo
pushTo
pushTo
Node 1
TaskLookup
Node 2 c1 c4
Node 3 c2 c5
Node 4 c3 c6
IVars
i1 i2 i3 i4 i5 i6
Stage 1: Closures are distributed in a round robin fashion
DEAD
rput
pushTo
rput
Node 4 c3 c6 c5
Stage 2: Node3 dies, closure 5 reallocated to Node4
Stage 3: Closure 5 evaluated; computation complete

Fig. 1. Reallocating closures scenario

```
-- |HdpH primitives
type IVar a = TMVar a                                 -- type synonym for IVar
data GIVar a                                          -- global handles to IVars
data Closure a                                        -- explicit, serialisable closures
pushTo :: Closure (IO ()) -> NodeId -> IO ()          -- explicit task placement
rput   :: GIVar (Closure a) -> Closure a -> IO ()     -- write a value to a remote IVar
get    :: IVar a -> IO a                              -- blocking get on an IVar
probe  :: IVar a -> STM Bool                          -- check if IVar is full or empty
```

Fig. 2. Type signatures of HdpH primitives

occur, and once all six `IVars` are full, the supervised workpool terminates. The mechanisms for detecting node failure, for identifying completed tasks, and the reallocation of closures are described in Section 3.3.

3.3 Implementation

The types of the relevant HdpH primitives are shown in Figure 2. The complete fault tolerant workpool implementation is available [23], and the most important functions are shown in Figure 3. All line numbers refer to this Figure. Two independent phases take place in the workpool:

1. Line 5 shows the `supervisedWorkpoolEval` function which creates the workpool, distributes tasks, and then uses the STM termination check described in item 2. The `distributeTasks` function on line 9 uses `pushTo` (from Figure 2) to ship tasks to the worker nodes and creates `taskLocations`, an instance of `TaskLookup a` (line 3). This is a mutable map from `NodeIds` to `SupervisedTasks a` on line 2, which is used for the book keeping of task locations. The `monitorNodes` function on lines 22 - 30 then monitors worker node availability. Should a worker node fail, `blockWhileNodeHealthy` (line

29) exits, and `reallocateIncompleteTasks` on line 30 is used to identify incomplete tasks shipped to the failed node, using `probe` (from Figure 2). These tasks are distributed to the remaining available nodes.

2. Haskell's STM [22] is used as a termination check. For every task, an `IVar` is created. A `TVar` is created in the workpool to store a list of values returned to these `IVar`s from each task execution. The `getResult` function on line 34 runs a *blocking* `get` on each `IVar`, which then writes this value as an element to the list in the `TVar`. The `waitForResults` function on line 42 is used to keep phase 1 of the supervised workpool active until the length of the list in the `TVar` equals the number of tasks added to the workpool.

```
1   -- |Workpool types
2   type SupervisedTasks a  = [(Closure (IO ()), IVar a)]
3   type TaskLookup a       = MVar (Map NodeId (SupervisedTasks a))
4
5   supervisedWorkpoolEval :: SupervisedTasks a -> [NodeId] -> IO [a]
6   supervisedWorkpoolEval tasks nodes = do
7          -- PHASE 1
8          -- Ship the work, and create an instance of 'TaskLookup a'.
9      taskLocations <- distributeTasks tasks nodes
10         -- Monitor utilized nodes; reallocate incomplete tasks when worker nodes fail
11     monitorNodes taskLocations
12
13         -- PHASE 2
14         -- Use STM as a termination check. Until all tasks are evaluated, phase 1 remains active.
15     fullIvars <- newTVarIO []
16     mapM_ (forkIO . atomically . getResult fullIvars . snd) tasks
17     results <- atomically $ waitForResults fullIvars (length tasks)
18
19         -- Finally, return the results of the tasks
20     return results
21
22  monitorNodes :: TaskLookup a -> IO ()
23  monitorNodes taskLookup = do
24     nodes <- fmap Map.keys $ readMVar taskLookup
25     mapM_ (forkIO . monitorNode) nodes
26     where
27      monitorNode :: NodeId -> IO ()
28      monitorNode node = do
29        blockWhileNodeHealthy node -- Blocks while node is healthy (Used in Figure 4)
30        reallocateIncompleteTasks node taskLookup -- reallocate incomplete tasks shipped to 'node'
31
32  -- |Takes an IVar, runs a blocking 'get' call, and writes
33  --  the value to the list of values in a TVar
34  getResult :: TVar [a] -> IVar a -> STM ()
35  getResult values ivar = do
36      v  <- get ivar
37      vs <- readTVar values
38      writeTVar results (vs ++ [v])
39
40  -- |After each write to the TVar in 'evalTask', the length of the list
41  --  is checked. If it matches the number of tasks, STM releases the block.
42  waitForResults :: TVar [a] -> Int -> STM [a]
43  waitForResults values i = do
44      vs <- readTVar values
45      if length vs == i then return vs else retry
```

Fig. 3. Workpool implementation & Use of STM as a termination check

The restriction to idempotent tasks in the workpool (Section 3.1) enables the workpool to freely duplicate and re-distribute tasks. Idempotence is permitted by the *write* semantics of IVars. The first write to an IVar succeeds, and subsequent writes are ignored — successive rput attempts to the same IVar are non-fatal. To support this, the write-once semantics of IVars in the Par Monad [19] are relaxed slightly in HdpH, to support fault tolerance. This enables identical closures to be raced on separate nodes. Should one of the nodes fail, the other evaluates the closure and rputs the value to the associated IVar. Should the node failure be intermittent, and a successive rput be attempted, it is silently ignored. It also enables replication of closures residing on overloaded nodes to be raced on healthy nodes.

It would be unnecessary and costly to reallocate tasks if they had been fully evaluated prior to the failure of the worker node it was assigned to. For this purpose, a probe primitive (Figure 2) is used to identify which IVars are full, indicating the evaluation status of its associated task. As such, all IVars that correspond to tasks allocated to the failed worker node are *probed*. Only tasks associated with empty IVars are reallocated as closures.

Node Failure Detection. A new transport layer for distributed Haskells [6] underlies the fault tolerant workpool. The main advantage of adopting this library is the typed error messages at the Haskell language level.

```
-- connection attempt
attempt <- connect myEndPoint remoteEndPointAddress <default args>
case attempt of
  (Left (TransportError ConnectFailed)) -> -- unblocks 'blockWhileNodeHealthy', Figure 3 line 29
  (Right connection) ->                   -- carry on
```

Fig. 4. Detecting node failure in the blockWhileNodeHealthy function

The connect function from the transport layer is shown in Figure 4. It is used by the workpool to detect node failure in the blockWhileNodeHealthy function on line 29 of Figure 3. Each node creates an endpoint, and endpoints are connected to send and receive messages between the nodes. Node availability is determined by the outcome of connection attempts using connect between the node hosting the supervised workpool, and each worker node utilized by that workpool. The transport layer ensures lightweight communications by reusing the underlying TCP connection. One logical connection attempt between the supervising node and worker nodes is made each second. If Right Connection is returned, then the worker node is healthy and no action is taken. However, if Left (TransportError ConnectFailed) is returned then the worker node is deemed to have failed, and reallocateIncompleteTasks (Figure 3, line 30) re-distributes incomplete tasks originally shipped to this node. Concurrency for monitoring node availability is achieved by Haskell IO threads on line 25 in Figure 3.

An alternative to this design for node failure detection was considered - with periodic *heartbeat* messages sent from the worker nodes *to* the process hosting the supervised workpool. However the bottleneck of message delivery would be the same i.e. involving the endpoint of the process hosting the workpool. Moreover, there are dangers with timeout values for *expecting* heartbeat messages in *asynchronous* messaging systems such as the one employed by HdpH. Remote nodes may be wrongly judged to have failed e.g. when the message queue on the workpool process is flooded, and heartbeat messages are not popped from the message queue within the timeout period. Our design avoids this danger by synchronously checking each connection.

It is necessary that each workpool hosted on a node monitors the availability of worker nodes. With nested or composed supervised workpools there is a risk that the network will be saturated with `connect` requests to monitor node availability. Our architecture avoids this by creating just one Haskell thread per node that monitors availability of all other nodes, irrespective of the number of workpools hosted on a node. Each supervisor thread communicates with these monitoring threads to identify node failure. See the complete implementation [23] for details.

4 High Level Fault Tolerant Abstractions

The HdpH paper [17] describes the implementation of algorithmic skeletons in HdpH. This present work extends these by adding resilience to the execution of two generic parallel algorithmic skeletons.

HdpH provides high level coordination abstractions: evaluation strategies and algorithmic skeletons. The advantages of these skeletons are that they provide a higher level of abstraction [25] that capture common parallel patterns, and that the HdpH primitives for work distribution and operations on `IVar`s are hidden away from the programmer.

We show here how to use fault tolerant workpools to add resilience to algorithmic skeletons. Figure 5 shows the type signatures of fault tolerant versions of the following two generic algorithmic skeletons.

pushMap is a parallel skeleton to provide a parallel map operation, applying a function closure to the input list.

pushDivideAndConquer is another parallel skeleton that allows a problem to be decomposed into sub-problems until they are sufficiently small, and then reassembled with a combining function.

`IVar`s are globalised and closures are created from tasks in the skeleton code, and `supervisedWorkpoolEval` is used at a lower level to distribute closures, and to provide the guarantees described in Section 3.3. The tasks in the workpool are eagerly scheduled into the threadpool of remote nodes.

The two algorithmic skeletons have different scheduling strategies — `pushMap` schedules tasks in a round-robin fashion; `pushDivideAndConquer` schedules tasks randomly (but statically at the beginning, not on-demand).

5 Evaluation

This section demonstrates the use of the fault tolerant mechanisms, and specifically the two fault tolerant algorithmic skeletons from Section 4. Implementations of two symbolic programs are presented: *Summatory Liouville* which is

```
pushMap
   :: [NodeId]          -- available nodes
   -> Closure (a -> b) -- function closure
   -> [a]              -- input list
   -> IO [b]           -- output list

pushDivideAndConquer
   :: [NodeId]                                       -- available nodes
   -> Closure (Closure a -> Bool)                    -- trivial
   -> Closure (Closure a -> IO (Closure b))          -- simplySolve
   -> Closure (Closure a -> [Closure a])             -- decompose
   -> Closure (Closure a -> [Closure b] -> Closure b) -- combine
   -> Closure a                                      -- problem
   -> IO (Closure b)                                 -- output
```

Fig. 5. Fault tolerant algorithmic parallel skeletons used in Appendix A and B of [24]

task parallel; and *Fibonacci* which is a canonical divide-and-conquer problem. The implementations of these can be found in the technical report for this paper [24].

The equivalent non-fault tolerant `pushMap` and `pushDivideAndConquer` skeletons are used for comparing the supervised workpool overheads in the presence and absence of faults, which are described in Section 5.4. These do not make use of the supervised workpool, and therefore do not protect against node failure.

5.1 Data Parallel Benchmark

To demonstrate the `pushMap` data parallel skeleton (Figure 5), Summatory Liouville [3] has been implemented in HdpH, adapted from existing Haskell code [27]. The Liouville function $\lambda(n)$ is the completely multiplicative function defined by $\lambda(p) = -1$ for each prime p. $L(n)$ denotes the sum of the values of the Liouville function $\lambda(n)$ up to n, where $L(n) := \sum_{k=1}^{n} \lambda(k)$. The *scale-up* runtime results measure Summatory Liouville $L(n)$ for $n = [10^8, 2 \cdot 10^8, 3 \cdot 10^8..10^9]$. Each experiment is run on 20 nodes with closures distributed in a round robin fashion, and the chunk size per closure is 10^6. For example, calculating $L(10^8)$ will generate 100 tasks, allocating 5 to each node. On each node, a partial Summatory Liouville value is further divided and evaluated in parallel, utilising multicore support in the Haskell runtime [18].

5.2 Control Parallel Benchmark Using Nested Workpools

The `pushDivideAndConquer` skeleton (Figure 5) is demonstrated with the implementation of Fibonacci. This example illustrates the flexibility of the supervised workpool, which can be nested hierarchically in divide-and-conquer trees. At the point when a closure is deemed too computationally expensive, the problem is decomposed into sub-problems, turned into closures themselves, and pushed to other nodes. In the case of Fibonacci, costly tasks are decomposed into 2 smaller tasks, though the `pushDivideAndConquer` skeleton permits any number of decomposed tasks to be supervised.

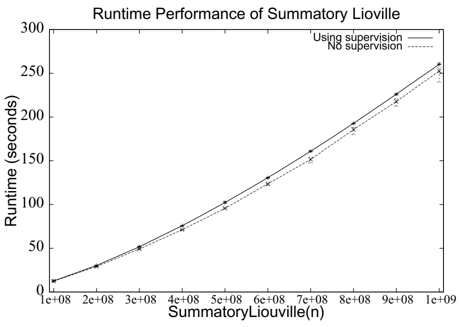
The runtime results measure Fibonacci *Fib*(n) for $n = [45..55]$, and the sequential threshold for each n is 40. Unlike the `pushMap` skeleton, closures are distributed to random nodes from the set of available nodes to achieve fairer load balancing.

5.3 Benchmark Platform

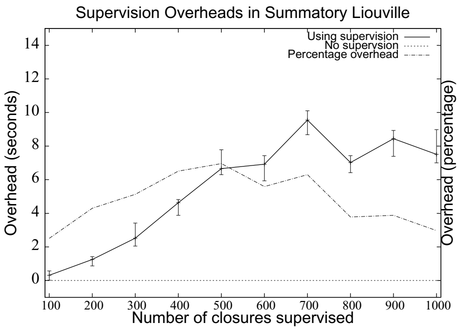
The two applications were benchmarked on a Beowulf cluster. Each Beowulf node comprises two Intel quad-core CPUs (Xeon E5504) at 2GHz, sharing 12GB of RAM. Nodes are connected via Gigabit Ethernet and run Linux (CentOS 5.7 x86_64). HdpH version 0.3.2 was used and the benchmarks were built with GHC 7.2.1. Benchmarks were run on 20 cluster nodes; to limit variability we used only 6 cores per node. Reported runtime is median wall clock time over 20 executions, and reported error is the range of runtimes.

5.4 Performance

No Failure The runtimes for Summatory Liouville are shown in Figure 6(a). The chunk size is fixed, increasing the number of supervised closures as n is increased in $L(n)$. The overheads of the supervised workpool for Summatory Liouville are shown in Figure 6(b). The runtime for Fibonacci are shown in Figure 7.



(a) Runtime performance



(b) Supervision overhead with no failures

Fig. 6. Runtime performance and supervision overheads with no failures for Summatory Lioville 10^8 to 10^9

The supervision overheads for Summatory Liouville range between 2.5% at $L(10^8)$ and 7% at $L(5 \cdot 10^8)$. As the problem size grows to $L(10^9)$, the number of generated closures increases with the chunk size fixed at 10^6. Despite this increase in supervised closures, near constant overheads of between 6.7 and 8.4 seconds are observed between $L(5 \cdot 10^8)$ and $L(10^9)$.

Overheads are not measurable for Fibonacci, as they are lower than system variability (owing probably to random work distribution).

The runtime for calculating $L(5 \cdot 10^8)$ is used to verify the scalability of the HdpH implementation of Summatory Liouville. The median runtime on 20 nodes (each using 6 cores) is 95.69 seconds, and on 1 node using 6 cores is 1711.51 seconds, giving a speed up of 17.9 on 20 nodes.

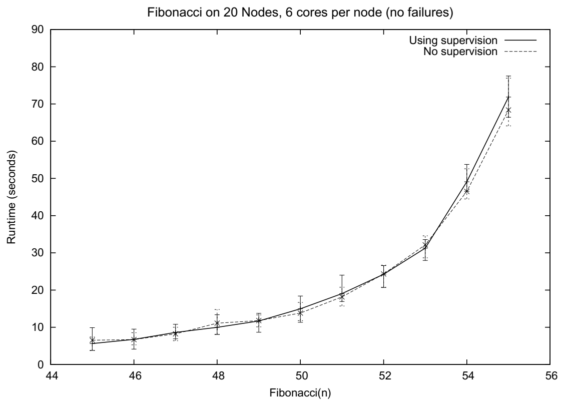


Fig. 7. Runtime performance for Fibonacci with no failures

Recovery From Failure. To demonstrate the fault tolerance and to assess the efficacy of the supervised workpool, recovery times have been measured when *one node* dies during the computation of Summatory Liouville. Nested workpools used by `pushDivideAndConquer` also tolerate faults. Due to the size of the divide-and-conquer graph for large problems, they are harder to analyse in any meaningful way.

To measure the recovery time, a number of parameters are fixed. The computation is $L(3 \cdot 10^8)$ with a chunk size of 10^6, which is initially deployed on 10 nodes, with one hosting the supervising task. The `pushMap` skeleton is used to distribute closures in a round robin fashion, so that 30 closures are sent to each node. An expected runtime utilising 10 nodes is calculated from 5 failure-free executions. From this, approximate timings are calculated for injecting node failure. The Linux `kill` command is used to forcibly terminate one running Haskell process prematurely.

The results in Figure 8 show the runtime of the Summatory Liouville calculation when node failure occurs at approximately [10%,20%..90%] of expected execution time. 5 runtimes are observed at each timing point. Figure 8 also reports the average number of closures that are reallocated relative to when node failure occurs. As described in Section 3.3, only non-evaluated closures are redistributed. The expectation is that the longer the injected node failure is delayed, the fewer closures will need reallocating elsewhere. Lastly, Figure 8 shows 5 runtimes using 10 nodes when *no* failures occur, and additionally 5 runtimes using 9 nodes, again with no failures.

The data shows that at least for the first 30% of the execution, no tasks are complete on the node, which can be attributed to the time taken to distribute 300 closures and for each node to begin evaluation. Fully evaluated closure values are seen at 40%, where only 16 (of 30) are reallocated. This continues to fall until the 90% mark, when 0 closures are reallocated, indicating that all closures had already been fully evaluated on the responsible node.

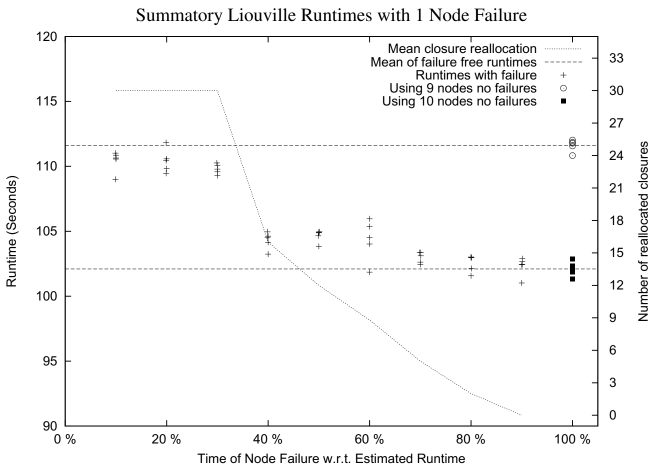


Fig. 8. Recovery time with 1 node failure

The motivation for observing failure-free runtimes using 9 and also 10 nodes is to evaluate the overheads of recovery time when node failure occurs. Figure 8 shows that when a node dies early on (in the first 30% of estimated total runtime), the performance of the remaining 9 nodes is comparable with that of a failure-free run on 9 nodes. Moreover, node failure occurring near the end of a run (e.g. at 90% of estimated runtime) does not impact runtime performance, i.e. is similar to that of a 10 node cluster that experiences no failures at all.

6 Conclusions and Future Work

As COTS and HPC platforms grow in size, faults will become more frequent, rather than failures being exceptional events as in existing architectures. Failures may be caused by hardware malfunction, intermittent network transmission, or software errors. Future dependability of such platforms should therefore rely on a multitude of fault tolerant approaches at all levels of the computational stack.

In this paper, we have presented a language based approach to fault tolerant distributed-memory parallel computation in Haskell: a fault tolerant workpool that hides task scheduling, failure detection and task replication from the programmer. On top of this, we have developed fault tolerant versions of two algorithmic skeletons. They provide high level abstractions for fault tolerant parallel computation on distributed-memory architectures. To the best of our knowledge the supervised workpool is a novel construct.

The supervised workpool and fault tolerant parallel skeleton implementations exploit recent advances in distributed-memory Haskell implementations, primarily HdpH [17] . The workpool and the skeletons guarantee the completion of tasks even in the presence of multiple node failures, withstanding the failure of all but the supervising node. The work is targeting fault tolerant symbolic computation on 10^6 cores within the HPC-GAP project.

The supervised workpool has acceptable runtime overheads — between 2.5% and 7% using a data parallel skeleton. Moreover when a node fails, the recovery costs are negligible.

Future Work. The full HdpH language affords both explicit and implicit closure placement. In contrast, the current supervised workpool implementation permits only static explicit closure distribution. We are adapting the supervised workpool approach to provide fault tolerance to the work stealing scheduler in HdpH.

Acknowledgements. This work has been supported by the European Union grant RII3-CT-2005-026133 'SCIEnce: Symbolic Computing Infrastructure in Europe', IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 'HPC-GAP: High Performance Computational Algebra and Discrete Mathematics'.

References

1. Armstrong, J., Virding, R., Williams, M.: Concurrent Programming in ERLANG. Prentice Hall (1993)
2. Bialecki, A., Taton, C., Kellerman, J.: Apache Hadoop: a Framework for Running Applications on Large Clusters Built of Commodity Hardware (2010), `http://hadoop.apache.org/`
3. Borwein, P.B., Ferguson, R., Mossinghoff, M.J.: Sign changes in Sums of the Liouville Function. Mathematics of Computation 77(263), 1681–1694 (2008)
4. Bouteiller, A., Cappello, F., Hérault, T., Krawezik, G., Lemarinier, P., Magniette, F.: MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. Super Computing 25 (2003)
5. Cappello, F., Geist, A., Gropp, B., Kalé, L.V., Kramer, B., Snir, M.: Toward Exascale Resilience. High Performance Computing Applications 23(4), 374–388 (2009)
6. Coutts, D., de Vries, E.: The New Cloud Haskell. In: Haskell Implementers Workshop. Well-Typed (September 2012)
7. M. development. Feature: Adding -disable-auto-cleanup to mpich2 (2010), `http://goo.gl/PNEaO`
8. Epstein, J., Black, A.P., Jones, S.L.P.: Towards Haskell in the Cloud. In: Haskell Symposium, pp. 118–129 (2011)
9. Fabre, J.-C., Nicomette, V., Pérennou, T., Stroud, R.J., Wu, Z.: Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. In: Symposium on Fault-Tolerant Computing, pp. 489–498 (1995)
10. Fagg, G.E., Dongarra, J.: FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In: Euro. PVM/MPI, pp. 346–353 (2000)
11. The GAP Group. GAP – Groups, Algorithms, and Programming, `http://www.gap-system.org`.
12. Gropp, W., Lusk, E.: Fault Tolerance in MPI Programs. Special Issue of the Journal High Performance Computing Applications 18, 363–372 (2002)
13. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. Scientific And Engineering Computation. MIT Press (1994)

14. Larson, J.: Erlang for Concurrent Programming. In: ACM Queue, vol. 6, pp. 18–23. ACM (September 2008)
15. Liskov, B., Shrira, L.: Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In: PLDI, pp. 260–267. ACM (1988)
16. Maier, P., Stewart, R., Trinder, P.: Reliable Scalable Symbolic Computation: The Design of SymGridPar2. In: SAC 2013. ACM (to appear, 2013)
17. Maier, P., Trinder, P.: Implementing a high-level distributed-memory parallel Haskell in Haskell. In: Gill, A., Hage, J. (eds.) IFL 2011. LNCS, vol. 7257, pp. 35–50. Springer, Heidelberg (2012)
18. Marlow, S., Jones, S.L.P., Singh, S.: Runtime Support for Multicore Haskell. In: ICFP, pp. 65–78 (2009)
19. Marlow, S., Newton, R., Jones, S.L.P.: A Monad for Deterministic Parallelism. In: Haskell Symposium, pp. 71–82 (2011)
20. Niehren, J., Schwinghammer, J., Smolka, G.: A Concurrent Lambda Calculus with Futures. Theoretical Computer Science 364(3), 338–356 (2006)
21. Schroeder, B., Gibson, G.A.: Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In: FAST. USENIX Association (2007)
22. Shavit, N., Touitou, D.: Software Transactional Memory. In: PODC 1995, pp. 204–213. ACM (1995)
23. Stewart, R., Maier, P., Trinder, P.: Implementation of the HdpH Supervised Workpool (July 2012), `http://www.macs.hw.ac.uk/~rs46/papers/tfp2012/SupervisedWorkpool.hs`
24. Stewart, R., Maier, P., Trinder, P.: Supervised Workpools for Reliable Massively Parallel Computing. Technical report, Heriot-Watt University (2012), `http://www.macs.hw.ac.uk/˜rs46/papers/tfp2012/TFP2012_Robert_Stewart.pdf`
25. Trinder, P.W., Hammond, K., Loidl, H.-W., Peyton Jones, S.L.: Algorithm + Strategy = Parallelism. Journal of Functional Programming 8(1), 23–60 (1998)
26. Trottier-Hebert, F.: Learn You Some Erlang For Great Good - Building an Application With OTP (2012), `http://learnyousomeerlang.com/building-applications-with-otp`
27. Zain, A.A., Hammond, K., Berthold, J., Trinder, P.W., Michaelson, G., Aswad, M.: Low-pain, High-Gain Multicore Programming in Haskell: Coordinating Irregular Symbolic Computations on Multicore Architectures. In: DAMP, pp. 25–36. ACM (2009)

RELEASE: A High-Level Paradigm for Reliable Large-Scale Server Software
(Project Paper)

Olivier Boudeville[1], Francesco Cesarini[2], Natalia Chechina[3], Kenneth Lundin[4], Nikolaos Papaspyrou[5], Konstantinos Sagonas[6], Simon Thompson[7], Phil Trinder[3], and Ulf Wiger[2]

[1] EDF R&D, 92140 Clamart, France
[2] Erlang Solutions AB, 113 59 Stockholm, Sweden
[3] Heriot-Watt University, Edinburgh, EH14 4AS, UK
[4] Ericsson AB, 164 83 Stockholm, Sweden
[5] Institute of Communication and Computer Systems (ICCS), Athens, Greece
[6] Uppsala University, 751 05 Uppsala, Sweden
[7] University of Kent, Canterbury, CT2 7NF, UK
`http://www.release-project.eu`

Abstract. Erlang is a functional language with a much-emulated model for building reliable distributed systems. This paper outlines the RELEASE project, and describes the progress in the first six months. The project aim is to scale the Erlang's radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines.

Currently Erlang has inherently scalable computation and reliability models, but in practice scalability is constrained by aspects of the language and virtual machine. We are working at three levels to address these challenges: evolving the Erlang virtual machine so that it can work effectively on large scale multicore systems; evolving the language to Scalable Distributed (SD) Erlang; developing a scalable Erlang infrastructure to integrate multiple, heterogeneous clusters. We are also developing state of the art tools that allow programmers to understand the behaviour of massively parallel SD Erlang programs. We will demonstrate the effectiveness of the RELEASE approach using demonstrators and two large case studies on a Blue Gene.

Keywords: Erlang, scalability, multicore systems, massive parallelism.

1 Introduction

There is a widening gap between state of the art in hardware and software. Architectures are inexorably becoming manycore, with the numbers of cores per chip following Moore's Law. Software written using conventional programming languages, on the other hand, is still essentially sequential: with a substantial effort, some degree of concurrency may be possible, but this approach just doesn't scale to 100s or 1000s of cores. However, manycore programming is not only about concurrency. We expect 100,000 core platforms to become commonplace, and the predictions are that core failures on such an architecture will become relatively common, perhaps one hour mean time between core failures [1]. So manycore systems need to be both scalable and robust.

The RELEASE project aim is to scale the radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines. The trend-setting concurrency-oriented programming model we will use is Erlang/OTP (Open Telecom Platform) – designed in the telecoms sector, is based on highly-scalable lightweight processes *which share nothing*, and used in strategic Ericsson products such as the AXD301 telecoms switch [2]. Erlang/OTP provides high-level coordination with concurrency and robustness built-in: it can readily support 10,000 processes per core, and transparent distribution of processes across multiple machines, using message passing for communication. The robustness of the Erlang distribution model is provided by hierarchies of supervision processes which manage recovery from software or hardware errors. Erlang is an Actor-based programming language where actions are performed by concurrent processes named actors [3]. Actors communicate with each other via asynchronous message passing, and each actor has an address and a mailbox. An actor also has a behaviour that may change in a response to a received message. Actors may create and kill other actors [4].

The Erlang/OTP has inherently scalable computation and reliability models, but in practice scalability is constrained by the transitive sharing of connections between all nodes and by explicit process placement. Moreover programmers need support to engineer applications at this scale and existing profiling and debugging tools do not scale, primarily due to the volumes of trace data generated. In the RELEASE we tackle these challenges working at three levels (Figure 1):

1. *evolving the Erlang Virtual Machine (VM)* so that it can work effectively on large scale multicore systems (Section 3.1);
2. *evolving the language to Scalable Distributed (SD) Erlang*, and adapting the OTP framework to provide both constructs like locality control, and reusable coordination patterns to allow SD Erlang to effectively describe computations on large platforms, while preserving performance portability (Section 3.2);
3. *developing a scalable Erlang infrastructure* to integrate multiple, heterogeneous clusters (Section 3.3).

These developments will be supported by state of the art tools to allow programmers to understand the behaviour of large scale SD Erlang programs, and to refactor standard Erlang programs into SD Erlang (Section 3.4). We will demonstrate the effectiveness of the RELEASE approach through building two significant demonstrators: a simulation based on a port of SD Erlang to the Blue Gene architecture [5] and a large-scale continuous integration service – and by investigating how to apply the model to an Actor framework [6] for a mainstream language (Section 4).

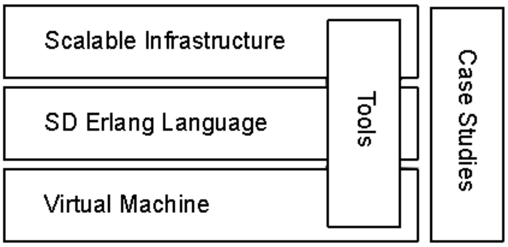


Fig. 1. RELEASE Project Research Areas

The Erlang community is growing exponentially, moreover it has defined a widely accepted concurrency-oriented programming and become a beacon language for reliable distributed computing. As such it influences the design and implementation of numerous Actor programming languages, libraries and frameworks like Cloud Haskell [7], Scala and its Akka framework [8], F# [9], and Kilim [10]. Hence, we expect that RELEASE will have a similar impact on the design and implementation of a range of languages, libraries and frameworks, and thus deliver significant results beyond the Erlang community.

2 Progress beyond the State-of-the-Art

The RELEASE project targets reliable scalable general purpose computing on heterogeneous platforms. Our application area is that of general server-side computation, e.g. a web or messaging server. This form of computation is ubiquitous, in contrast to more specialised forms such as traditional High Performance Computing (HPC). Moreover, this is computation on stock platforms, with standard hardware, operating systems and middleware. We aim for 10^5 cores, for example, the Blue Gene/Q that will be exploited during the project has 65,000 cores. Our focus on commodity hardware implies that we do not aim to exploit the experimental many-core architectures like the Intel Tera-scale [11].

The project makes advances in a number of areas, primarily in Virtual Machine (VM) support for high-level concurrency (Section 2.1), in scalable high-level distributed languages (Section 2.2), in tools for concurrent software engineering (Section 2.3), in cloud hosting infrastructure (Section 2.4), and in massive scale simulation (Section 2.5).

2.1 VM Support for High-Level Concurrency

Due to the development of multicore architectures and the distributed nature of modern computer systems, the quantity and variance of processors on which software is executed has increased by orders of magnitude in recent years and is expected to increase even further. In this setting the role of high-level concurrent programming models is very important. Such models must abstract from variations introduced by differences between multicore architectures and uniformly treat different hardware architectures, number of cores and memory access characteristics. Currently, the implementation of such models in the form of high-level languages and libraries is not sufficient; these two must be complemented with efficient and reliable VMs that provide inherent support for high-level concurrency [12].

Historically the efficient implementation of VMs for single core processor systems has presented a number of challenges, largely unrelated to concurrency. For example, in order to optimally use the hardware, a VM has to exploit deep hierarchies of cache memory by reorganizing data layouts, and to support e.g. out-of-order execution, the hardware's prefetching heuristics, branch prediction. With multicore processors, concurrency and the inherent shared memory model introduce new challenges for VMs, as it is not only arbitrary thread interleavings but parallel execution on limited shared resources which has to be taken into account. On top of the implementation considerations mentioned before, cache coherence becomes a critical issue, memory barriers become a necessity and compiler optimisations, e.g. instruction reordering, must often be more conservative to be semantics-preserving. Furthermore, multicore machines currently rely on Non-Uniform Memory Access (NUMA) architectures, where the cost of accessing a specific memory location can be different from core to core and data locality plays a crucial role for achieving good performance.

In the past few years, there has been sustained research on the development of VMs for software distributed shared memory. Some recent research aims to effectively employ powerful dedicated and specialized co-processors like graphic cards. Notable VM designs in this direction are the CellVM [13] for the Cell Broadband Engine, a VM with a distributed Java heap on a homogeneous TILE-64 system [14]; an extension of the JikesVM to detect and offload loops on CUDA devices [15]; and VMs for Intel's Larrabee GPGPU architecture [16]. Most of these designs are specific to VMs for Java-like languages which are based on a shared-memory concurrent programming model. Despite much research, the implementation of shared memory concurrency still requires extensive synchronisation and for this reason is inherently non-scalable. For example, in languages with automatic memory management, the presence of a garbage collector for a heap which is shared among all processes/threads imposes a point of synchronisation between processes and thus becomes a major bottleneck.

Although a language based on the Actor concurrency model is in principle better in this respect, its VM implementation on top of on shared memory hardware in an efficient and scalable way presents many challenges [17]. In the case of the implementation of Erlang, various runtime system architectures have been explored by the High Performance Erlang (HiPE) group in Uppsala, based either on process-local memory areas, on a communal heap which is shared among all threads, or following some hybrid scheme. However, the performance evaluation [18] was conducted on relatively small multiprocessor machines (up to 4 CPUs) in 2006 and cannot be considered conclusive as far as scalability on the machines that the RELEASE project is aiming at. More generally, the technology required to reach the scalability target of the current project requires significant extensions to the state of the art in the design of VMs for message passing concurrency and will stretch the limits of data structures and algorithms for distributed memory management.

2.2 Scalable Reliable Programming Models

Shared memory concurrent programming models like OpenMP [19] or Java Threads are generally simple and high level, but do not scale well beyond 10^2 cores. Moreover reliability mechanisms are greatly hampered by the shared state, for example, a lock becomes permanently unavailable if the thread holding it fails. In the HPC environment the distributed memory model provided by the MPI communication library [20] dominates. Unfortunately MPI is not suitable for producing general purpose concurrent software as it is too low level with explicit, synchronous message passing. Moreover the most widely used MPI implementations offer no fault recovery[1]: if any part of the computation fails, the entire computation fails.

For scalable high-level general purpose concurrent programming a more flexible model is required. Actors [6] are a widely used model and are built into languages like Scala [22], Ptolemy [23], and Erlang [3]. There are also Actor frameworks and libraries for many languages, for example, Termite Scheme [24], PARLEY for Python [25], and Kilim [10] for Java. The Erlang style concurrency has the following key aspects [26]: fast process creation/destruction; scalability to support more than 10^4 concurrent processes; fast asynchronous message passing; copying message-passing semantics, i.e. share-nothing concurrency; process monitoring; selective message reception.

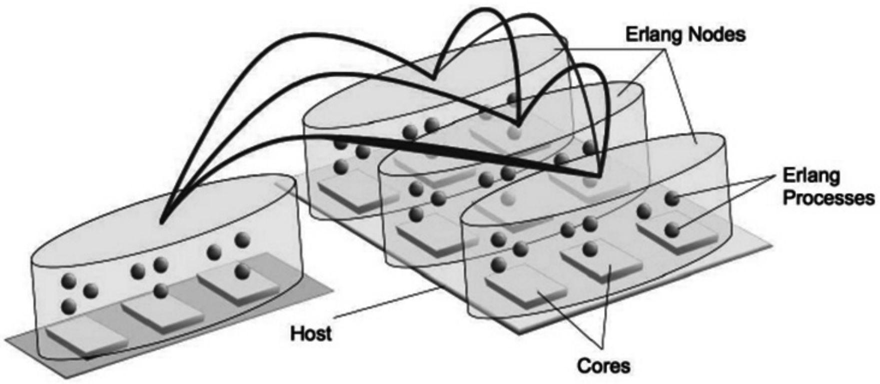


Fig. 2. Conceptual View of Erlang's Concurrency, Multicore Support and Distribution

[1] Some fault tolerance is provided in less widely used MPI implementations like [21].

Figure 2 illustrates Erlang's support for concurrency, multicores and distribution. A rectangle represents a host with an IP address, and an arc represents a connection between nodes. Multiple Erlang processes may execute in a node, and a node can exploit multiple processors, each having multiple cores. Erlang supports single core concurrency where a core may support as many as 10^8 lightweight processes [2]. In the Erlang distribution model a node may be on a remote host, and this is almost entirely transparent to the processes. Hosts need not be identical, nor do they need to run the same operating system.

Erlang currently delivers reliable medium scale distribution, supporting up to 10^2 cores, or 10^2 distributed memory processors. However, the scalability of a distributed system in Erlang is constrained by the transitive sharing of connections between all nodes and by explicit process placement. The transitive sharing of connections between nodes means that the underlying implementation needs to maintain data structures that are quadratic in the number of processes, rather than considering the communication locality of the processes. While it is possible to explicitly place large numbers of processes in a regular, static way, explicitly placing the irregular or dynamic processes required by many servers and other applications is far more challenging. The project addresses these limitations.

2.3 Tools for Concurrency and Erlang

From the inception of parallel programming, the availability of tools that ease the effective deployment of programs on parallel platforms has been a crucial criterion for success. The Intel Trace Analyzer and Collector [27] is a typical modern tool, which supports the analysis, optimisation and deployment of applications on Intel-based clusters with MPI communication.

The Erlang VM is equipped with a comprehensive, low-level tracing infrastructure provided by the trace built-in functions [3]. There is a number of higher-level facilities built upon this, including the debugger and the Trace-Tool Builder (TTB) [28]. The later provides facilities for managing tracing across a set of distributed Erlang nodes. TTB is designed to be extensible to provide different types of tracing tuned to different applications and environments. While these tools support monitoring Erlang programs across distributed nodes, there is a problem in dealing with the volume of data generated; on a multicore chip, or highly parallel system, it will be impossible to ship all the data off the chip without swamping communication entirely [29]. So, we will ensure that local monitoring and analysis can be performed within the system, leveraging the locality supported by SD Erlang to support a hierarchical 'tracing architecture'.

In building analyses we will be able to leverage work coming from the FP7 ProTest project [30] including the on-line monitoring tool Inviso, which has recently been integrated into TTB, and the off-line monitoring tool Exago. Higher-level analysis can be provided independently [31], and building on this it is also possible to analyse the process structure of Erlang applications, particularly those structured to use the OTP standard behaviours [32, Chapter 6]. These existing systems need to be extended to integrate monitoring with process discovery.

Building on the Erlang syntax_tools package and the standard Erlang compiler tool chain, Wrangler is a tool for refactoring Erlang programs [33]. In the RELEASE project we will develop and implement refactorings from Erlang to SD Erlang within Wrangler. To give users guidance about which refactorings could be applied we will develop a refactoring assistant, that will suggest refactorings of a given system on the basis of the system behaviour.

Traditional breakpoint debuggers are of little use for Reliable Massively Parallel (RMP) software; there are too many processes and breakpoints conflict with the frequent timeouts in Erlang's communication. The RELEASE project will build a debugger for massively parallel systems that retains a recent history of the current computation state, as well as saving fault-related information, so that this can be retraced and explored for debugging purposes.

2.4 Cloud Hosting

The cloud hosting area is moving rapidly, and progress is driven mainly by entrepreneurs and competing cloud infrastructure providers. While users can already choose between many hosting providers offering price/performance alternatives, provisioning is typically manual and time consuming, making it unnecessarily difficult to switch providers.

With increased competition comes specialisation, which in its turn introduces an integration challenge for users. This calls for a broker layer between users and cloud providers [34], but creating such a broker layer, especially a dynamic one, is no easy task, not least because Cloud Provision APIs currently do not support ad-hoc, capability-based provisioning and coordination of cloud resources. A problem is that basic cloud APIs must primarily serve mainstream languages, not least REST (stateless) clients, where management of dynamic and complex state is considered extremely difficult [35].

We intend to advance the concept of a capability-based dynamic cloud broker layer, building on the cloud broker component of Erlang Solutions' recently launched Hosted Continuous Integration tool SWARM. SWARM is capable of allocating a mix of cloud and physical resources on demand and running complex tests in parallel, and draws on the ease with which Erlang can spawn and coordinate parallel activities, as well as its considerable strengths as a test automation and load testing environment. We intend to use SWARM itself as a testbed for the concepts in the RELEASE project, aiming to increase both the capabilities of SWARM and its cost-effectiveness.

At a basic level, this broker would be useful for any kind of cloud application (even non-Erlang) making it easier for users to switch provider, and create on-demand, capability-driven mashups of specialised clusters. For more dynamic capabilities, we will lead by providing a Cloud Broker API customised for the Erlang programming model.

2.5 Scalable Simulation

Many business and scientific fields would benefit from larger-scale simulations as the need for intrinsically risky extrapolation can be removed if we are able to simulate a complex system as a whole. Moreover, some important behaviours only appear in detailed simulations. Distributed memory clusters are cost effective scalable platforms. However, simulations are typically initially developed using sequential, imperative technologies and these are ill suited for distributed execution. Additionally parallelism poses a number of challenges for simulation. We believe that declarative programming is an appropriate tool to harness parallel platforms.

Sim-Diasca is a distributed engine for large scale discrete simulations implemented in Erlang. It is among the most scalable discrete simulation engines and currently is able to handle more than one million relatively complex model instances using only hundreds of cores. However, more accurate models are required – ideally we would like 50-200 million model instances. We would also like to introduce reliability. The achievement of these goals requires scalable reliability improvements across the software stack, i.e. VM, language, and engine.

3 Developing a Scalable Programming Model

This section gives an overview of the technical and research components of the project covering the following areas: scaling the Erlang VM (Section 3.1), scaling the Erlang programming model (Section 3.2), scalable virtualisation infrastructure (Section 3.3), and tool support for the development of RMP software (Section 3.4).

3.1 Scaling the Erlang VM

The Erlang VM was initially designed as a machine to support cooperative concurrent execution of processes ("green threads") on physical machines with a single CPU. Since 2006, the Erlang VM has been extended to support Symmetric MultiProcessing (SMP) in the form that is commonly found these days in multicore machines. The goal has been to focus on stability while giving incremental performance improvements in each release. This approach has worked well for the important class of high-availability server software running on machines with up to 16 cores, but needs significant extensions to achieve scalability on bigger multicore machines.

In the RELEASE project we aim to re-design and improve some core aspects of Erlang's VM so that it becomes possible for applications to achieve highly scalable performance on high-end multicore machines of the future with minimal refactoring of existing applications. Our goal is to push a big part of the responsibility for achieving scalability from the application programmer to the VM. In particular, we will investigate architectural changes and alternative implementations of key components of Erlang's VM that currently hinder scalability of applications on large multicore machines. A key consideration is to design scalable components without sacrificing crucial aspects of the existing architecture such as process isolation, high-reliability, robustness and soft real-time properties.

To achieve these goals, we will begin our work by a detailed study of the performance and scalability characteristics of a representative set of existing Erlang applications running on the current VM. This will help us both to identify the major scalability bottlenecks and to prioritize changes and extensions of the runtime system and of key VM components. Independently of the results of this study however, there are parts of the VM which could definitely benefit from extensions or redesign. One of them is the Erlang Term Storage (ETS) mechanism.

Although Erlang processes do not share any memory at the language level, at the implementation level, the Erlang VM provides built-ins that allow processes to store terms in shared global data structures, called ETS tables, and to destructively modify their contents. Currently, many Erlang applications make extensive use of this mechanism, either directly in their code or indirectly via using in-memory databases such as Mnesia [3]. With the current implementation of ETS, when the number of processes gets big, an access to these tables becomes serialisation points for applications and hinder their scalability. Moreover, due to the nature of the garbage collector currently employed by the Erlang VM, an access to terms in ETS tables requires a physical copy of the term from the table to the heap of the process. We will investigate scalable designs of ETS tables that avoid these performance bottlenecks. In addition we will experiment with runtime system organisations and design language built-ins that avoid the need for copying data from ETS tables to the process-local memory areas when it is provably safe to do so.

In the current Erlang VM processes allocate the majority of their data in process-local memory areas. Whenever processes need to communicate, they must explicitly copy their data from the heap of the sender to that of the receiver. The processes also need to wait to get rescheduled when execution reaches a receive statement with no appropriate messages in the process mailbox. We will design and investigate scalable runtime system architectures that allow groups of processes to communicate without the need for explicit copying. We will also develop the runtime support for processes to temporarily yield to their appropriate senders when reaching a receive statement that would otherwise block. One possible such architecture is a clustered shared heap aided by the presence of language constructs such as fibers available in languages like C++.

To identify "frequently-communicating processes" and to guide the VM schedulers a scalable design of the Erlang VM needs to be supported both by language extensions and by static and dynamic analysis. A significant effort in this task is not only to design and implement the analysis, but also to integrate it in the development environment in a smooth and seamless way. No matter how scalable the underlying VM will get, large scale applications will need tool support to identify bottlenecks (Section 3.4). Finally, to test the scalability of our implementation and to enable a case study we will port the Erlang VM to a massively parallel platform. Our current plan is to use a Blue Gene/Q machine. This plan may change, or be extended to include more platforms, if we gain access to more powerful such machines during the duration of the project.

3.2 Scaling the Erlang Programming Model

We will extend both Erlang, to produce Scalable Distributed (SD) Erlang, and the associated OTP library. The SD Erlang name is used only as a convenient means of identifying the language and VM extensions as we expect them to become standard Erlang in future Erlang/OTP releases.

Controlling Connections. The scalability of a distributed Erlang system is constrained by the transitive sharing of connections between all nodes and by explicit process placement. SD Erlang will regain scalability using layering, and by controlling connection locality by grouping nodes and by controlling process placement affinity.

Process Placement. Currently the Erlang distribution model permits explicit process placement: a process is spawned on a named node. Such a static, directive mechanism is hard for programmers to manage for anything other than small scale, or very regular process networks. We propose to add an abstraction layer that maintains a tree of node groups, abstractly modelling the underlying architecture. We will provide mechanisms for controlling *affinity*, i.e. how close process must be located, e.g. two rapidly communicating processes may need to be located in the same node. We will also provide mechanisms for controlling *distribution*, i.e. how far the process must be from the spawning process. For example, two large computations, such as simulation components, may need to be placed on separate clusters.

Scaling Reliability. Erlang/OTP has world leading language level reliability. The challenge is to maintain this reliability at massive scale. For example, any node with a massive number of connections should be placed in a different node group from its supervisor. A new OTP principle could be to structure systems with the supervision tree preserving this property.

Performance Portability. The abstract computational control mechanisms are not strongly related to a specific architecture. As far as possible we intend to construct performance portable programs, and computational patterns, by computing distance metrics for the affinity and distribution metrics. Moreover, locality control enables us to use layering as a principle to structure systems: for example, the control processes for a layer appearing in a different group from the controlled processes. This facilitates performance portability as the top layers can be refactored for the new architecture while preserving the lower layers unchanged.

Scalable and Portable OTP. Some OTP principles and behaviours will need to be extended with new scalable principles, and perhaps to some extent redesigned and refactored to control locality and support layering. The supervisor group discussed above and the control layering are examples of new scalable principles.

3.3 Scalable Virtualisation Infrastructure

Given the aim of the RELEASE project to develop a model of participating clusters, it is logical to also explore the possibility of creating super-clusters of on-demand clusters provisioned from competing Cloud providers. This would make it possible to cost-optimise large clusters by matching capability profiles against the requested work, and combining groups of instance types, possibly from different providers, into a larger grid. The complexities of running a computing task across such a cluster generally fall into the categories addressed within the RELEASE project: providing a layer of distribution transparency across cooperating clusters; monitoring neighbouring clusters and recovering from partial failures; and tolerance to latency variations across different network links.

A possible small-scale use for a Cloud cluster broker could be to act as a "Pricerunner" for on-demand computing resources. For Erlang Solutions, it is a natural extension of their Hosted Continuous Integration and Testing as a Service offerings (SWARM), where the system can match the computing needs of different build-and-test batches against availability and pricing of virtual images from different providers. In the context of Hosted Continuous Integration, this capability can also be used to simulate both server- and client-side, using different capabilities, and possibly different providers, for each. It needs to be easy to configure and automate.

The infrastructure that we construct will be novel as few cloud computing providers today offer a powerful enough API for this task. For this reason, we will build our own virtualisation environment, e.g. based on the Eucalyptus Cloud software, which is API-compatible with Amazon EC2, but available as open source.

3.4 Tool Support for Developing Reliable Parallel Software

The Erlang programming model provides abundant concurrency, with no theoretical limit to the number of concurrent processes existing – and communicating – at any particular time. In practice, however, there can be problems in the execution of highly concurrent systems on a heterogeneous multicore framework. Two particular difficulties are, first, *balancing of computational load between cores.* Each Erlang process will run on a single core, and a desirable property of the system is that each core is utilised to a similar extent. Secondly, there may be *bottlenecks due to communication.* Each process executes on a single core, but a typical process will communicate with other processes which are placed on other cores, and in a large system this can itself become a bottleneck. Using more cores can ease load balancing, but communication bottlenecks may be alleviated by keeping related processes close together, potentially on the same core, but these two are in tension.

We will supply tools that can measure and visualise performance in a number of different ways. The Erlang VM is equipped with a comprehensive, low-level tracing infrastructure on which other tools can be built. DTrace also provides dynamic tracing support at the operating system level on a number of Unix platforms, complementing the built-in facilities. Because of the volume of data generated, we will need to ensure that local monitoring and analysis are performed, leveraging the locality properties of the running system. The tools will be built in sequence. First we will develop textual reports and graphical visualisations for off-line monitoring, secondly, these will be generated as snapshots of a running system, and finally we will build tools to support interactive, real-time on-line system monitoring.

For SD Erlang to be taken up in practice, it will be essential to provide users with a migration path from Erlang: using the Wrangler refactoring platform we will provide the infrastructure for users to migrate their programs to SD Erlang, and moreover provide decision-support tools suggesting migration routes for particular code. Finally, we will supply tools to debug concurrency bugs (or Heisenbugs) in SD Erlang programs, and to develop an *intelligent debugger* based on saving partial histories of computations.

4 Case Studies

The case studies will demonstrate and validate the new reliable massively parallel tools and methodologies in practice. The major studies using SD Erlang and the scalable infrastructure are large-scale discrete simulations for EDF, i.e. 10^7 model instances, and dynamically scalable continuous integration service for Erlang Solutions. Key issues include performance portability, scalability and reliability for large-scale software. We will also investigate the feasibility of impacting dominant programming models by adding our scalable reliability technologies to an Actor framework for a mainstream programming language, such as Java.

EDF Simulation. The goals of this first case study are to enhance the reliability and scalability of an existing open-source simulation engine, Sim-Diasca that stands for "Simulation of Discrete Systems of All Scales". Sim-Diasca is a general-purpose engine dedicated to all kinds of distributed discrete simulations, and currently used as the corner stone for a few simulators of smart metering systems. The increased scalability will enable Sim-Diasca to execute simulations at an unprecedented scale by exploiting large-scale HPC resources, clusters or Blue Gene supercomputer that in total provide 10^5 cores.

Currently, Sim-Diasca is designed to halt if any node fails during a simulation. The goal is to make the engine able to resist to the crash of up to a pre-determined number of computing nodes, e.g. 3 nodes. This is a twofold task:

- The main part of the task is to add application-level fault tolerance to the engine, so that the engine is able to store its state based on triggers, e.g. wall clock durations elapsed, simulation intermediate milestone met, in such a way that state is spread over all the computed nodes involved. Thus, a distributed snapshot is to be created, and the state of a given node must be duplicated to k other nodes depending on the targeted k-reliability class, either in RAM or on a non-volatile memory, e.g. local files, a distributed file system, or a replicated database. This check-pointing is meant to allow for a later restart on a possibly different resource configuration.

– The second part of the task is to integrate the lower-level mechanisms for reliability provided by SD Erlang. Some mechanisms will be transparent, whereas others will be used as building blocks for the first part of the task.

The distributed snapshot/restart feature will be implemented using a trigger, e.g. every N simulation ticks all the attributes of all the simulation instances are replicated on k other nodes. This requires that simulation agents like the time managers and data-loggers have means of serialising their state for future re-use. The main difficulty is to do the action for all model instances.

Tolerating the common case of losing a "slave" node at the leaf of the simulation tree is the basic requirement. More advanced challenges to be investigated include the following:

– Tolerating the loss of intermediate nodes, or even the root node. For the latter case consensus and leader election are required, and may be generic reliability mechanisms for Sim-Diasca.
– The dynamic addition of nodes that naturally follows from tolerating the loss of nodes.
– Supporting instance migration, for example, for planned down-time or load balancing.

Target platforms include both large clusters such as the Blue Genes and multicore cards like Tilera. The advantages of the latter are ready access and local administration.

Continuous Integration Service. The second case study will be to integrate the support for on-demand, capability-driven heterogeneous super clouds into Erlang Solutions' Continuous Integration framework, SWARM. We envisage integrating up to 4 Amazon EC2 clusters ranging from small, e.g. four 1GHz cores, to large, e.g. hundreds of 3GHz cores. To meet client requirements we will also include in-house clusters, such as bespoke embedded device clusters, e.g. 20 ARM cores, or clusters of dedicated machines currently not available in virtualised environments.

Erlang Solutions has a dynamic stream of customer projects and the exact configurations will depend on the clients' use cases, but already, prospective clients of SWARM include massively scalable NoSQL databases, multisite instant messaging systems and middleware for nation-wide trading infrastructures. Erlang Solutions is also currently developing ad-hoc networking solutions for mobile devices. We plan to explore the potential of Hosted Continuous Integration in all these areas, and derive from that experience what a common provisioning framework for heterogeneous clusters should look like.

Scalable Reliability for a Mainstream Actor Framework. The goal of this study is to evaluate how the scalable reliable concurrency oriented paradigm we will develop can be effectively applied to mainstream software development. We will investigate the feasibility and limitations of adding SD Erlang scalability constructs to a popular Actor framework for a dominant language. The investigation will focus on an open source framework with an active user base. One such framework that meets the requirements is Kilim for Java [10].

5 Progress So Far

In the first six months of the RELEASE project we have made the following progress [36].

We have started to design SD Erlang by making an overview of architecture trends, possible failures, and Erlang modules that might impinge on Erlang scalability [37]. We have formulated the main design principles, and analysed in-memory and persistent data storage mechanisms. We also have developed an initial SD Erlang design that includes implicit process placement and scalable groups to reduce node connections.

We have started to benchmark and trace multicore systems using both the built-in Erlang tracing and DTrace for Unix systems. We are also compiling a set of programs for Erlang and in particular Distributed Erlang, so that we can benchmark the performance of multicore SD Erlang on a set of practical problems [38].

We have produced a survey of the state-of-the-art cloud providers and management infrastructures. The survey is a foundation for the initial work on providing access to cloud infrastructure for different tasks, such as system building, testing and deployment. We have added a few useful features to Sim-Diasca to ease troubleshooting, e.g. a distributed instance tracker. We are also working on a scalable simulation case that can be used for benchmarking purposes.

6 Conclusion

The RELEASE project intends to scale Erlang's concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel commodity hardware. We outline the state of the art (Section 2), and then explain how we plan to tackle the challenge at the following three levels. We are evolving the Erlang VM for large scale multicore systems (Section 3.1). We are evolving the language to Scalable Distributed (SD) Erlang, and adapting the OTP framework to provide both constructs like locality control, and reusable coordination patterns (Section 3.2). We are developing a scalable Erlang infrastructure to integrate multiple, heterogeneous clusters (Section 3.3). To support these activities we are developing tools to enable programmers to understand the behaviour of large scale Erlang systems, and to refactor standard Erlang programs into SD Erlang (Section 3.4). We have outlined how case studies will be used to investigate and validate the new reliable massively parallel tools and methodologies in practice (Section 4). Finally, we have briefly outlined our initial progress (Section 5). We look forward to reporting on the progress of the project in the years to come.

Acknowledgements. We would like to thank all our colleagues who work on the RELEASE project. This work has been supported by the European Union grant RII3-CT-2005-026133 'SCIEnce: Symbolic Computing Infrastructure in Europe', IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 'HPC-GAP: High Performance Computational Algebra and Discrete Mathematics'.

References

1. Trew, A.: Parallelism and the exascale challenge. Distinguished Lecture. St Andrews University, St Andrews, UK (2010)
2. Ericsson, A.B.: Erlang/OTP Efficiency Guide, System Limits (2011), `erlang.org/doc/efficiency_guide/advanced.html#id67011`
3. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly (2009)
4. Agha, G.: An overview of Actor languages. SIGPLAN Not. 21(10), 58–67 (1986)
5. IBM Research: IBM Research Blue Gene Project (2012), `www.research.ibm.com/bluegene/index.html`
6. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: IJCAI 1973, pp. 235–245. Morgan Kaufmann, San Francisco (1973)
7. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards Haskell in the Cloud. In: Haskell 2011, pp. 118–129. ACM (2011)
8. Odersky, M., et al.: The Scala programming language (2012), `www.scala-lang.org`
9. Syme, D., Granicz, A., Cisternino, A.: Expert F#. Springer (2007)
10. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
11. Intel Corporation: Tera-scale computing architecture overview (2012), `http://www.intel.com/content/www/us/en/research/intel-labs-terascale-computing-demo.html`
12. Marr, S., Haupt, M., Timbermont, S., Adams, B., D'Hondt, T., Costanza, P., Meuter, W.D.: Virtual machine support for many-core architectures: Decoupling abstract from concrete concurrency models. In: PLACES, pp. 63–77 (2009)
13. Noll, A., Gal, A., Franz, M.: CellVM: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. In: Workshop on Cell Systems and Applications (2009)
14. Ungar, D., Adams, S.S.: Hosting an object heap on manycore hardware: An exploration. SIGPLAN Not. 44(12), 99–110 (2009)
15. Leung, A., Lhoták, O., Lashari, G.: Automatic parallelization for graphics processing units. In: PPPJ 2009, pp. 91–100. ACM, New York (2009)
16. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A many-core x86 architecture for visual computing. ACM Trans. Graph. 27(3), 18:1–18:15 (2008)
17. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: A comparative analysis. In: PPPJ 2009, pp. 11–20. ACM, New York (2009)
18. Sagonas, K., Wilhelmsson, J.: Efficient memory management for concurrent programs that use message passing. Sci. Comput. Program. 62(2), 98–121 (2006)

19. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann, San Francisco (2001)
20. Snir, M., Otto, S.W., Walker, D.W., Dongarra, J., Huss-Lederman, S.: MPI: The Complete Reference. MIT Press, Cambridge (1995)
21. Dewolfs, D., Broeckhove, J., Sunderam, V.S., Fagg, G.E.: FT-MPI, fault-tolerant metacomputing and generic name services: A case study. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 133–140. Springer, Heidelberg (2006)
22. Odersky, M., Altherr, P., Cremet, V.: The Scala language specification. Technical report, EFPL, Lausanne, Switzerland (April 2004)
23. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
24. Germain, G.: Concurrency oriented programming in termite scheme. In: ERLANG 2006, p. 20. ACM, New York (2006)
25. Lee, J., et al.: Python actor runtime library (2010), `osl.cs.uiui.edu/parley/`
26. Wiger, U.: What is Erlang-style concurrency? (2010), `http://www.ulf.wiger.net/weblog/2008/02/06/what-is-erlang-style-concurrency/`
27. Intel Software Network: Intel trace analyzer and collector (2011), `http://www.software.intel.com/en-us/articles/intel-trace-analyzer/`
28. Erlang Online Documentation: Trace tool builder (2011), `http://www.erlang.org/doc/apps/observer/ttb_ug.html`
29. Zhao, J., Madduri, S., Vadlamani, R., Burleson, W., Tessier, R.: A dedicated monitoring infrastructure for multicore processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 19(6), 1011–1022 (2011)
30. The ProTest Project: Framework 7 project 215868 (2008-11), `http://www.protest-project.eu`
31. Arts, T., Fredlund, L.A.: Trace analysis of Erlang programs. SIGPLAN Not. 37(12), 18–24 (2002)
32. Nystrom, J.: Analysing Fault Tolerance for Erlang Application. PhD thesis, Division of Computer Systems, Uppsala University (2009)
33. Thompson, S., et al.: Wrangler Tool (2011), `http://www.cs.kent.ac.uk/projects/wrangler/`
34. Gartner Group: Gartner says Cloud consumers need brokerages to unlock the potential of Cloud services (2009), `http://www.gartner.com/it/page.jsp?id=1064712`
35. Eucalyptus Systems: Cloud APIs (2010), `http://www.eucalyptus.com/blog/2010/03/11/cloud-apis`
36. The RELEASE Project: Framework 7 project 287510 (2011-14), `http://www.release-project.eu`
37. Chechina, N., Trinder, P., Ghaffari, A., Green, R., Lundin, K., Virding, R.: The design of scalable distributed (SD) Erlang. In: Draft Proceedings of IFL 2012, pp. 461–476. Oxford University (2012)
38. Aronis, S., et al.: BenchErl: A Scalability Benchmark Suite for Erlang/OTP (2012), `release.softlab.ntua.gr/bencherl/`

Towards Heterogeneous Computing without Heterogeneous Programming

Miguel Diogo and Clemens Grelck

University of Amsterdam, Institute of Informatics
Science Park 904, 1098XH Amsterdam, Netherlands
diogo@science.uva.nl, c.grelck@uva.nl

Abstract. From laptops to supercomputer nodes hardware architectures become increasingly heterogeneous, combining at least multiple general-purpose cores with one or even multiple GPU accelerators. Taking effective advantage of such systems' capabilities becomes increasingly important, but is even more challenging.

SaC is a functional array programming language with support for fully automatic parallelization following a data-parallel approach. Typical SaC programs are good matches for both conventional multi-core processors as well as many-core accelerators. Indeed, SaC supports both architectures in an entirely compiler-directed way, but so far a choice must be made at compile time: either the compiled code utilizes multiple cores and ignores a potentially available accelerator, or it uses a single GPU while ignoring all but one core of the host system.

We present a compilation scheme and corresponding runtime system support that combine both code generation alternatives to harness the computational forces of multiple general-purpose cores and multiple GPU accelerators to collaboratively execute SaC programs without explicit encoding in the programs themselves and thus without going through the hassle of heterogeneous programming.

1 Introduction

Single Assignment C (SaC)[1] is a purely functional array programming language for compute-intensive and performance-sensitive applications. SaC has a syntax that very much resembles C, yet it is a fully-fledged programming language in its own right with context-free substitution of expressions as the driving principle of program execution. SaC features truly multidimensional arrays as the main data aggregation principle as well as shape- and dimension-invariant definitions of array operations. Arrays are truly stateless and, thus, functions (conceptually) consume their array arguments and produce new array values. The focus of the SaC project is on compiler technology that transforms high-level functional specifications into executable code that competes well with C or Fortran through aggressive compiler optimization and fully compiler-directed parallelization[2].

Heterogeneous computing receives increased attention due to the rise of powerful and affordable accelerator hardware, such as general purpose GPUs, the

IBM Cell processor, and FPGAs [3, 4]. These accelerators can provide speedups of one or two orders of magnitude depending on the problem [5, 6, 7]. Most available heterogeneous programming environments [8, 9, 10] provide a way to offload computations to one of many types of accelerators. These approaches mainly solve portability issues, but leave programmers alone with low-level, architecture-specific coding of applications, and thus very much limit productivity the more heterogeneous the target architectures become.

With all the above approaches, a selected accelerator does the heavy work while the host CPU is typically idle, as well as any other available accelerators. Making effective use of heterogeneous computing environments may well provide additional performance gains, as shown by the MAGMA project [11] in the area of algebraic computations. Further studies demonstrate performance gains by sharing computations between multiple GPUs and CPU cores [11, 12, 13].

Our approach taken for SaC is more ambitious than the ones above: from the very same declarative and architecture-agnostic program we fully automatically generate efficient parallel code for multi-core CPUs [14] and many-core GPUs [15]. However, for now a choice needs to be made at compile time to either utilize one or multiple multi-core CPUs or a single CUDA-enabled GPU.

In this paper we propose the necessary compilation and runtime system technology to realize additional performance gains by simultaneously using one or more multicore CPUs and one or more GPUs to cooperatively execute array operations without sacrificing our declarative, compiler-directed approach. This requires the integration of the currently separate multicore and CUDA compilation paths within the SaC compiler. Various issues need to be addressed from high-level program transformations over code generation to dealing with explicit memory transfers between the various memories involved.

The CUDA backend of the SaC compiler generates CUDA-kernels from SaC array operations. Only such kernels can run on CUDA-enabled devices. They are subject to several constraints imposed by the underlying hardware. For example, the absence of a stack on GPUs rules out function calls, and branches are costly as they result in parts of the GPU being masked out during execution. Consequently, CUDA kernels must follow relatively simple patterns of nested loops, and complex array operations must often be split into several consecutive kernels. In contrast, the SaC multicore backed puts a lot of effort into improving data locality in cache-based systems [20], which often results in fairly complex codes that would be impossible or at least inefficient to run on GPUs. This example shows that the choice of target architecture does not only affect the final compilation stages of target code generation, but already needs to be taken into account in central parts of the compilation process.

GPUs have their own memory, which makes heterogeneous computing systems also distributed memory systems. Explicit data transfers between the host memory and the various GPU device memories must be introduced into the generated code. For good performance it is crucial to avoid redundant data transfers. The SaC CUDA backend schedules entire array computations on the (single)

GPU. With this trivial static scheduling, the compiler can make all data movement decisions statically, and only entire arrays are transferred between host and device memories. If multiple GPUs and host cores are to collaborate in a single array operation, we must dispense with this static model. Instead (conceptual) arrays must be distributed across the various memories of a heterogeneous computing system. Consequently, as such an array becomes the argument of a subsequent array operation, parts of this array are likely not to be present in the memory of the host or device that needs the data. Heterogeneity naturally limits the applicability of static scheduling in such a context, and we expect dynamic approaches to be more effective in the long run [16, 17, 13]. Therefore, the runtime system needs to keep track of where which parts of an array are stored and initiate the necessary data transfers to make the data available where it is needed. In light of these problems, we specifically aim to:

– track location of computed array slices dynamically;
– perform as few copies from/to device memory as possible;
– avoid redundant copying between different memories;
– generate two code versions for each array operation, one optimized for CUDA and the other for multicore execution.

The remainder of the paper is organized as follows. In Section 2 we provide some background information on SaC and in Section 3 on relevant existing work in the SaC compiler. Section 4 describes how to concert the CUDA and multicore compilation paths. In Section 5 we introduce the notion of distributed arrays. Section 6 shows how this proposed scheme integrates with existing SaC dynamic scheduling facilities. In Section 7 we present and analyze preliminary experimental results. Eventually, we discuss some related work in Section 8 and draw conclusions in Section 9.

2 Single Assignment C

As the name suggests, SaC leaves the beaten track of functional languages and adopts a C-like syntax to ease adoption by imperative programmers. Core SaC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions [1, 18]. Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs show exactly the operational behaviour expected by imperative programmers. This allows programmers to choose their favourite interpretation while the compiler exploits the side-effect free semantics for advanced optimization and automatic parallelization.

On top of this language kernel SaC provides genuine support for processing truly multidimensional and truly stateless/functional arrays advocating a rank- and shape-generic programming style. Array types include arrays of fixed shape, e.g. `int[3,7]` for a 3x7-matrix of integers, arrays of fixed rank, e.g. `float[.,.]`

for a float matrix of any size and arrays of any rank, e.g. `double[*]` for a double array of any rank, which could be a vector, a matrix, etc. SaC only provides a small set of built-in array operations, essentially to retrieve the rank (`dim(`*array*`)`) or shape (`shape(`*array*`)`) of an array and to select elements or subarrays (*array*`[`*idxvec*`]`). All aggregate array operations are specified using with-loop expressions, a SaC-specific array comprehension:

```
with {
  ( lower_bound <= idxvec < upper_bound) : expr;
   ...
  ( lower_bound <= idxvec < upper_bound) : expr;
}: genarray( shape, default)
```

The key word `genarray` makes this with-loop define an array whose shape is given by the ***shape*** expression and whose elements default to the value of the ***default*** expression. The body consists of multiple (disjoint) *partitions*. Here, *lower_ bound* and *upper_ bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to induction variables in for-loops. We call the specification of such an index set a *generator* and associate it with some potentially complex SaC expression. Thus, we define a mapping between index vectors and values, in other words an array. We deliberately use index *sets*, which make any with-loop expose fine-grained concurrency and thus the ideal basis for our parallelization efforts.

```
...
foo = with {
        (. <= iv <= .): 20;
      }: genarray([1000, 1000]);

y = foo[[1,20]] + 1;

bar = with {
        ([0,10] <= iv <= .): foo[iv] + y;
      }: modarray(foo);

bar[[1,2]] = 10;
...
```

Fig. 1. Example SaC code, consisting of a `genarray`, a `modarray`, and a few array operations in between

We illustrate SaC code in general and the with-loop in particular by the (artificial) code fragment of Fig. 1. This code creates a `1000x1000` matrix where each element is set to `20` using the `genarray` with-loop. The dots in the lower and upper bound expression positions are syntactical sugar for the least and the greatest legal index vector of the defined array. So, the generator covers the entire index space, thus making an explicit default expression obsolete. We then access this new array at some index and name the result `y`. Next, we define a new matrix `bar` based on the existing matrix `foo` using a `modarray` with-loop, a variant of the `genarray`-with-loop introduced above. Matrix `bar` has the same rank and shape as matrix `foo`. Likewise the values of the leftmost 10 columns are taken from the corresponding values of `foo`. The remaining columns covered by the generator are also taken from `foo`, but incremented by the value of `y`. Finally, we change element `[1,2]` of the new matrix to `10`. More precisely, we create a third matrix also named `bar` that is identical to the previously existing matrix `bar` except for the value at row 1, column 2.

3 Existing Work in the SAC Compiler

3.1 Generating Multi-threaded Code

The SAC-compiler takes considerable effort to condense multiple (often computationally light-weight) with-loops into fewer, more heavy-weight ones by means of high-level code transformations [19]. This typically results in complex multi-generator with-loops. Subsequently, such with-loops undergo a transformation into what is called the *canonical order* [20]. This canonical order is an internal representation of with-loops as a single, fairly complex (imperfect) nesting of loops, whose execution follows the way elements are stored in memory, rather than as a sequence of simple, perfect loop nests that would arise from the individual compilation of generators. This technique considerably improves performance on conventional architectures through increased cache utilization.

The multicore code generator then selects individual with-loops for multi-threaded execution based on a simple cost model and creates parallel sections with the necessary communication and synchronization facilities. At runtime, the main thread executes any sequential code as normal, while worker threads remain inactive. Before starting one of the parallel sections and waking up the worker threads, the main thread sets up a memory area, known as task frame, where it inserts any variables the worker threads require. On a shared memory system arrays are not copied to the task frame, only their references. For with-loops in parallel sections, a scheduler assigns disjoint regions of the array index space to each thread for computing. Threads repeat the with-loop computation with different regions until the whole iteration space is assigned by the scheduler. The default strategy is block scheduling, but programmers can assign other static and dynamic ones to each with-loop.

3.2 Generating CUDA Kernels

The first step towards generating CUDA kernels lies in selecting those with-loops that can actually run on the GPU, e.g. it is impossible to call functions within kernels. With-loops with function calls in the body (after optimizations like inlining) thus take the standard compilation route, while selected with-loops go through CUDA-specific transformations.

The SaC CUDA backend [21, 15] maps individual with-loops to CUDA kernels. The canonical order, however, is not suitable for CUDA as the corresponding loop structure becomes too complicated and the GPUs can much less benefit from this form of data locality. Hence, each generator is individually compiled into one kernel instead.

In order to statically distinguish between host-allocated and device-allocated arrays in a sound way, the CUDA backend employs a simple type system where every array type is tagged as either *host type* or *device type*. Relatively free array variables inside CUDA with-loops are converted from a host type to the corresponding device type by means of the primitive function `host2device`. Conversely, arrays computed by a CUDA with-loop (thus having a device type) can be converted to a host type by the corresponding primitive function `device2host`. At runtime these type conversion functions take care of the necessary memory transfers. It is noteworthy that the SaC CUDA backend takes considerable effort to avoid unnecessary memory transfers through high-level code transformations.

Consider the code from the example in Fig. 1. As both with-loops are CUDA-compatible, the code is transformed to look like in Fig. 2. Note that to compute `bar_cuda` there is no need to transfer `foo` back to device memory as `foo_cuda` already resides there and as a functional data structure remains unchanged.

```
...
foo_cuda = with_cuda {...}: genarray([1000, 1000]);

foo_host = device2host(foo_cuda);
y = foo_host[[1,20]] + 1;

bar_cuda = with_cuda {...}:modarray(foo_cuda);
...
```

Fig. 2. Excerpt from the code of Fig. 1, both with-loops selected for CUDA

4 Compiling With-Loops for Multiple Targets

The same with-loop will produce very different code whether it is compiled for CUDA or for multicore execution. To solve this incompatibility, we generate both versions side-by-side. The first step is to allow existing compiler procedures to mark all with-loops capable of running on CUDA. Then, these with-loops are duplicated, and the copy is deselected for CUDA execution. This way, one of the with-loops can go through the CUDA-specific transformations while the other goes through the multicore-specific transformations and among others is transformed into canonical order.

The resulting code for our running example is shown in Fig. 3. To simultaneously represent the two with-loop variants, these are encapsulated in a conditional. For illustration we use the keywords `with_cuda` and `with_host`.

The conditional will later be moved to parallel sections executed by multiple threads in SPMD-style, and the predicate will be evaluated by each thread individually. One host thread per GPU will launch the CUDA kernels. The remaining worker threads will run multithread code on the host directly.

```
...
y = foo_host[[1,20]] + 1;

if(cudaBranch) {
  bar_cuda = with_cuda {...}:modarray(foo_cuda);
} else {
  bar_host = with_host {...}:modarray(foo_host);
}

bar_host[[1,2]] = 10;
...
```

Fig. 3. Excerpt from the SaC code in Fig. 1 after with-loop duplication

5 Managing Multiple Memories

The existing CUDA backend only transfers arrays in their entirety between host and (one) device memory. This is not viable if we are to execute with-loops simultaneously on multiple devices or on device and host. To keep track of distributed arrays scattered across multiple memories, we introduce a third type: distributed variables. To access the array, it first needs to be converted from distributed type to a concrete type through explicit conversion. Whenever these arrays are written to, however, they again must be converted to the distributed type to make the runtime system aware of any changes. To represent these conversions we introduce dedicated primitive functions: `dist2host` and `host2dist` for the host, `dist2device` and `device2dist` for CUDA devices.

5.1 Distributed Variables

To keep track of distributed arrays scattered over different memories we introduce a control structure inside distributed variables. This structure has references to the concrete array addresses on the host and on the GPUs as well as a table recording which parts of an array are present where. Currently, we restrict ourselves to dividing arrays into blocks along the outermost axis.

To avoid unnecessary copying, the dynamic tracking scheme should also be able to detect cases where data is available in several memories simultaneously. The proposed scheme addresses this using techniques derived from cache coherency protocols [22]. The idea is to keep a table in host memory tracking the state of each block on each device. The possible states are (M)odified, (S)hared and (I)nvalid, mimicking the simple MSI cache coherence protocol [23], as illustrated in Fig. 4.

This proposed control structure allows for some scaling, as it is possible to track more devices simply by adding another row to the control structure table. Each block currently corresponds to a position along the outermost dimension. For a matrix, for example, each block would correspond to a row. We find this to be reasonable in most situations, and it simplifies implementation.

	Blocks ⇒				
Host	S	M	I	I	...
CUDA 1	S	I	M	I	...
CUDA 2	S	I	I	M	...
...					

Fig. 4. Dynamic tracking control structure

5.2 Dedicated Conversion Functions

The primitive functions to convert from a distributed to a concrete type or vice-versa make use of the control structure in distributed variables to copy only the required parts to the memory system that needs them. These functions can be thought of as extensions to the `device2host` and `host2device` primitive functions introduced by the SaC CUDA backend. Instead of explicitly copying the whole array from one memory to another, the new primitive functions transparently perform the copying of only those data blocks needed but not present in some memory. This is possible because the compiler always knows which parts of an array will be accessed, even if just in a symbolic manner. We can thus pass a range of elements as an argument to the conversion functions.

```
int_dist[*] host2dist(int_dist[*] dist_array, int[*] host_array, int[*] range)
{
  foreach (block i in range) {
    mark_block(i, host, Modified)
    mark_block(i, cuda, Invalid)
  }
  return dist_array;
}

int[*] dist2host(int_dist[*] dist_array, int[*] host_array, int[*] range){
  foreach (block i in range) {
    if( get_block_mark(i, host) == Invalid) {
      copy_block(i, cuda, host)
      mark_block(i, cuda, Shared)
      mark_block(i, host, Shared)
    }
  }
  return host_array;
}
```

Fig. 5. Pseudo-code for the `host2dist` and `dist2host` primitive functions

Pseudo code implementations of `host2dist` and `dist2host` are shown in Fig. 5. For each block that falls into the specified range, `host2dist` sets the corresponding entry in the distributed variable table to *Modified* on the host and to *Invalid* on the CUDA devices. For each block that falls into the specified range, `dist2host` checks the corresponding entry in the distributed variable table for whether or not the host line is set to *Invalid*. If not, it is done with the block; otherwise, the data is copied from a CUDA device, and both host and device entries are set to *Shared*. The CUDA device functions `device2dist` and `dist2device` are very similar. Note that the distribution control data structures always resides in host memory.

5.3 Making Use of the Type Extensions

Continuing the example of Fig. 3, a new code transformation creates the distributed variables and inserts the required conversion functions. This transformation makes the code look like in Fig. 6. With-loops now operate only on a certain range, as the conversion functions need this information. This range information will eventually come from the scheduler, but for now we use a placeholder function. Note that unlike in the CUDA-only example of Fig. 2, it is now necessary to explicitly convert `bar_dist` to `bar_host`. Also note that distributed variables have to be initialized before the concrete arrays, so that we can update the control structure afterwards.

```
...
y = foo_host[[1,20]] + 1;

bar_dist = initialize_dist_var(...);
range = rangeOracle();
if (cudaBranch) {
  foo_cuda = dist2device(foo_dist, foo_cuda, range);
  bar_cuda = with_cuda { ...  | range}:modarray(foo_cuda);
  bar_dist = device2dist(bar_dist, bar_cuda, range);
} else {
... /* same as above, using host code */
}

bar_host = dist2host(bar_dist, bar_host, [1,2]);
bar_host[[1,2]] = 10;
bar_dist = host2dist(bar_dist, bar_host, [1,2]);
...
```

Fig. 6. Excerpt of the SaC code from Fig. 3 after insertion of distributed variables and primitive conversion functions

6 Code Generation

During code generation both with-loop versions and the corresponding type conversions must be integrated with the multicore parallel sections and scheduler. Our approach is to create a parallel section around the conditionals with both with-loop code variants, rather than single with-loops. The predicate becomes a check for a specific kind of threads, which either launch CUDA kernel or execute sections of a with-loop directly on the host.

Continuing the example from Fig. 6, the above transformation yields the code in Fig. 7. For brevity, we again only consider the `bar` with-loop; effects on the `foo` with-loop are very similar. First, in the master thread, we start a parallel section, so that the worker threads wake up and run the worker function. We pass the function and its arguments through the task frame to the worker threads before the master thread joins parallel execution as worker #0.

```
...
  y = foo_host[[1,20]] + 1;

  bar_dist = initialize_dist_var(...);
  _start_parallel_section(&_spmd_bar, bar_dist, foo_dist, foo_cuda, foo_host);
  _spmd_bar(0, bar_dist, foo_dist, foo_cuda, foo_host);

  bar = dist2host(bar_dist, bar_host, [1,2]);
...

void _spmd_bar( thread_id, bar_dist, foo_dist, foo_cuda, foo_host)
{
  if( _isCUDAthread( thread_id) ) {
    do {
      range, continue = _getSubset( thread_id, ...);

      foo_cuda = dist2device(foo_dist, foo_cuda, range);
      bar_cuda = with_cuda {... | range}:modarray(foo_cuda);
      bar_dist = device2dist(bar_dist, bar_cuda, range);
    } while (continue);
  } else {
    do {
      range, continue = _getSubset( thread_id, ...);

      foo_host = dist2device(foo_dist, foo_host, range);
      bar_host = with_host {... | range}:modarray(foo_host);
      bar_dist = device2dist(bar_dist, bar_host, range);
    } while (continue);
  }

 _sync_barrier( thread_id)
}
```

Fig. 7. The `bar` with-loop of Fig. 6, after creating worker thread functions: original with-loop context (top) and lifted SPMD-style worker function (bottom)

In the parallel section we have the conditional with the two with-loop variants. Both branches are very similar, the only differences are that one uses the CUDA conversion functions and the CUDA with-loop code while the other uses the host conversion functions and the multicore with-loop code. The predicate `_isCUDAthread` uses the thread id to choose the correct branch for each thread.

All threads then enter a loop. Each thread obtains some subset of the elements to compute from the scheduler (represented here as the `_getSubset()` function). The next step is converting the with-loop dependencies from a distributed type to the appropriate concrete type. We do not convert the whole array though, only the parts we need for computing the specific iteration space the scheduler assigned to this thread. The conversion functions will take care of copying any required missing data from the assigned area. Then, we can perform the computation on this partial concrete array, generating a partial concrete result. The last step in the loop is to update the distributed variable. As long as there are sections of the array to compute, the scheduler sets the output flag `continue` to true, so threads will continue to query the scheduler for a new subset of the array to compute. Once the whole array has been assigned, threads will start to exit the loop and hit a synchronization barrier (`_sync_barrier()`). When all threads are done, the program resumes sequential execution.

7 Preliminary Results

7.1 Benchmark Code

To test the viability of the system described in the previous section, we decided to implement it manually based on a relatively simple but nonetheless representative program. We chose a prototypic stencil computation where over a number of iterations each element in a matrix is set to the arithmetic mean of its four direct neighbours in the previous iteration. There are many ways to write this code in SaC (for examples see [18]), but in one way or another high-level program transformations lead to code similar to the program fragment in Fig. 8. Note that even though the computation itself is rather simple, communication and synchronization are inevitable between iterations of the for-loop, which makes this a clearly non-trivial scenario.

```
for (k=0; k<LOOP; k++) {
  A = with {
      ( . < x < . ) : 0.25f*( A[x+[1,0]] +A[x-[1,0]] +A[x+[0,1]] +A[x-[0,1]]);
    } : modarray( A );
}
```

Fig. 8. Computational kernel of 2-d convolution example

We compiled this program with both the multicore and the CUDA backend to obtain original intermediate C codes. We then manually combined these codes implementing the concepts presented so far. Instead of fully dynamic scheduling, however, we divided arrays statically into two parts: one part is divided equally among the CPU cores, the other among the GPUs. Nonetheless, locations of array slices are indeed tracked "dynamically", following the scheme presented in Section 5. The distributed memory control structure is consulted on every iteration, even though the mapping remains the same throughout execution.

7.2 Experiment 1: Host Plus 2 GPUs

We use the Distributed ASCI Supercomputer (DAS-4) [24] for evaluation. Our first experiment runs on a dual hexa-core 2.67 GHz Intel Xeon node equipped with two NVidia GTX480 GPUs. We run our benchmark for 2000 iterations on a double precision floating point matrix of 9000x9000 elements with different SaC backends: sequential, multicore, CUDA and our proposed hybrid scheme. We use `gcc` with optimization level `-O3` as binary code generator and report the shortest time out of 3 runs. We experiment with different work division schemes

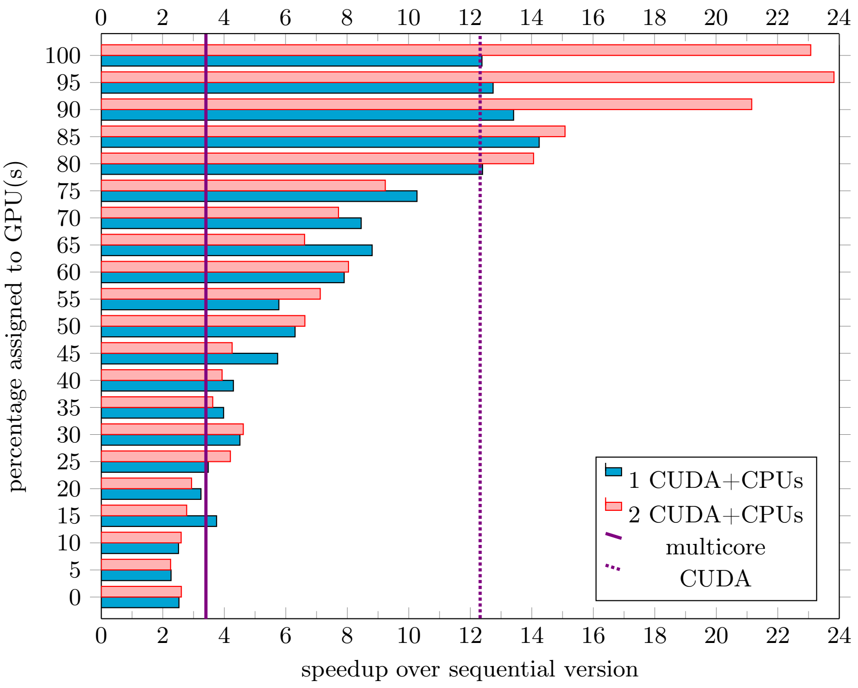


Fig. 9. Comparison of the runtimes between different work division percentages; results for the CUDA and multicore backends are shown as vertical lines

and compare the performance we achieve using the hybrid scheme with that of the existing CUDA and multicore backends. Results are shown in Fig. 9.

In Fig. 9, we see that while the CUDA backend provides a reasonable speedup of about 12, the multicore backend performs very poorly, with a speedup of just over 3 for 12 threads. The bottleneck is most likely the host memory system. There is so little computing that much CPU time is spent waiting for memory load instructions to complete. In previous experiments we did achieve almost linear speedups for the same kind of code on smaller matrices.

Comparing the performance of our implementation with the others, we see that using the GPU(s) with less than 25% of the iteration space introduces so much overhead that it is better to just use the CPUs. To actually outperform the CUDA-only implementation, one has to use the GPU(s) for at least 80% of the workload. Assigning the whole array to the CPUs with our implementation is slower than the existing multicore backend. The other way around, assigning the whole array to one GPU, achieves practically the same run time as the existing CUDA backend. We believe this is due to the fact that the CUDA-bound threads can perform the bookkeeping while its assigned GPU is computing the result, thus overlapping overhead with computing. In contrast, CPU-bound threads must do both computation and bookkeeping on the CPU.

With one GPU, our solution achieves the best results with 15% of the workload assigned to the host and 85% assigned to the one GPU. The added benefit from using the CPUs is small, but still significant. Using only the two GPUs, we practically double the performance compared with the existing CUDA backend. The best results are achieved with 95% of the workload assigned to the GPUs and 5% assigned to the CPUs. This, however, results in a marginal benefit only. Although these results show that it is not worthwhile to use the CPUs when more than one GPU is available for this particular problem, other problems, containing sections that cannot be parallelized for the GPU, for example, can still benefit from being able to use all the available CPU cores, rather than running sequentially as is happening today.

7.3 Experiment 2: Up to 8 GPUs

To explore the scalability of simultaneously using multiple GPUs we run experiments on another node of the DAS-4 system equipped with two quad-core 2.4 GHz Intel Xeon CPUs and 8 NVidia GTX580 GPUs. In this experiment we increase the problem size to 10000 iterations and 13000x13000 elements. Given the results of the first experiment we leave out the host and focus on increasing numbers of GPUs instead.

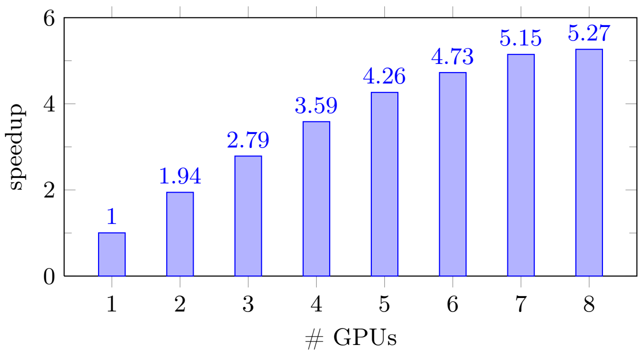


Fig. 10. Speedups over the CUDA backend obtained with our hybrid approach for a varying number of GPUs, without CPU assistance

Considering the results shown in Fig. 10, we see that while using two GPUs provides a nearly two-fold speedup, using 3 or more GPUs provides significantly less improvements. This can be attributed to an increase in communication while the problem size remains constant. As the iteration space is block-wise divided between the different GPUs, two GPUs have only one communication partner while any additional GPU has two. We try to minimize the impact of communication by not communicating redundant data. We believe it does a good job, as the speedup obtained with 8 GPUs over the single GPU version is on par with that obtained with the multicore version over the sequential one.

8 Related Work

We are not aware of other functional approaches to programming heterogeneous systems such as the one we presented in this paper. Recent projects such as Accelerate [25] for Haskell or Microsoft's Accelerator for F# [26] facilitate using one GPU, but lack support for using the CPU simultaneously or for multiple GPUs. This work also differs from our's in general spirit: while Accelerate offers a form of high-level CUDA programming with skeletons in Haskell, with SaC we strive for a completely compiler-directed solution.

On the imperative programming side, a number of heterogeneous computing environments have recently been developed. The approach has mostly been to define and implement APIs. One such heterogeneous environment is Qilin[27]. Qilin is built on top of the Intel Threading Building Blocks (TBB) and the Nvidia CUDA libraries to support both multicore CPUs and Nvidia GPUs. Compilation is done dynamically, Qilin generates the machine code at runtime using static scheduling of operations between the CPU and GPU. This static scheduling however is adaptive. The Qilin system keeps track of operation runtimes in a database, and uses this information to generate a better static schedule on each program run. Qilin was shown to adapt well to varying input sizes, but the amount of offline training required is unclear.

StarPU[22, 28] defines *codelets* as non-preemptible offloadable tasks. Programmers must provide the corresponding kernel functions themselves though. They also have to specify the data dependencies of the *codelets*, which then allows the StarPU runtime system to efficiently schedule tasks to each of the available computation resources. The runtime system also provides a data management facility. Data is divided into disjoint subsets using *filters*, which can be applied recursively. Memory consistency uses a write-back model and a directory-based protocol so that each piece of data has a state associated with it: modified, shared, or invalid. Additionally, it is possible to define custom schedulers and *codelet* drivers for different architectures. With our work in SaC, we intend to introduce a similar system, yet simpler. One significant difference is that we intend to make existing SaC code readily able to make use of heterogeneous computing environments. Specifically, we do not want the programmer to specify codelets or data dependencies, rather should the SaC-compiler infer them automatically.

Song et al. [29] developed a new methodology for matrix computations using multi-core CPUs and multiple GPUs. In this work, a new tiling algorithm is introduced as well as corresponding partition and load balancing schemes. From a SaC perspective, the tiling schemes of Song et al. are rather specific, and out of scope of current research. The work on minimizing communication would be interesting to apply on a SaC scheduler, however.

Ravi et al. [13] describe a heterogeneous computing system for map-reduce applications. Programmers only need to annotate the reduction processing structure, the compiler then generates code for multi-core CPUs and GPUs. At runtime, the work is distributed dynamically. Support for reduction operations is an integral part of SaC. Some results from their scheduling scheme may prove useful for further work in a SaC scheduler.

9 Conclusion

Nowadays, a typical computer contains at least one multicore CPU and one GPU. Together these resources provide significant computing power, but their heterogeneity requires a heroic programming effort to effectively harness it. In this paper we presented compiler and runtime systems extensions for SaC to effectively exploit heterogeneous hardware resources by concerting the existing separate compilation paths for multicore CPUs and single GPUs and adding support for arrays dynamically distributed across multiple memories. Thus, we create support for systems with multiple multicore CPUs and multiple GPUs without requiring programmers to explicitly code for such systems. Preliminary experiments based on a hand-coded prototype show encouraging results.

We are currently implementing the proposed techniques in the SaC-compiler. Once completed, we will investigate a larger range of existing SaC benchmarks and applications. Future work beyond this will be in two directions. Firstly, we aim at completing fully dynamic scheduling between CPU and GPUs. Secondly, we would like to generalize the proposed techniques from host/device memories to network-connected cluster nodes, potentially each equipped with GPU accelerators.

References

[1] Grelck, C., et al.: SAC: a functional array language for efficient multi-threaded execution. International Journal of Parallel Programming 34(4), 383–427 (2006)
[2] Wieser, V., et al.: Combining High Productivity and High Performance in Image Processing Using Single Assignment C on Multi-core CPUs and Many-core GPUs. Journal of Electronic Imaging 21(2) (2012)
[3] Chamberlain, R., et al.: Visions for application development on hybrid computing systems. Parallel Computing 34(4), 201–216 (2008)
[4] Kumar, R., et al.: Heterogeneous chip multiprocessors. Computer 38(11) (2005)
[5] Guo, Z., et al.: A quantitative analysis of the speedup factors of FPGAs over processors. In: Field Programmable Gate Arrays, Monterrey, CA, USA (2004)
[6] Che, S., et al.: A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing 68(10), 1370–1380 (2008)
[7] Williams, S., et al.: The potential of the cell processor for scientific computing. In: 3rd Conference on Computing Frontiers, Ischia, Italy. ACM (2006)
[8] RapidMind Inc.: Writing Applications for the GPU Using the RapidMind^{TM} Development Platform (2006)
[9] Papakipos, M.: The PeakStream platform: High-Productivity software development for multi-core processors. Technical report, PeakStream Inc. (2007)
[10] Dolbeau, R., et al.: HMPP^{TM}: A hybrid multi-core parallel programming environment. In: General Purpose Processing on Graphics Processing Units, Boston, MA, USA (2007)
[11] Tomov, S., et al.: MAGMA Users' Guide. University of Tennessee (2010)
[12] Horton, M., et al.: A Class of Hybrid LAPACK Algorithms for Multicore and GPU Architectures. In: Application Accelerators in High-Performance Computing, Knoxville, TN, USA (2011)

[13] Ravi, V., et al.: Compiler and runtime support for enabling reduction computations on heterogeneous systems. Concurrency and Computation: Practice and Experience 24(5), 463–480 (2011)
[14] Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming 15(3), 353–401 (2005)
[15] Guo, J., et al.: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: Declarative Aspects of Multicore Programming, Austin, TX, USA (2011)
[16] Hummel, S., et al.: Load-sharing in heterogeneous systems via weighted factoring. In: Parallel Algorithms and Architectures, Padua, Italy, pp. 318–328. ACM (1996)
[17] Boyer, M., et al.: Automatic Intra-Application Load Balancing for Heterogeneous Systems. In: AMD Fusion® Developer Summit 2011, Bellevue, Washington, USA (2011)
[18] Grelck, C.: Single Assignment C (SAC): High Productivity meets High Performance. In: Zsók, V., Horváth, Z., Plasmeijer, R. (eds.) CEFP. LNCS, vol. 7241, pp. 207–278. Springer, Heidelberg (2012)
[19] Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing 32(7+8), 507–522 (2006)
[20] Grelck, C., et al.: On code generation for multi-generator with-loops in SAC. In: Koopman, P., Clack, C. (eds.) IFL 1999. LNCS, vol. 1868, pp. 77–94. Springer, Heidelberg (2000)
[21] Guo, J.: Compilation of SAC to CUDA. PhD thesis, University of Hertfordshire, Hatfield, UK (2012)
[22] Augonnet, C., et al.: A unified runtime system for heterogeneous multi-core architectures. In: Euro-Par 2008, Las Palmas, Spain (2008)
[23] Papamarcos, M., et al.: A low-overhead coherence solution for multiprocessors with private cache memories. Computer Architecture News 12(3), 348–354 (1984)
[24] DAS-4: Distributed ASCI Supercomputer 4, `http://www.cs.vu.nl/das4/`
[25] Chakravarty, M., et al.: Accelerating Haskell array codes with multicore GPUs. In: Declarative Aspects of Multicore Programming, Austin, TX, USA (2011)
[26] Microsoft Research: An Introduction to Microsoft Accelerator v2 (July 2012)
[27] Luk, C.K., et al.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Microarchitecture, New York, NY, USA (2009)
[28] Augonnet, C., et al.: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In: Euro-Par 2009, Delft, Netherlands (2009)
[29] Song, F., et al.: Efficient Support for Matrix Computations on Heterogeneous Multi-core and Multi-GPU Architectures, University of Tennessee (2011)

On Using Erlang for Parallelization

Experience from Parallelizing Dialyzer[*]

Stavros Aronis[1] and Konstantinos Sagonas[1,2]

[1] Department of Information Technology, Uppsala University, Sweden
[2] School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
{stavros.aronis,kostis}@it.uu.se

Abstract. Erlang is a functional language that allows programmers to employ shared nothing processes and asynchronous message passing for parts of applications which can naturally execute concurrently. This paper reports on a non-trivial effort to use these concurrency features to parallelize a widely used application written in Erlang. More specifically, we present how Dialyzer, consisting of about 30,000 lines of quite complex and sequential Erlang code, has been parallelized using the language primitives and report on the challenges that were involved and lessons learned from engaging in this feat. In addition, we evaluate the performance improvements that were achieved on a variety of modern hardware. On a 32-core AMD "Bulldozer" machine, the parallel version of Dialyzer can now complete the analysis of Erlang/OTP's code base, consisting of about two million lines of Erlang code, in about six minutes compared to more than one hour twenty minutes that the sequential version (still) requires.

1 Introduction

In recent years more and more developers realize that the use of functional languages allows for faster development, both during rapid prototyping and for deployment. Code written in such languages is usually more succinct and can be organized and maintained more easily than in imperative languages. The productivity of developers is therefore enhanced and applications are also easier to maintain. On the other hand, the major arguments against using functional languages focus on performance compared with imperative code. No matter how many optimizations the compiler implements, no matter how the runtime system is structured, imperative programs are, in most cases, faster.

In the modern multicore architectures, however, functional languages found an unexpected ally. As developers need to take advantage of the multiple processors to improve the performance of their applications, techniques for parallelization came into the picture. Code can be parallelized more easily if written in languages with characteristics similar to those of functional languages. Function purity, for example, can ensure that if we need to apply the same function on different data we can simply allocate each computation on a different process and then collect the results. Process creation however, is in itself not a pure operation and requires some special constructs in the language.

In Erlang one can simply use the `spawn` primitive with a function closure as its argument to have a new process that will evaluate the closure on the first free *scheduler*

[*] This work has been partially supported by the European Union grant IST-2011-287510 RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software.

of the runtime system[1]. Together with the `send` and `receive` operations for message passing, this can lead to very simple and efficient parallelization, either when developing an application from scratch or when modifying an existing one. Nevertheless, very few attempts have been made to parallelize existing applications in Erlang.

In this paper we present our experiences from such an attempt, focusing on how using Erlang made this effort easier in some cases and trickier in others. We also identify parallelization approaches that perform well on the main implementation on Erlang (the Erlang/OTP distribution). Finally, given that the application we are parallelizing (Dialyzer) is a tool that is widely used by the Erlang community, we measure the performance improvements we obtained on real use cases.

In the next section we describe briefly the features of Erlang that are relevant to this paper followed by a description of Dialyzer in Section 3. In Sections 4–6 we describe the main design decisions we took, explaining why they were preferred over alternatives, evaluating them and comparing them with relevant previous work. We end with a performance evaluation section, followed by a section with concluding remarks.

2 Erlang: A Concurrent Functional Language

Erlang is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly code reloading, automatic memory management and support for multiple platforms [1]. Its main implementation, the Erlang/OTP (Open Telecom Platform) system, is open source and used by companies around the world to develop software for large commercial applications. Erlang/OTP is a set of libraries and design principles providing middleware to develop systems in Erlang. Nowadays, Erlang applications are significant both in number and in code size, making Erlang one of the most industrially relevant functional languages. Dialyzer analyzes Erlang code and the parallel version uses some of the language's main features; some familiarity with them is thus required to follow this paper.

Erlang code is organized in modules that export some of their functions and keep others private. Each source file defines a single module. Functions in a module may call local functions of the same module and functions from other modules only when these are exported. In Fig. 1 we show part of the callgraph that is formed by the functions in the `lists` module. In the graph, functions that have mutual calls to each other form *strongly connected components* (SCCs), that have been grouped in one node. The highlighted SCCs contain functions that are exported.

Erlang functions can create new processes using the built-in function `spawn`, which accepts a function as an argument (this will be the function that the created process will evaluate). The return value of `spawn` is the *process identifier* (*PID*) of the newly spawned process. Processes can use these PIDs to send messages to each other. Message passing is asynchronous and a call to the `send` operation returns immediately. Processes can receive messages using a `receive` statement, which has a structure identical to that of a `case` statement: it blocks until the *process mailbox* has messages and then uses *pattern matching* to selectively remove one of them and continue execution.

[1] Erlang's runtime system uses threads from the operating system to run a number of schedulers, which manage Erlang's processes as *green threads* that do not share state.

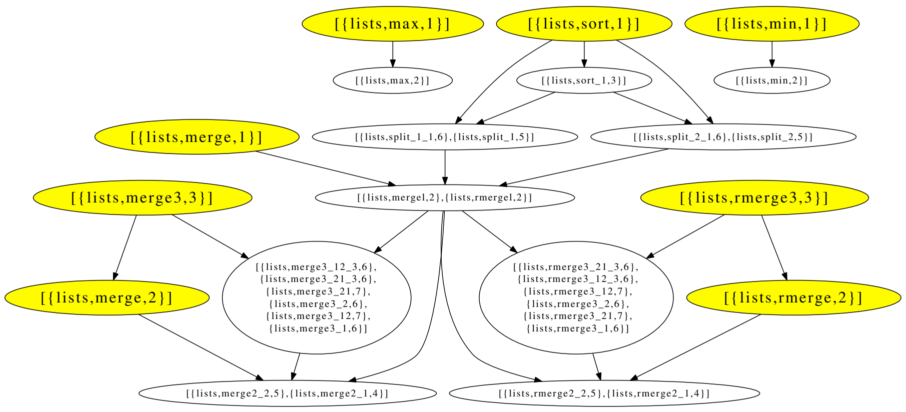

[{lists,max,1}]
[{lists,sort,1}]
[{lists,min,1}]
[{lists,max,2}]
[{lists,sort_1,3}]
[{lists,min,2}]
[{lists,merge,1}]
[{lists,split_1_1,6},{lists,split_1,5}]
[{lists,split_2_1,6},{lists,split_2,5}]
[{lists,merge3,3}]
[{lists,mergel,2},{lists,rmergel,2}]
[{lists,rmerge3,3}]
[{lists,merge,2}]
[{lists,merge3_12_3,6}, {lists,merge3_21_3,6}, {lists,merge3_21,7}, {lists,merge3_2,6}, {lists,merge3_12,7}, {lists,merge3_1,6}]
[{lists,rmerge3_21_3,6}, {lists,rmerge3_12_3,6}, {lists,rmerge3_12,7}, {lists,rmerge3_2,6}, {lists,rmerge3_21,7}, {lists,rmerge3_1,6}]
[{lists,rmerge,2}]
[{lists,merge2_2,5},{lists,merge2_1,4}]
[{lists,rmerge2_2,5},{lists,rmerge2_1,4}]

Fig. 1. Part of the callgraph of the `lists` module

```erlang
parent() ->
  ParentPID = self(),
  ChildPid = spawn(fun() -> child(ParentPID) end),
  receive {ChildPid, Result} -> Result end.

child(ParentPID) ->
  ParentPID ! {self(), do_something()}.
```

Listing 1. Trivial code illustrating Erlang's concurrency primitives

An example is given in Listing 1, where an Erlang process executing the `parent` function spawns a new process that will evaluate the `child` function and send a message to the waiting parent process. The built-in function `self` is used to get the PID of the process evaluating it. This is the simplest form of process interaction. In Erlang, processes may also monitor other processes, associate and register a PID with a symbolic atom that can then be used in the place of the PID or perform other advanced operations which were not used in our parallelization attempt.

3 Dialyzer's Overview

Dialyzer [6] is a static analysis tool included in the Erlang/OTP distribution since 2007. It can detect a wide variety of discrepancies (e.g. type errors, software defects such as exception-raising code, hidden failures, unsatisfiable conditions, redundancies such as unreachable code, race conditions, etc.) in single modules or entire applications. Dialyzer is totally automatic, easy to use and particularly successful in identifying software defects which may be hidden in Erlang code, especially in program paths which are not exercised by testing. It is widely used within the Erlang community.

In the heart of Dialyzer lies a soft typing system. Its purpose is essentially to capture the biggest set of terms for which it can be proven that type clashes will occur. The type signatures that Dialyzer infers, called *success typings* [4], are the complement

```
dialyzer(Input) ->
  InitialState = initialize_state(Input)
  FinalState = fixpoint(InitialState)
  case should_generate_warnings(Input) of
    true  -> generate_warnings(FinalState)
    false -> store_success_typings_in_plt(FinalState)
  end.

fixpoint(State) ->
  State1 = find_types(State),
  State2 = refine_types(State1),
  case are_states_equal(State, State2) of
    true  -> State2;
    false -> fixpoint(State2)
  end.
```

Listing 2. Dialyzer's algorithm in a nutshell

of that set of terms. To infer success typings, Dialyzer uses an iterative approach consisting of two separate phases: the first is based on constraint solving and the second on dataflow analysis. In this paper we will refer to the constraint-solving phase using a routine named *find_types* and to the dataflow phase as *refine_types*. Initially all functions are assumed to have a success typing of $(any(), any(), \ldots, any()) \rightarrow any()$, meaning that they accept all terms as arguments and return an arbitrary term as a result. After the *find_types* phase, any functions that have a tighter success typing than before, together with any functions that call them, are given as input to the *refine_types* phase and vice-versa until all success typings reach a fixed point. If Dialyzer is checking for discrepancies, it makes one last pass over the code of all modules to produce warnings. This pass' inner mechanics are identical to the *refine_types* phase, only this time, instead of merely tightening the success typings, warnings are also emitted when a discrepancy is detected. Alternatively, Dialyzer can also save the success typings for all the exported functions in order to use them when analyzing other code. In this case the success typings are stored in an external *persistent lookup table* (PLT) file. A high-level view of the algorithm used by Dialyzer is given in Listing 2.

Within each of the *find_types* and *refine_types* phases the "unit of work" is the SCC. In *find_types* Dialyzer uses the SCCs of functions, analyzing one at a time, while in *refine_types* it converts those into module SCCs and analyzes all the functions in those modules together. For simplicity, in the rest of this paper we will use the term SCC to refer to both (function and module) cases.

A Note about Amdahl's Law. In every parallelization attempt, the best possible result, regardless of language and implementation, is to maximize the part of the program which runs in parallel and thus hit the inherent speedup limit imposed by Amdahl's Law. Our initial goal was therefore to organize the parallel version of Dialyzer to minimize its sequential parts and push this limit as far as possible.

In Listing 2 we see four different phases: *initialize_state*, *find_types*, *refine_types* and *generate_warnings*. Drawing from previous experience [7] and performing measurements of our own we found that *find_types* is the most time consuming phase, followed by *refine_types*, then *generate_warnings* and finally *initialize_state* (storing success typings in the PLT has negligible cost). Not wanting to perform major changes in the structure of the application, we decided that we will keep the synchronization points between these distinct phases and focus on how we could parallelize their inner workings.

4 Distributing Work

We will now describe how the tasks within each phase are distributed into workers. As an example we will try to parallelize an excerpt of the *find_types* phase using Erlang's primitives. The sequential *find_types* phase processes a list of SCCs in the way shown in Listing 3 to find the success typings for the functions in them. For each SCC the code of the functions is retrieved from the state and analyzed.

```erlang
sequential_find_types(SCCs, State) ->
  FoldFun = fun (SCC, Acc) -> find_type(SCC, Acc, State) end,
  Results = lists:foldl(FoldFun, [], SCCs),
  NewState = update_types(Results, State).

find_type(SCC, Acc, State) ->
  Code = retrieve_code(SCC, State),
  Type = analyze_code(Code, State),
  [{SCC, Type}|Acc].
```

Listing 3. Sequential *find_types*

We will initially ignore any dependencies and assume that all the elements in the SCCs list can be analyzed at the same time. In that case, the parallelization could look like the one shown in Listing 4, where a separate *worker* is used to find the success typings of each SCC.

```erlang
parallel_find_types(SCCs, State) ->
  ParentPID = self(),
  FoldFun = fun (SCC, Counter) ->
                spawn(fun () -> find_type(SCC, ParentPID, State) end),
                Counter + 1
            end,
  Workers = lists:foldl(FoldFun, 0, SCCs),
  Results = receive_results(Workers, []),
  NewState = update_types(Results, State).

find_type(SCC, ParentPID, State) ->
  Code = retrieve_code(SCC, State),
  Type = analyze_code(Code, State),
  ParentPID ! {SCC, Type}.

receive_results(0, Acc) -> Acc;
receive_results(N, Acc) ->
  receive Result -> receive_results(N-1, [Result|Acc]) end.
```

Listing 4. Parallel *find_types*

The problem with such a scheme is that each process has its own memory area in Erlang; the heap is not shared. This design decision was made, among other reasons, to enable independent garbage collection, allowing processes to continue their execution, while others reclaim unused memory on their heaps without "stopping the world". This design decision, however, implies that all data that will be used by a process must be *copied* to its heap, both at process spawning (this includes all the arguments that are being passed to the initial function) and during message passing. Therefore, passing a large data structure as argument leads to a new copy of that structure in the new process.

Returning to Listing 4, we can notice that in this way each worker will receive a copy of the original `State` which contains the code of *all* the functions, as the sequential version of Dialyzer stores all the code in a dictionary, which is just a regular Erlang term. There are two different approaches that can be used to fix this: the first is to "wrap" the original dictionary in a separate process and convert the `retrieve_code` function to send a message with a request to that process and retrieve the code. However with many processes sending such requests at the same time, the dictionary server process would become a bottleneck, as the requests would necessarily be serialized. Using multiple such processes would be a possible workaround, but would complicate the dictionary's operations (as distribution and/or replication of the keys should be taken into account).

The second approach is to use Erlang Term Storage (ETS) tables, which are data structures for associating keys with values, but in essence are just collections of Erlang tuples. The most common operations they support are insertions and lookups. Their implementation is highly efficient – they can store colossal amounts of data (if enough memory is available) and perform lookups in constant (or in some cases logarithmic) time. The reason behind this is that although ETS tables look as if they were implemented in Erlang, they are in fact implemented in the underlying runtime system and have different performance characteristics than ordinary Erlang objects.

When parallelism enters the picture, however, ETS tables are used for a different reason: they provide shared memory. Depending on the table's access restrictions[2], many processes may read and write on a table using a *table identifier* (*TID*), returned upon table creation. This design does not eliminate copying though: currently when a tuple is inserted into an ETS table, all the data structures representing the tuple are copied from the process' heap into the ETS table. When a lookup operation is performed on a table, the tuples are copied back from the ETS table to the heap of the process. On the other hand, the TID itself is small in size so each process can take it as an argument and use it to retrieve from the table *only* the information that it needs.

ETS tables support special optimizations, which are enabled using a set of flags on table creation. The flags we used for Dialyzer's tables are the following[3]:

[2] A table is always owned by a process and access rights might allow: only the owner to read and write to it (`private`), read operations to be performed also by other processes (`protected`), or both read and write operations to be performed by any process (`public`).

[3] More details can be found in the official documentation [3].

compressed: The table data is stored in a more compact format to consume less memory. The downside is that it will make table operations slower. Compression is a significant feature, as it can reduce the size of data up to an order of magnitude, depending on the size of an individual entry.

read_concurrency: This option makes read operations much cheaper; especially on multicore systems with multiple physical processors. However, switching between read and write operations becomes more expensive.

Effect on Dialyzer. The *find_types* phase is not the only one to use this worker-oriented approach; workers are in fact used in all phases. During initialization, for example, Dialyzer determines which files will be analyzed, what kind of code discrepancies the user is interested in, whether user annotations in the source code will be taken into account and other parameters for the analysis. The code from each file is then stored in a codeserver, the full callgraph is generated and the contents of any existing PLT file are loaded into the application's working PLT (which will also be used to store the success typings for the code under analysis). As no order needs to be preserved, we can parallelize the initial storing of the code by letting all the files be processed in parallel. Moreover, in the *generate_warnings* phase (after the success typings have reached a fixed point), we can again assign a worker to each module to find discrepancies.

The codeserver, callgraph and PLT are all the state structures that are interesting for the parallelization and we now briefly explain their role in the application. We also describe some conversions performed in order to make them more suitable for use in the parallel version.

Codeserver: Dialyzer works on the Core Erlang [2] representation. During initialization, all the modules that are given as input need to be transformed into Core Erlang and stored in a dictionary-like data structure in such a way that it is easy to retrieve the code of either a specific function or all the functions in a module. This phase is also referred as 'compilation' of the code under analysis.
The sequential version of Dialyzer implements the codeserver's dictionary using Erlang's `dict` library [3]. As a lot of modules can be given as input to be analyzed together, the code is stored in compressed binary format to reduce its size. The keys of the dictionary are the module names, making the retrieval of a specific function a two-step process: first the code of all the functions of the module that contained the requested function is retrieved and decompressed and subsequently a traversal selects the requested function. The motivation behind this organization is that compressing larger terms yields better overall compression. (When Dialyzer was initially developed memory consumption was a significant concern.) To improve performance, a *caching mechanism* is used; all the phases that require retrieval of code from the codeserver are organized so that they bundle together as many requests for functions from a single module as possible before moving to another.
In the parallel version it makes little sense to use this caching mechanism, as we want to be able to analyze functions from different modules at the same time. For this reason, the organization of the dictionary was changed, making a separate entry for each function. This makes the retrieval of a single function easy but in turn makes the retrieval of all the functions in a module a multi-step process, as each

function must be looked up separately. This was not a major issue, however, as the retrieval of all the functions is used only in *refine_types* and *generate_warnings*. These phases, even combined, consume significantly less time that the *find_types* phase which retrieves the code for specific functions each time.

Finally, as discussed in the previous section, the `dict` was substituted with an ETS table. This table has both the read concurrency (as the input data can still be large in size) and compression features enabled (as after the initial write only concurrent read operations will be performed). In combination, the changes in the structure of the codeserver and the conversion to ETS table showed no changes in performance of the sequential version as the faster lookup for particular functions was countered by the need to copy the actual code out of the ETS table.

Callgraph: Having loaded all the code to be analyzed, Dialyzer can generate the callgraph. Erlang/OTP has libraries for working with graphs, which even provide functions for condensation (to form the SCCs) and topological sorting (`digraph` and `digraph_utils` [3]). However for reasons that will become clear later we need to be able to find the dependencies between different SCCs in a concurrent way. We could assign the whole callgraph in a dedicated server process, but as we explained earlier this would create a bottleneck. Instead we chose to store each SCC together with all the SCCs that have calls to it and all SCCs that the SCC itself calls in an ETS table, with read concurrency.

PLT: Dialyzer contains in a hard-coded library the success typings of all the Erlang functions that are not written in Erlang but have implementations in C. However, all other functions, including those in libraries written in Erlang and included in Erlang/OTP, are susceptible to change and hard-coding information about them would be problematic. Analyzing them from scratch every time an analysis on some application code is performed would also be impractical. Instead Dialyzer uses a Persistent Look-up Table (PLT) to store the success typings for code that has already been analyzed.

Dialyzer has an internal PLT data structure which is initialized with the contents of the PLT file and extended with the information that is inferred about the code under analysis. This data structure was also a dictionary, therefore we were able to directly convert it into an ETS table. As this PLT is used to store success typings throughout the analysis, we did not enable the read concurrency optimization, as reads and writes will be interleaved.

5 Coordination

The workers in the initialization and warning generation phases do not require any special coordination, as each module is processed only once. In *find_types* and *refine_types* phases, on the contrary, the order of the analysis of the SCCs is important, as it affects the number of iterations to reach the fixed point. This imposes coordination requirements to the workers used in these phases which will be explained and addressed in this section. First, however we will explain the need for a fixed point computation and the techniques that the sequential version uses to reach the fixed point faster[4].

[4] More information about the algorithms used in the two analysis phases are beyond the scope of this paper. The reader is directed to the relevant publication [6].

5.1 Fixed Point Computation

For the first *find_types* phase of the analysis Dialyzer has to form SCCs out of all the functions and traverse those in reverse topological order to perform the analysis in a bottom-up manner. In that way, functions appearing later in the order can use the results that the analysis has produced. After this first *find_types* phase, all functions with success typings tighter than the initial one advance to the *refine_types* phase.

For the *refine_types* phases Dialyzer generates a module callgraph and uses a similar approach, with module-SCCs. Again, functions with tighter success typings, together with any callers they have advance to another *find_types* phase. In general, functions that have calls to any functions that are advancing to a next phase should also advance, as possible tighter success typings for the callees can result in tighter success typings for the callers. This continues for as long as there exist functions that have not reached a fixed point. Experience shows that for typical Erlang code this requires three to four iterations. Within each *find_types* and *refine_types* phase the sequential analysis uses the following algorithm:

1. From the whole callgraph, obtain the subgraph that is induced by the set of the functions currently in our worklist.
2. If this is a *refine_types* analysis, convert the induced callgraph into the relevant module callgraph.
3. Condense the callgraph into SCCs.
4. Process the SCCs in reverse topological order, to make sure that SCCs with dependencies are analyzed after all their dependencies have been analyzed.

5.2 Coordinating the Workers Efficiently

Step 4 in the previous list introduces a serialization which is not required. A simple example is the case of all the functions which have no calls: all these can be analyzed immediately in the beginning of the *find_types* phase. In a similar way, the analysis of a function-SCC with calls to a set of other function-SCCs can start as soon as all the members of the set have been analyzed. The same arguments apply to module-SCCs in the *refine_types* phase.

In order to have many workers running in parallel, Dialyzer could use a *coordinator process* that would keep track of these dependencies using a dictionary, with each key being an SCC and the two values associated with it being the sets of all the SCCs that are called by it and call it. Initially the coordinator process could spawn worker processes for all the SCCs which had no calls to other SCCs. The coordinator could then wait for all the spawned workers to finish before spawning the next "level" of SCCs without dependencies. Notice however that with the callgraph being a tree-like data structure it is expected to be narrower on the higher levels and might also be significantly imbalanced. Therefore this approach would not perform well enough.

Instead as soon as a worker process finishes the analysis of an SCC, the coordinator should be notified to check all the SCCs that have calls to it and remove the SCC that was just analyzed from the respective sets. For each set that becomes empty due to this operation, the coordinator should then immediately spawn a new worker to analyze the corresponding SCC. The algorithm is correct, but one can argue that in this way the coordinator becomes a bottleneck, as the notifications will be serialized at it. Both these coordinator-based approaches were used in a previous attempt to parallelize Dialyzer but the performance results were unsatisfactory [7] and that code was never integrated in Dialyzer's code base.

With Erlang being a language that can support a significant number of concurrently running processes, we instead opted to start by spawning *all* the worker processes and have each one keep track of its own dependencies. In this scheme, each process needs to know which processes are waiting for its results and notify them directly after it produces them. This algorithm is summarized below:

1. The coordinator spawns all workers, storing a mapping between each SCC and the worker process that will analyze it. This mapping, which should also be stored in an ETS table in order to be accessible by the workers independently, will be used to determine the workers that must be notified after a worker has finished.
2. Each worker initially checks which SCCs it is waiting for:
   (a) If it has no dependencies it can start working on its own SCC.
   (b) Otherwise it waits until it receives a confirmation message from each worker it depends on.
3. After a worker has finished processing its SCC it uses the SCC-to-process mapping to notify all the workers that are waiting for its results. It can also determine whether any of the functions it analyzed had a different success typing, therefore needing to be analyzed again and send this information to the coordinator, which is also keeping track of the workers that have not yet finished.

The initial memory requirements of each worker must be small, as Dialyzer might spawn lots of them (in the order of thousands) in the beginning. However, Erlang/OTP's runtime system creates processes with very small initial memory requirements. For our particular use, each worker initially needs to have the description of the SCC it will analyze, the PID of the coordinator in order to report back the functions that need to advance to a next phase and some way to access information in the callgraph, codeserver and PLT. From the callgraph it figures whether it needs to wait for other workers to finish and when all its dependencies have been satisfied it uses the codeserver and PLT to perform its analysis.

Were we to keep the original `dict`-based representations of the callgraph, codeserver and PLT we would have to send a copy of each to every worker and therefore immediately run out of memory. However, as we described in Section 4, the compact representation of ETS tables keeps these initial memory requirements minimal.

Notice that workers do not have to lookup the PIDs of the workers they are waiting for, as we can allow them to accept any incoming message labeled with the SCC or module of a worker they are waiting for. This can be simplified even further by having the worker just wait for a specific number of "done" messages, each sent by one of the workers that are responsible for one of their dependencies. Information about success typings is exchanged only via the PLT (so each worker will write the new success typings in the PLT before notifying that it has finished).

Another detail concerns the order of spawning. In the sequential version of Dialyzer, SCCs or modules are processed in the reverse topological order produced by the condensed callgraph. For the parallel version we instead need to spawn the modules in the normal topological order, to ensure that each SCC has a registered PID at the time when another wants to notify it that it has finished. Experimental measurements showed, however, that the condensation function provided by the library consumes a significant amount of time in cases where the callgraph has nodes with many dependencies. A simpler data structure was therefore used to keep the dependencies and Dialyzer spawns the workers in a random order as it can no longer use the topological ordering function. Should the lookup for a particular SCC yield no result (as the coordinator has not yet spawned or registered it in the publicly accessible data structure) the worker that has to notify it will try again after a short pause.

Finally, as the main process of Dialyzer cannot continue until all the workers have finished, it can play the role of the coordinator, spawning all the workers and waiting for them to finish. A simple counter is sufficient to know if there are still workers running. After finishing, each worker can directly report which functions did not reach a fixed point to prepare the next phase.

6 Fine Tuning the Parallelization

In this section we describe some more advanced issues that we encountered during the parallelization of Dialyzer. Nevertheless, the conclusions apply to any parallel application that uses many workers.

6.1 Granularity of Parallelization: Big SCCs

The previous parallelization attempt [7] had already identified that the parallelization might be too coarse grained to allow for maximum benefits. This is particularly evident in the case of modules containing automatically generated code. Such code forms SCCs consisting of hundreds of functions and the solving algorithm is slow in such cases as a local fixed point is sought for all the functions of the SCC. We were able to detect this as an issue by noticing that sometimes during *find_types*, Dialyzer had just one active worker before resuming with many workers again. For that reason, we decided to split such big SCCs into child workers, each taking a portion of the original SCC's functions and exchanging information on those through the use of a local PLT. Experimentation showed that a threshold of 40 functions before splitting an SCC gave the best results.

6.2 Throttling the Number of Active Workers

For the phases of compilation and warning generation the tasks performed in parallel do not have any dependencies and could all be allowed to run simultaneously. This unregulated approach, however, showed reduced performance.

- **Compilation:** During compilation we have a lot of I/O requests for the filesystem in order to load the code that will be analyzed. At the same time, each compilation worker writes information to the codeserver and the callgraph. This write contention on the shared data structures and read contention on the filesystem degrades performance.

– **Warning collection:** In the warnings pass we have the same data requests as we had during *refine_types* analysis, meaning that we need all the code in each module as well as the success typings of local and external functions. This causes read contention on all the shared data structures.

To amend this reduced performance, a simple ticket-based throttling mechanism was introduced. When each worker process is spawned, it immediately blocks on a receive statement, waiting for an activation message. The coordinator starts with an initial number of "tickets" and as long as these are not exhausted, after spawning each job it sends an activation message to it. After the tickets are exhausted it keeps the rest of the job PIDs in a queue and activates the next one every time another has finished. In both cases the ticket mechanism reduced the simultaneous access requests on the data structures and improved performance. This encouraged us to try the same regulation mechanism on the other two phases as well. The only difference was that these workers should wait until all the dependencies have been satisfied before asking for "permission to run".

Experimenting with the initial number of tickets we found that the best choice was to have as many tickets as the number of logical cores on the running platform. This leads to a number of active workers equal to the number of schedulers of the system.[5] In this scheme, the scheduling component is able to handle the big number of inactive processes efficiently. Moreover, having only as many workers as schedulers running would ensure that each worker finishes its work completely. Without the regulation, the Erlang/OTP runtime system switches out active workers before they finish, leading to many partially-finished workers that reserve memory space for their heaps. This increases memory consumption and degrades performance. More importantly, it runs the risk that, on machines with little memory, the analysis would fail due to excessive amount of simultaneously used memory.

6.3 Idle Workers: Data Prefetching

As we described in Section 5 both the *find_types* and *refine_types* analyses are implicitly regulated by the callgraph dependencies. However not all the information that the analysis requires depends on the constraints imposed by the callgraph. The information coming from the codeserver (source code and user specifications) can be retrieved, processed and be ready when the PLT-related dependencies have been fulfilled.

We therefore tried to implement a prefetching mechanism which retrieves this information for any worker that waits for just one more SCC, as it would be "imminent to run". This did not work well in practice, as the retrieval and preprocessing requires only little time compared with the generation and solving of constraints (which requires the success typings of any dependencies). Moreover, workers waiting for their final dependency now consume more memory while being idle, as their heaps contain the preprocessed information. This can lead to a big number of processes which are parts of a simply linked chain to retrieve data and keep them for an indefinite amount of time, reducing the memory available to the rest of the system.

[5] If one excludes the coordinator and a few processes supporting the runtime system.

More elaborate schemes could potentially work, but they would need either to increase the amount of work done before the dependencies need to be satisfied or to predict better when the prefetching can happen without delays before the data is used.

7 Evaluation

7.1 Impact on the Code

Dialyzer consists of about 19,000 lines of code plus another 9,500 lines shared with the HiPE compiler [5], the latter containing the hard-coded type information for Erlang's built-ins (5,000 lines) and the functions that are used to manipulate the type representations (4,500 lines). A plain diffstat of our patches for the parallel version reports that 1,800 lines were inserted and another 1,000 lines deleted. The total number of code changes is therefore in the order of 10%. The bulk of the changes was on the files containing the representation and interfaces of the codeserver, callgraph and PLT, which were converted to ETS tables.

7.2 Benchmarks

In order to evaluate the parallelization, we used two benchmarks which correspond to the most typical use cases of Dialyzer. These are building a PLT with the Erlang/OTP distribution and analysis of a big application. We run the benchmarks on two different platforms: the first one is a desktop with an i7-2600 CPU (3.40 GHz) and 16GB of RAM running Debian Linux 2.6.32-5-amd64 (a total of eight logical cores but only four physical). The second is a server with two AMD Opteron(TM) 6274 CPUs (2.20 GHz) and 128GB of RAM running Scientific Linux 2.6.32-220-x86_64 (a total of 32 cores). In the diagrams we have omitted the sequential phases of the computation (e.g. the condensation of the callgraph before each *find_types* and *refine_types* phase).

As mentioned in Section 4, the changes in the tool's core data structures (especially the codeserver) were applied before attempting any parallelization, with no observable differences in performance. Together with the fine-tuning that was described in Section 6, the overall effect was a final parallel version which had the same performance as the old sequential version if only one core was used. This enables us to use the parallel version running on just one core as the baseline for our comparisons.

Building a Standard PLT. As explained in Section 4, creating a PLT is a necessary first step before using Dialyzer. The contents of the PLT, however, depend on the particular application that we want to analyze. Key Erlang/OTP applications like `erts`, `kernel` and `stdlib` are used in almost all applications, while others are not so common. To benchmark the benefits of parallelization in the generation of a PLT, we picked the most common Erlang/OTP applications[6]. This benchmark does not include a warnings pass, as no warnings are reported when generating a PLT. The results are shown in Fig. 2 and 3. Dialyzer requires three rounds of *find_types* and *refine_types* to reach a fixed point for these applications.

[6] Applications included: `asn1 edoc erts compiler crypto dialyzer inets hipe gs kernel mnesia public_key runtime_tools ssl stdlib syntax_tools wx xmerl`

Analysis of the Whole Erlang/OTP. The second benchmark is the analysis of all applications that are included in the Erlang/OTP R15B01 distribution. These applications have a big diversity, including long chains of dependencies both in function-SCC and module levels, small and big function SCCs, hand-written and automatically generated code, as well as a significant use of all the language's features. The results are shown in Table 1 and Fig. 4, 5, 6 and 7. Dialyzer requires five rounds of *find_types* and *refine_types* to reach the fixed point, but we are only showing the three of them as the last two include very few SCCs and the amount of work does not allow for any speedups.

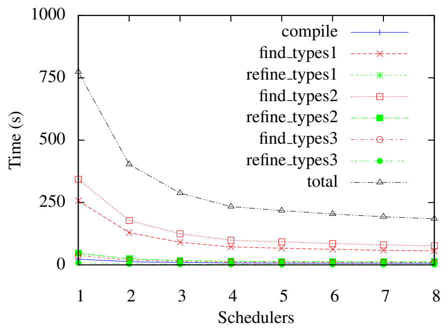


Fig. 2. Time required for PLT generation, 4 cores (8 threads)

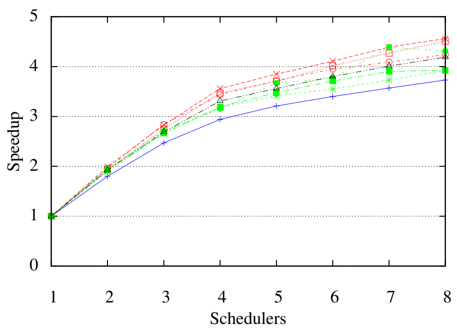


Fig. 3. Speedup diagram for PLT generation, 4 cores (8 threads)

Table 1. Comparing the duration of each phase while Dialyzing OTP, 1 vs. 32 cores

```
              1-core                          32-cores
compile        :  114.67s      compile        :  23.41s       ( 1493 modules)
prepare        :    4.83s      prepare        :   5.59s
order          :   11.16s      order          :  11.47s
find_types 1   : 1408.07s      find_types 1   :  78.61s       (97347    SCCs)
order          :    9.93s      order          :   8.86s
refine_types 1:   240.22s      refine_types 1:   22.39s       ( 1493 modules)
order          :   15.14s      order          :  15.23s
find_types 2   : 2443.59s      find_types 2   : 110.74s       (80323    SCCs)
order          :    6.35s      order          :   5.81s
refine_types 2:   247.81s      refine_types 2:   21.09s       ( 1414 modules)
order          :    0.28s      order          :   0.27s
find_types 3   :   95.45s      find_types 3   :  15.38s       ( 2429    SCCs)
order          :    0.12s      order          :   0.11s
refine_types 3:    28.99s      refine_types 3:    3.15s       (  203 modules)
[round 4 & 5]  :  < 0.50s      [round 4 & 5]  : < 0.50s
warning        :  308.26s      warning        :  23.58s       ( 1493 modules)
       done in 82m29.87s               done in 6m0.80s
```

7.3 Evaluation of the Results

It is easy to observe that the parallelization results vary both between the different phases as well as between the different platforms. For example, the i7-based machine shows linear speedups up to the amount of physical cores and reduced speedups from

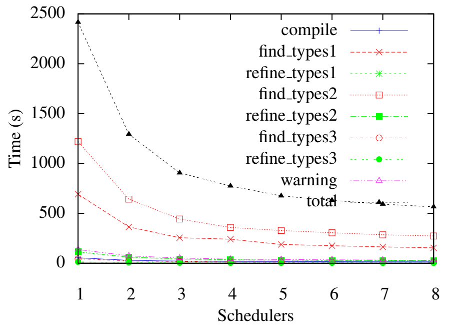


Fig. 4. Time required to Dialyze OTP, 4 cores (8 threads)

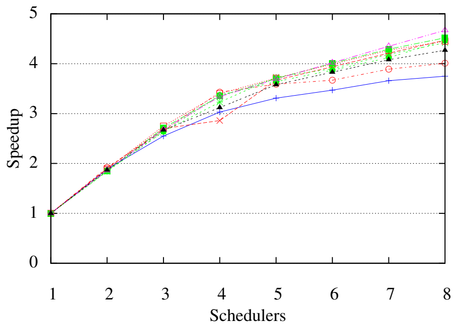


Fig. 5. Speedup diagram for Dialyzing OTP, 4 cores (8 threads)

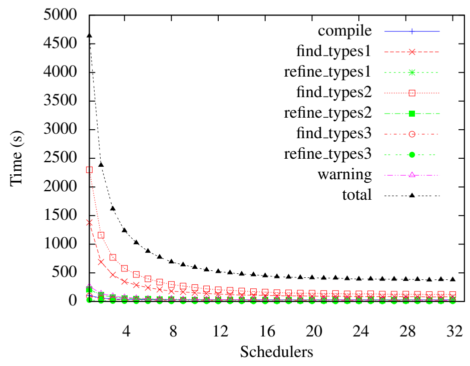


Fig. 6. Time required to Dialyze OTP, 32 cores

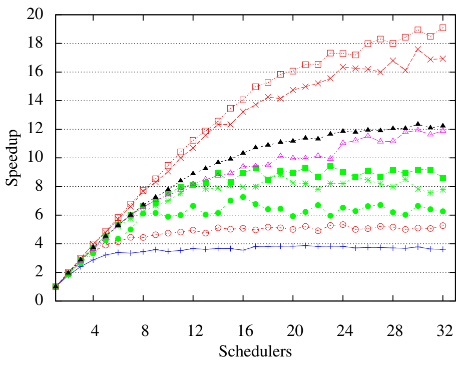


Fig. 7. Speedup diagram for Dialyzing OTP, 32 cores

that point on, which is a typical observation. This is not observed on the 32-core architecture, where all cores are physical. Comparing how these two particular architectures perform, however, is beyond the scope of this paper. The point we wish to make by their choice is that our implementation both has benefits for a user of the tool with a desktop machine, while at the same time being interesting from an scientific point of view as a study of the capabilities of Erlang in parallel programming.

The differences between the phases are more interesting and we will comment on them using Fig. 6 and 7 as a reference. It is evident that different phases have different speedups but the best overall speedup is about 12 times and is reached at 20 cores. The overall implementation clearly outperforms the one in previous work [7], which peaked at a speedup of around 2.5 times on four cores and remained stable up to sixteen cores.

The parallelization of the *find_types* phase has the best results, getting more and more speedup with the addition of every single core up to 32. For the analysis of the OTP function SCCs the first *find_types* pass has 97,347 workers, the second 80,323, the third 2,429 and the fourth and fifth (which are included in the table only) seven and five workers respectively. The reason why the second *find_types* pass takes more time than the first is that even though fewer workers are running, the information about the functions is more rich (due to the intermediate *refine_types* pass) and therefore the analysis has more work to do. This work however is spread among the workers and a greater speedup is achieved as the proportional cost from initialization and communication is smaller. The third *find_types* pass shows reduced speedup as only few SCCs have not reached a fixed point after the second *refine_types* pass.

The *refine_types* passes have fewer workers: 1,493 for the first pass, 1,414 for the second and 203 for the third (in the fourth and fifth pass only two modules have not reached a fixed point). This explains the overall smaller speedups. The performance of the warning-collection pass is similar, as its algorithm is similar to that of the *refine_types* pass.

Finally, the phase that has the worst scalability is the compilation. Here, however, we are bound by the hardware limits on the filesystem and we cannot expect significant speedups. It is fortunate that this phase consumes only a small part of the total time.

8 Concluding Remarks

We presented our efforts to parallelize Dialyzer using Erlang's concurrency primitives. We believe that the techniques we used can be applied to other parallel applications, both when developing from scratch and when evolving them from their sequential counterparts. In particular, we have demonstrated how the Erlang/OTP implementation can efficiently support thousands of processes running concurrently, sharing data via ETS tables and communicating freely with each other for coordination purposes, possibly with some throttling. The final parallel version of Dialyzer has already been included in the R15B02 release of Erlang/OTP.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice Hall Europe (1996)
2. Carlsson, R.: An introduction to Core Erlang. In: Proceedings of the PLI 2001 Erlang Workshop (2001)
3. Ericsson, A.B.: Stdlib reference manual (2011)
4. Lindahl, T., Sagonas, K.: Practical type inference based on success typings. In: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, pp. 167–178. ACM Press, New York (2006)
5. Pettersson, M., Sagonas, K., Johansson, E.: The HiPE/x86 Erlang Compiler: System description and performance evaluation. In: Hu, Z., Rodríguez-Artalejo, M. (eds.) FLOPS 2002. LNCS, vol. 2441, pp. 228–244. Springer, Heidelberg (2002)
6. Sagonas, K.: Experience from developing the Dialyzer: A static analysis tool detecting defects in Erlang applications. In: Proceedings of the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools (2005)
7. Tsavliri, Y.: Parallelizing Dialyzer: a Static Analyzer that Detects Bugs in Erlang Programs. Diploma thesis, School of Electrical and Computer Engineering, National Technical University of Athens (2010)

Author Index