Jay McCarthy (Ed.)

LNCS 8322

Trends in Functional Programming

14th International Symposium, TFP 2013
Provo, UT, USA, May 2013
Revised Selected Papers

Springer

Lecture Notes in Computer Science 8322

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Jay McCarthy (Ed.)

Trends in Functional Programming

14th International Symposium, TFP 2013
Provo, UT, USA, May 14-16, 2013
Revised Selected Papers

Springer

Volume Editor

Jay McCarthy
Brigham Young University
Computer Science Department
3361 TMCB, P.O. Box 26576
Provo, UT 84602-6576, USA
E-mail: jay.mccarthy@gmail.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-45339-7 e-ISBN 978-3-642-45340-3
DOI 10.1007/978-3-642-45340-3
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013956014

CR Subject Classification (1998): D.1, D.3, F.3, 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

From the Crossroads of the West, the 14th Symposium on Trends in Functional Programming took place on the Brigham Young University campus in Provo, Utah, May 14–16, 2013. The program included presentations of 27 papers submitted by researchers from many nations and an invited talk by Jeremy Siek on gradual typing. Most of the authors submitted revisions of their papers, based in part on responses to their presentations. The revisions were reviewed and discussed in detail by the Program Committee, and 10 of them were accepted for publication in this volume. About half of the revisions accepted for publication were student papers (that is, papers with a student as first author).

TFP aspires to be a forum for new directions in functional programming research. This year was no exception. Presentations covered new ideas for distributed systems, education, functional language implementation, hardware synthesis, static analysis, testing, and total programming.

The editor wants to thank the Program Committee and all of the referees for their diligence and for their well-considered reviews. We also want to thank Brigham Young University for their generous support. Finally, we thank the participants for their lively attention during the symposium. Again we leave you, from within the shadows of the everlasting hills; may peace be with you, this day and always.

November 2013 Jay McCarthy

Organization

Program Committee

Sergio Antoy	Portland State University, USA
James Caldwell	University of Wyoming, USA
John Clements	California Polytechnic State University, USA
Wolfgang De Meuter	Vrije Universiteit Brussel, Belgium
Marko van Eekelen	Open University of the Netherlands and Radboud University Nijmegen, The Netherlands
Andy Gill	University of Kansas, USA
Arjun Guha	Cornell University, USA
Jurriaan Hage	Universiteit Utrecht, The Netherlands
Suresh Jagannathan	Purdue University, USA
Rita Loogen	Philipps-Universität Marburg, Germany
Jay McCarthy (chair)	Brigham Young University, USA
Keiko Nakata	Institute of Cybernetics at Tallinn University of Technology, Estonia
Henrik Nilsson	University of Nottingham, UK
Tom Schrijvers	Ghent University, Belgium
Clara Segura	Complutense University of Madrid, Spain
Nikhil Swamy	Microsoft Research, USA
Viktória Zsók	Eötvös Loránd University, Budapest, Hungary

Additional Reviewers

Ki Yung Ahn	Mischa Dieterle	Manuel Montenegro
Nada Amin	Simon Frankau	Aseem Rastogi
Kenichi Asai	Nicolas Frisby	K.C. Sivaramakrishnan
Jasmin Blanchette	Thomas Horstmeyer	Josef Urban
Nicolás Cardozo	Bas Joosten	Bernaard van Gastel
Theo D'Hondt	Magnus Madsen	Robert Zinkov

Sponsoring Institutions

Brigham Young University (USA)

Table of Contents

Total Functional Software Engineering
Overview Paper

Baltasar Trancón y Widemann

Programming Languages and Compilers
Ilmenau University of Technology
baltasar.trancon@tu-ilmenau.de

Abstract. Methods for mathematically basic and precise description of system behavior at discrete interfaces have been developed by David Parnas and his groups and collaborators over many years. Total functions can play a crucial role as constructive and effectively executable semantics for various levels of these descriptions. Straightforward analysis and transformation techniques for functional programs, particularly effective for total functions, can be used as significant steps towards automated generation of implementations. Theoretical claims are supported by practical examples. The focus is on insight into applications from the functional perspective rather than on innovations in functional programming itself.

1 Introduction

The software engineer David Parnas has been influential, besides many other areas, in the development of a particular, mathematically sound methodology for the description and specification of system behavior. The general style of the work of his groups in the Software Quality Research Laboratories at McMaster University, Ontario and the University of Limerick, and many collaborators, can be summarized in the following two maxims:

1. The essential complexity of real-world systems must be acknowledged, and met with appropriately scalable methods, but
2. the mathematics underlying these methods must be as simple and rigorous as possible.

In recent years, it has emerged that many of the proposed methods can be understood in a framework of total functional programming. This paradigm shift, away from the original presentation in elementary set theory, implies multiple illuminating changes in perspective:

1. *Algebraic structure is uncovered.* Method design choices that have been based on practical experience and justified pragmatically by mathematical fitness and economy, can be strengthened theoretically by being mapped to natural algebraic constructs.

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 1–16, 2014.

2. *Executability is emphasized.* In contrast to descriptive set-theoretic models where effective evaluation procedures are implicit meta-information, the functional programming view puts denotational and operational semantics on equal footing.
3. *Tools are leveraged.* Beyond executability in principle, actual implementations are needed at the end of the day, to serve as test oracles, simulators or prototypes. In the traditional set-theoretic approach, there is a choice between confinement to some theorem prover sandbox, and naive translation to a general-purpose programming language, with all the associated pitfalls. The tried and proven tools of functional programming language implementation support symbolic evaluation and code generation in a way that is both reliable and flexible.

The current state of the art is such that the theoretical basis is established fairly comprehensively. By contrast, the practical side of real tools is just academic prototypes with limited scope and service life-time. The purpose of the present paper is to serve as a conceptual reference for future implementations.

The following three sections each discuss one particular method, from basic to advanced. Note that the focus here is on the application of functional technologies outside their principal programming domain. Most of the material discussed in the following is of little novelty from the functional programming perspective proper, but summarizes, condenses, and in places even improves the understanding of the subject with respect to its traditional presentation.

2 Predicate Logic

The Parnas approach to algebraic–logical language [22] differs from most other software engineering methodologies by treating algebraic (value-level) expressions as *partial*, but logic (truth-level) expressions as *total*.

2.1 Two Worlds

The algebraic world is one of strict partial functions: An expression of the form $f(e_1, \ldots, e_n)$ is defined if and only if all subexpressions e_i are defined, and the tuple $(v_1, \ldots, v_n)$ of their respective values is in the domain of f. Denotationally, every value type has a bottom element. By contrast, the logic world is one of total predicates: An expression of the form $p(e_1, \ldots, e_n)$ is true if all subexpressions e_i are defined, and the tuple $(v_1, \ldots, v_n)$ of their respective values is in the extension of p, and false otherwise. That is, the type of truth values has no bottom element. Case distinction operators have condition arguments of truth value type, reflexively embedding the logic world in the algebraic world.

This account of partiality is in line with the IEEE 754 standard for floating-point arithmetics, where comparison operators on numbers totalize in the same way with respect to the bottom value `NaN`. It is, however, distinct from the Z [9] approach, where logic is three-valued with an *undetermined* bottom element,

predicates are strict, but logical connectives are non-strict in both arguments. For example, in the expression $(0/0 = 1) \vee (1 = 1)$, the first clause is false according to [22], but undetermined according to Z, whereas the whole expression is true in either case, albeit for different reasons.

The special role of the bottom element has profound implications on evaluation strategies, which can be seen clearly from basic considerations of denotational semantics of functional programs: A definedness predicate can be formalized within the language, simply as "defined $e \equiv \mathfrak{t}$" which entails that each evaluation strategy must

- either have a solvable halting problem, making the predicate *defined* total,
- or wrongly assign the value bottom rather than zero to some instances of the expression scheme "if_then_else(defined $e, 1, 0$)" by failing to terminate.

2.2 Enter Total Functions

An elegant solution of this dilemma is to restrict the expression language such that it can be interpreted in a strongly normalizing calculus, thus reducing the halting problem to triviality.

Such calculi define total rather than partial functions; the partiality of algebraic expressions is emulated adequately by the monad $M = 1 + \_$, known as *Maybe* in Haskell. In the standard category-theoretic notation, it comes with the natural transformations $\eta_X : X \to M(X)$ (unit, *return* in Haskell) and $\mu_X : M^2(X) \to M(X)$ (multiplication, *join* in Haskell), as well as the family of constants $\bot_X \in M(X)$ as left injections. Partial functions of type $A \rightharpoonup B$ are encoded as $A \to M(B)$. Values of type A are encoded as $M(A)$. Strict, checked application is given by the operator

$$\frac{e : M(A) \quad f : A \to M(B)}{e \triangleright f : M(B)} \qquad e \triangleright f = \mu_B\big(M(f)(e)\big)$$

known as *bind* ($\gg\!=$) in Haskell. An emulation of the intended partial language can then be given by induction over the syntactic structure of expressions:

- Constants and variables are taken to be always defined.

$$\frac{c : A}{c^\dagger : M(A)} \qquad c^\dagger = \eta_A(c)$$

- The pseudo-constant $*$ denotes an atomic undefined expression.

$$\frac{*_A : A}{*_A^\dagger : M(A)} \qquad *_A^\dagger = \bot_A$$

- Tupling (only binary shown for simplicity) is strict.

$$\frac{(e_1, e_2) : A_1 \times A_2}{(e_1, e_2)^\dagger : M(A_1 \times A_2)} \quad (e_1, e_2)^\dagger = e_1^\dagger \triangleright \Big(\lambda\, x_1.\, e_2^\dagger \triangleright \big(\lambda\, x_2.\, \eta_{A_1 \times A_2}(x_1, x_2)\big)\Big)$$

– References to partial functions are Kleisli-extended.

$$\frac{f : A \rightharpoonup B}{f^\dagger : M(A) \to M(B)} \qquad f^\dagger = \_ \triangleright f$$

– References to predicates are totalized sending bottom to false.

$$\frac{p : A \to \mathbb{B}}{p^\dagger : M(A) \to \mathbb{B}} \qquad p^\dagger = [p, \mathrm{const}\,\mathsf{f}]$$

where $[g, h] : A + B \to C$ combines cases $f : A \to C$ and $g : B \to C$.

– Emulation distributes over application.

$$\big(s(e_1, \ldots, e_n)\big)^\dagger = s^\dagger\big((e_1, \ldots, e_n)^\dagger\big)$$

It is easy to see that this emulation preserves well-typing, and extends to equational definitions of functions and predicates.

2.3 Simplification

The administrative operations inserted by the emulation complicate the structure of expressions considerably at first sight. But fortunately, a substantial part can be eliminated by straightforward partial evaluation and simplifications using the monad laws. In particular we have:

$$f^\dagger\big(\eta(x)\big) = \eta(x) \triangleright f = f(x) \qquad p^\dagger\big(\eta(x)\big) = [p, \mathrm{const}\,\mathsf{f}]\big(\eta(x)\big) = p(x)$$

Note that such program transformations are rather easier, and can be applied more aggressively, in a strongly normalizing setting. For instance, consider a definition of partial function composition:

$$\mathit{compose}(g, f)(x) = (g(f(x)))^\dagger = g^\dagger(f^\dagger(x^\dagger)) = \eta(x) \triangleright f \triangleright g = f(x) \triangleright g$$

Here the inner application is reduced to unchecked form because x is necessarily defined, seeing that applications to undefined arguments are handled at the call site. The outer application needs to remain checked for spontaneous undefinedness of the subexpression $f(x)$.

If additionally f is total by construction, that is $f = \eta \circ f'$—not an uncommon case, see for example the definition of tupling as a (strict) partial function given above—, then we can reduce this further and eliminate another check:

$$\mathit{compose}(g, \eta \circ f')(x) = \eta\big(f'(x)\big) \triangleright g = g\big(f'(x)\big)$$

In summary, a reflexive combination of partial algebraic and total logical language can be represented faithfully in a calculus of total functions, by using a well-known monadic lifting. Locally total subexpressions are reduced to their natural form by straightforward simplification of the resulting monadic expressions. This is of course a fairly banal insight in a functional programming context. But it is nowhere nearly as automatic and self-evident in contexts of set-theoretic proof systems or ad-hoc code generators, the standard tools of system engineers, where partiality is an implicit side condition rather than integral part of data flow.

2.4 Discussion: Expressive Power

Of course the proposed framework, based on a calculus of total functions, has a significant limitation: The language that can be interpreted in it has to be quite restricted, compared with the full power of first-order predicate logic that would be available (albeit incompletely operationalized) in a theorem prover environment. Expressing any nontrivial algorithm in terms of constructively total functions is known to be a difficult task.

Fortunately, the existing method base definitions and the examples suggest that very simple data structures and algorithms go a long way. Simple algebraic datatypes and primitive recursive access functions, whose representation in strongly normalizing calculi is well-understood, make up most of the framework. By contrast, particularly troublesome features, notably infinite set comprehensions and general recursive function definitions, are apparently not required.

This finding suggests an interesting, open philosophical question: is the computational simplicity just a happy coincidence, or are mathematically more involved constructs pragmatically ill-suited to the task of behavioral description of systems, because they are harder to understand for the human engineer, or less evident from empirical observation?

3 Tabular Expressions

Classical mathematics are heavily biased in terms of algebraically simple functions which have a homogeneous representation as a simple expression with one or more variables over their whole domain. Mildly heterogeneous definitions such as piecewise definitions for a domain partitioned into intervals are admissible, but more general case distinctions are generally avoided. By contrast, the theory of computation in computer science is discrete by nature, and case distinctions feature pervasively and nestedly in function definitions.

Where case distinctions are made in logically rigorous descriptive formalisms, it is important to ascertain that cases are non-contradictory and complete. Functional programming provides a notation for case distinction that is powerful, theoretically elegant and easy to implement, namely by pattern matching on algebraic datatypes. Unfortunately this approach has little acceptance in system engineering contexts. Engineers traditionally favor a different form, namely tabular expressions, where alternative cases are laid out spatially as columns or rows of a table. The tabular notation has attractive pragmatic advantages [21,24]:

1. It scales from simple two- and three-way distinctions to extremely complex expressions where many-way and/or hierarchical case distinctions along multiple, more or less orthogonal criteria are combined.
2. It is fairly easy and intuitive to read for domain experts without formal training in symbolic programming, and can be used, edited and archived effectively in paper form.
3. It supports manual and machine-supported inspection, validation and verification of descriptions by systematic coverage of rows, columns or cells; applications include soundness and completeness proofs as well as inspection of

safety-critical procedures [23], protocol checks for concurrent systems [18,19] and test suite design [5,6,7].

The discussion here summarizes material exposed in more detail in [28].

3.1 Simple Example

Fig. 1 shows a simple real-world tabular expression. It appeared in inspection documents of the Darlington Nuclear Power Generation Station [16,15]. In the shutdown system, parts of the monitoring logic only apply near maximum power. The pertaining sensors are "conditioned in" (activated) above a certain power level and "conditioned out" (deactivated) below. To avoid a *jitter* effect (high-frequency switching events), the respective threshold levels K_{in} and K_{out} are set to slightly different values, thus introducing artificial hysteresis. Between the two, the previous value is maintained. See the top half of Fig. 1 for an illustration of the behavior, and the bottom half for the tabular description as a function of the threshold parameters, the current power level and the previous value.

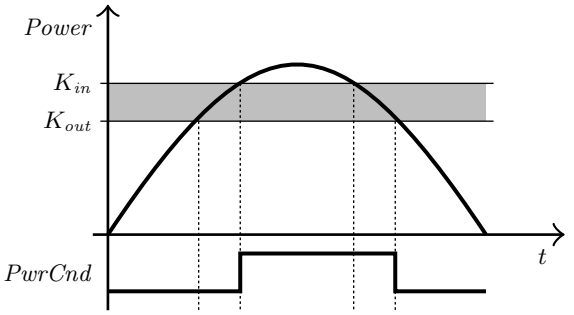
The function definition is organized as a one-dimensional decision table. The top (*header*) (1×3)-grid of truth-level expressions specifies a three-way case distinction. The bottom (*main*) (1×3)-grid of value-level expressions contains the respective function results. Note that values are Boolean by accident, but the main grid is three-valued $(\mathit{true}, \mathit{false}, \bot)$ whereas the header grid is two-valued.

The function is simple in structure and does not appear to merit the tabular form at first sight. But closer inspection reveals a subtlety that illustrates why symmetric case distinctions, as expressed by a header grid, are sometimes superior to if-then-else cascades or first-fit pattern matching: The cases in the upper grid are only consistent and complete under the implicit constraint $K_{out} < K_{in}$. Hence a thorough inspection of the table, either by a human expert or by a theorem prover (such as PVS, which found the error in actual fact [15]), will reveal that the description is formally deficient. By contrast, a cascading definition where each case implies the negation of the preceding, would just silently go wrong.

3.2 Complex Example

Fig. 2 shows a complex tabular expression. It specifies a test procedure for computer keyboards [1]. The general idea is fairly simple: all keys are to be pressed in order, and if all registered keycodes correspond to the expected sequence, the keyboard passes the test. This homogeneous principle is then complicated by provisions for correcting errors both of the hardware and of the human tester by pressing the escape key, without exempting the escape key from the test sequence. The tabular expression is the result of a formal analysis of several pages of prose, eliminating several ambiguities, inconsistencies and loopholes.

The function N specifies the number of the next key to be pressed or a verdict (*Pass* or *Fail*), depending on the sequence T of keys pressed so far. The



$PwrCnd(Prev : bool; Power, K_{in}, K_{out} : real) : bool \equiv$

$Power \leq K_{out}$	$K_{out} < Power < K_{in}$	$Power \geq K_{in}$
false	*Prev*	*true*

Fig. 1. Simple tabular expression: power conditioning for sensors

$N(T) \equiv$

			$T = \_$	$T \neq \_$		
				$N(p(T)) = 1$	$1 < N(p(T)) < L$	$N(p(T)) = L$
$keyOK(T)$				$N(p(T)) + 1$		*Pass*
$\neg keyOK(T)$	$\neg keyEsc(T)$	$pkeyOK(T)$			$N(p(T)) - 1$	
		$\neg pkeyOK(T) \wedge pkeyEsc(T) \wedge ppkeyOK(T)$				
		$\neg pkeyOK(T) \wedge pkeyEsc(T) \wedge \neg ppkeyOK(T)$			$N(p(T))$	
		$\neg pkeyOK(T) \wedge \neg pkeyEsc(T)$	1			
	$keyEsc(T)$	$\neg pkeyEsc(T)$				
		$pekeyEsc(T)$				
		$pkeyEsc(T) \wedge \neg pekeyEsc(T)$			*Fail*	

where $keyOK(T) \equiv r(T) = N(p(T))$ $\quad keyEsc(T) \equiv r(T) = Esc$

$pkeyOK(T) \equiv keyOK(p(T))$ $\quad pkeyEsc(T) \equiv keyEsc(p(T))$

$ppkeyOK(T) \equiv keyOK(p^2(T))$ $\quad pekeyEsc(T) \equiv N(p^2(T)) = Esc$

Fig. 2. Complex tabular expression: computer keyboard checking procedure

constant _ denotes the empty sequence. Nonempty sequences T can be deconstructed into a most *recent* event $r(T)$ and a *previous* sequence $p(T)$.

The tabular form highlights several features that are hard to emulate in functional programming style or any other textual format:

1. The table is two-dimensional: the chosen variant depends on two more or less orthogonal case distinctions. They are specified by the top and left header grids of truth-level expressions, respectively. Together they select a value-level cell of the bottom-right main grid at the spatial intersection of their axes. The choice which of these to evaluate first is completely arbitrary.
2. Each header is hierarchical, having a binary decision tree structure ending in flat two-or-more-way distinctions. The choice of decision criteria and order of flat distinctions is logically arbitrary, but pragmatically highly relevant for good readability of the resulting table: Related cases should end up close together, ideally in adjacent cells of the main grid. Note that the presentation in Fig. 2 improves over the original from [1] in this respect by reordering rows.
3. Homogeneous regions of cases in the main grid, or "modes" of the system, are indicated by invisible cell boundaries. Missing expressions indicate unsatisfiable conditions; those are an artifact of the headers not being fully independent.
4. Case distinctions make effective use of undefined values, discussed in the previous section, for keeping things simple. For instance, all auxiliary predicates involve equations whose parts are defined for nonempty sequences T only; hence the only row that is satisfiable for the column corresponding to T being empty is the fifth, where all predicates occur in negative form. The header expression $T \neq \_$ is redundant and included for making completeness more obvious. More generally, nested distinctions can always be read as conjunctions and simplified accordingly, which would be difficult in logically three-valued frameworks, because one clause may govern the definedness of another.

3.3 Table Combinators

A connection between tabular expressions and functional programming has already been noted by [14]. There, a combinator approach is followed: a collection of compositional table construction operators is given, with executable Haskell implementation for operational semantics, and proof support in the Isabelle system for denotational semantics. While both theoretically elegant and technically effective, the approach suffers from severe limitations regarding the shapes of tables that can be constructed; a drawback shared with both other practical tools such as [25] and theoretical formalizations such as [10,12,11,4].

3.4 General Table Model

The examples have already shown a weakness, or rather looseness, of the tabular notation: Considerable amounts of semantic detail are implicit in the graphical

$$
\begin{aligned}
Content[I, J, X, Y] &= Map[I, Map[J, Expr[X, Y]]] \\
TType[I, J, X, Y] &= \begin{pmatrix} wellf : Content[I, J, X, Y] \times X \to bool; \\ eval \ : Content[I, J, X, Y] \times X \nrightarrow Y \end{pmatrix} \\
Table[I, J, X, Y] &= \begin{pmatrix} content : Content[I, J, X, Y]; \\ type \ \ \ : TType[I, J, X, Y] \end{pmatrix}
\end{aligned}
$$

Fig. 3. Table model as functional datatype

layout or the conventions of users. This is typically adequate for a team of experts, but not for broader communication, formal verification or automated evaluation. Support for certain ad-hoc tabular formats has been built into many engineering tools. These may be pragmatically useful, but there is no theoretical boundary of what should be included; the examples already show both plain one- and two-dimensional forms, and several advanced features, such as branching headers [8,27], and shared and empty main grid cells. Various generalizations (for instance extension to n dimensions, grids with circular or otherwise fancy topology, and specialized case distinctions such as C-style *switch*) are mathematically straightforward, but defeat the capabilities of implemented, hard-wired table models.

A unified theoretical approach [13] defines tabular expressions abstractly as a formal structure of three components:

1. The *content* of the table as an indexed set of indexed grids containing cell expressions. Both grid and cell indices are abstract; no layout geometry is implied. This is the only component particular to a concrete table.
2. A *well-formedness* predicate that decides whether the table content conforms to given shape and consistency constraints. This component is shared among tabular expressions of a common *type*.
3. One or more *evaluation* functions that interpret the cell content, conditional on its well-formedness, as a function of its argument variables. This component is also shared among tabular expressions of a common type.

Examples of *content*, with a conventional graphical layout, are depicted in Fig. 1 and 2. Examples of the other, more generic, components are given informally in section 3.2. An early tool prototype is described in [26].

3.5 Functional Table Model

The general table model also translates smoothly to a total functional context [28]. Fig. 3 shows a datatype definition for table models. Type parameters are I, J for grid and cell indices, and X, Y for domain and range, respectively.

The function argument may occur in each table cell subexpression. Whereas first-order tools default to symbolic representations, in higher-order functional programming the obvious encoding technique is *lambda lifting*: every cell contains an individual function of the global arguments. The transformation is trivial

because cell contents are independent (neither individually not mutually recursive). Consequently we simply have $\mathit{Expr}[X, Y] = (X \nrightarrow Y)$.

The well-formedness predicate may contain both static and dynamic constraints. These could be separated by binding time analysis, in order to be checked as early as possible; cf. [2]. For n-dimensional regular function tables, which subsume the two given examples with $n = 2$, the types parameters are chosen such that:

1. The cell indices of each header grid are sets of finite paths closed under prefixes.
2. The cell indices of the main grid are the Cartesian product of the cell indices of the header grids.

The well-formedness constraints are:

1. There are $n+1$ grids. The first n grids are headers and contain truth-valued expressions. The last grid is the main grid and contains Y-valued expressions.
2. In each header, the respective conjunctions of formulas along maximal paths partition the function domain.
3. For each main index $(j_1, \ldots, j_n)$, if the formulas indexed by j_i in the i-th header, respectively, are jointly satisfiable for all i, there is a corresponding cell; other main cells may be omitted.

We treat hierarchical header structure as a syntactic abbreviation for simplicity. The corresponding evaluation function is:

1. For each i-th header grid, find the maximal path j_i such that the conjunction of all formulas in cells along the path is satisfied.
2. Evaluate the main grid cell at $(j_1, \ldots, j_n)$.

The table type effectively defines a semantic checker and interpreter for the table content. Obviously when both type and content are provided, these algorithms can be simplified drastically by partial evaluation of the pair. Assume well-formedness is split by binding time analysis into a static and a dynamic part

$$\begin{aligned} &\mathit{swellf} : \mathit{Content}[I, J, X, Y] \to \mathit{bool};\\ &\mathit{dwellf} : \mathit{Content}[I, J, X, Y] \times X \to \mathit{bool} \end{aligned}$$

Then we can partially evaluate the table type components applied to the concrete content, obtaining the following record of table operations, which hides the defining content completely:

$$\begin{pmatrix} \mathit{pswellf} & : \mathit{bool};\\ \mathit{pdwellf} & : X \to \mathit{bool};\\ \mathit{peval} & : X \nrightarrow Y \end{pmatrix}$$

Possibly more static safety is required, such as a guarantee that the "compiled" evaluation function *peval* is defined whenever the dynamic well-formedness

check *pdwellf* holds. The strong connection between total function calculi and constructive proof systems, exemplified in tools such as Coq or Agda, could be used to resolve these issues statically. Simple but useful prover support has been given for the table combinators of [14], by virtue of their inductive structure. No practical attempts to perform these tasks manually or automatically for the general table model have been documented so far.

4 Trace Function Method

The trace function method is a formalism for black-box description of observable system or component behavior at an interface; see for instance [17]. It is intended as a mathematically simple and direct replacement for algebraic and automata-theoretic approaches, as well as the earlier trace assertion [3,20] and trace rewriting [33,32] methods.

A *trace* is a sequence of relevant events at some interface, where each event is a discrete point in time at which interface variables may change their values. Input and output variables under control of the environment and the system, respectively, are unified. Valid system behavior is specified by giving, for each output variable, a trace function or relation, which maps traces to possible output values for the final event. The set of valid traces is then defined inductively:

- The empty trace is valid.
- A valid trace can be extended by a following event if and only if all output variables of that event conform to the respective trace functions.

Having the trace, that is the sequence of preceding interface events, instead of internal state as the causal determinant of future behavior makes this approach mathematically and epistemologically very abstract and elegant. However, there are a couple of logical, philosophical and technical issues:

1. Trace functions specify part of an event, namely the value of one output variable, in terms of traces ending in that event; how can we avoid circularity?
2. The proposed way to avoid circularity is to include only the input part of the most recent event in the argument to trace functions; is that solution natural, and is it uniquely so?
3. Trace functions take syntactically well-formed, as opposed to valid, traces as their arguments (necessarily to avoid meta-circularity); how do counterfactual values in invalid traces ("fake history") affect the specification of behavior?
4. A trace function has two distinct ways of depending recursively on its own value for trace prefixes, namely by retrieval from events and by recursive self-application; are they exchangeable, and if not, which one is preferred?
5. Trace functions can depend on variables in past events in many ways: on the last event only, on a fixed sliding window, on the most recent event matching a certain pattern, etc.; which (space) complexity classes are there with respect to implementation as a state system, and can efficient iterative representations be derived automatically?

All of these can be addressed by rephrasing trace functions, with their peculiar recursive structure, in a recursion scheme for total functions, namely *course-of-values* iteration, in category-theoretic presentation [31].

The formal scheme, in a nutshell, is as follows: Consider an endofunctor F whose initial algebra $(\mu F, \mathrm{in}_F : F\mu F \to \mu F)$ is a datatype of interest. The simplest total recursion scheme over F is iteration: For every F-algebra $(C, \varphi : FC \to C)$ there is a unique function $(\!|\varphi|\!) : \mu F \to C$ such that

$$(\!|\varphi|\!) = \varphi \circ F(\!|\varphi|\!) \circ \mathrm{in}_F^{-1}$$

The canonical example is the functor $N = 1 + \_$ with the initial algebra $(\mathbb{N}, [0, \mathit{succ}])$. Every N-algebra $(C, \varphi = [z, s])$ gives rise to a function $(\!|\varphi|\!) : \mathbb{N} \to C$ with $(\!|\varphi|\!)(n) = s^n(z)$, hence the name *iteration* for the scheme. Written as an explicitly self-referential definition, this gives

$$(\!|\varphi|\!)(n) = \begin{cases} z & n = 0 \\ s\big((\!|\varphi|\!)(n-1)\big) & n > 0 \end{cases}$$

That is, the function may depend recursively on its own value for the *immediate* predecessor(s) of the current argument value. The recursive tabular expression depicted in Fig. 2 is easily seen to be of this form.

The scheme of course-of-value iteration generalizes ordinary iteration to functions that depend on their own value for all *transitive* predecessors of the current argument value. Consider the composite functor $CF = C \times F(\_)$ and a final CF-coalgebra $(\nu CF, \mathrm{out}_{CF} : \nu CF \to CF\nu CF)$ as a (co)datatype. Then for every map $\psi : F\nu CF \to C$ there is a unique function $\{\!|\psi|\!\} : \mu F \to C$ whose precise definition and characterizing universal property are discussed in detail in [31]. The theory seems like overkill for many applications, where actually infinite sequences are irrelevant

Coming back to the previous example, we find that $\nu CN \cong C^+ \cup C^\omega$ (the set of non-empty finite *and* infinite sequences over C) and $N\nu CN \cong C^* \cup C^\omega$. Then we have specifically

$$\{\!|\psi|\!\}(n) = \psi\big(\langle \{\!|\psi|\!\}(n-1), \ldots \{\!|\psi|\!\}(0)\rangle\big) \tag{1}$$

The canonical example is $C = \mathbb{N}$ and

$$\psi(s) = \begin{cases} 0 & |s| = 0 \\ s_0 + 1 & |s| = 1 \\ s_0 + s_1 & |s| > 1 \end{cases}$$

where $s = \langle s_0, s_1, \ldots \rangle$, which generates the *Fibonacci* function $\{\!|\psi|\!\}$.

Let I, O be the record type of inputs/outputs of an interface, respectively, and write $IO = I \times O$. Defining a functor $T = IN = I \times (1 + \_)$ gives $\mu T \cong I^+$. Consequently $\nu OT \cong IO^+ \cup IO^\omega$ and $T\nu OT \cong I \times (IO^+ \cup IO^\omega)$. In words, $T\nu OT$ differs from $IO^* \cup IO^\omega$ only in the fact that the first O element is missing. A map

$$bag(T) \equiv \begin{array}{|c|c|c|}\hline T = \_ & & 0 \\ \hline r(T).op = & \mathsf{clr} & \\ & \mathsf{cnt} & n \\ & \mathsf{inc} & min(n+1, B) \\ & \mathsf{dec} & max(n-1, 0) \\ \hline \end{array}$$

$$\textbf{where} \qquad n \equiv bag\big(p(T) \blacktriangleright r(T).arg\big)$$

$$T \blacktriangleright x \equiv \begin{array}{|c|c|}\hline T = \_ & \_ \\ \hline r(T).op = \mathsf{clr} \vee r(T).arg = x & T \\ \hline r(T).op \neq \mathsf{clr} \wedge r(T).arg \neq x & p(T) \blacktriangleright x \\ \hline \end{array}$$

Fig. 4. Trace function specification of multiset data structure

of the form $\psi : T\nu OT \to O$ maps such a partial trace to the (missing) output, and hence defines a trace function $\{\!|\psi|\!\}$ with

$$\{\!|\psi|\!\}(\langle i_0, \ldots, i_n \rangle) = \psi\Big(i_0, \big\langle (i_1, \{\!|\psi|\!\}(\langle i_1, \ldots, i_n \rangle)), \ldots, (i_n, \{\!|\psi|\!\}(\langle i_n \rangle)) \big\rangle\Big) \qquad (2)$$

It can be seen, analogously to equation (1), that values for either infinite or illegal traces are irrelevant for the result.

The following simple but nontrivial example specifies the behavior of a mutable multiset component that can hold elements of some type E. Its interface supports four operations $op \in \{\mathsf{clr}, \mathsf{cnt}, \mathsf{inc}, \mathsf{dec}\}$ which completely remove all elements, count the multiplicity of a given element arg, and add or remove one instance of arg, respectively. Each operation returns the resulting multiplicity of the given element. Real-world constraints are added with requirements for robust behavior: No element may exceed a fixed multiplicity B, and removal of non-existent elements is ignored. These simple but natural constraints break naive attempts at algebraic specification: the algebraic structure may be correct, but is too complicated and irregular to be considered truly adequate.

Fig. 4 depicts a trace function specification of the multiset component. It is explicitly recursive via the auxiliary term n, but not in ordinary iteration form: the recursive argument is *some* predecessor of the current one, determined dynamically by the auxiliary function $T \blacktriangleright x$ which implements the (ordinarily iterative) search "subtrace of T up to the most recent event that affects the multiplicity of element x". This definition is easily transformed to a course-of-value generator map ψ, by replacing recursive calls with retrievals of recorded output values from the trace.

The functor for the recursion scheme of trace functions can be simplified from T to N, at the price of complicating the range type from O to $O^{I^+ \cup I^\omega}$, thus making trace function recursion a higher-order iteration [29]. The functor N is distinguished because for each of its course-of-value iterations, there is a

whole category of simulating deterministic transition systems with more or less elaborate state space [30], which form a semantic framework for both prototypic and production-quality implementations.

The selection of a particular implementation is a trade-off between ease of derivation and efficiency of execution, and cannot be automated straightforwardly. The initial implementation, which simply accumulates inputs and outputs, has unbounded space requirements, and may not be acceptable for any but the most prototypic uses. But practical hints can be gained from the analysis of access patterns to recursive predecessors: For instance, if there is a horizon such that each output depends only on the preceding k, then a ring buffer of size k is a fairly good state space for canonical implementation. In the Fibonacci example, we have $k = 2$ for the standard imperative implementation.

Other, more dynamic access patterns such as in the multiset example could possibly be classified according to their complexity and associated implementation techniques. For instance, the access pattern encoded in the operation $T \blacktriangleright x$ is of the very common variety "most recent event fulfilling φ", where φ here is a predicate depending with $x : E$ free. This suggests a map-like state with domain type E, which is already the implementation of choice for many applications.

5 Conclusion

The three levels of Parnas-style mathematical description of system behavior form a stack of methods, where each layer benefits from a (total) function viewpoint in a particular way. Predicate logic with partial value level and total truth level requires an implementation in terms of total (strongly terminating) functions because of the mutual dependencies between the levels. Tabular expression scale inhomogeneous function definitions up to very complex case distinctions. Their generic definition requires datatypes to contain explicit functions for checking and evaluation, and implicit functions for cell expressions with free variables. The generic parts can be fused with concrete contents to form specific table semantics by means of standard specialization techniques such as binding time analysis and partial evaluation. Finally, the trace function method employs tabular expressions with a particular recursion scheme over traces represented as sequences of hypothetical past events, for the description of component behavior without explicit reference to internal state. The precise form and meaning of this recursion scheme is given by categorical course-of-value iteration, for which rough but general implementation guidelines can be given, depending on the pattern of access to the past.

An interesting, informal but deep observation on all levels is that what appears as fundamental and directly understandable to the human reader of mathematical descriptions of behavior coincides largely with what can be expressed in constructive calculi of total functions. It seems plausible that this relation will be shown to apply even more widely by future experience in practical tool-making.

Acknowledgments. Thanks to David Parnas, Michael Hauhs, and several anonymous referees for valuable comments and suggestions.

References

1. Baber, R., Parnas, D., Vilkomir, S., Harrison, P., O'Connor, T.: Disciplined methods of software specification: A case study. In: Proc. ITCC 2005, vol. 2, pp. 428–437. IEEE Computer Society (2005)
2. Balat, V., Danvy, O.: Strong normalization by type-directed partial evaluation and run-time code generation. In: Leroy, X., Ohori, A. (eds.) TIC 1998. LNCS, vol. 1473, pp. 240–252. Springer, Heidelberg (1998)
3. Bartoussek, W., Parnas, D.L.: Using assertions about traces to write abstract specifications for software modules. In: Bracchi, G., Lockemann, P.C. (eds.) ECI 1978. LNCS, vol. 65, pp. 211–236. Springer, Heidelberg (1978)
4. Desharnais, J., Khédry, R., Mili, A.: Interpretation of tabular expressions using arrays of relations. In: Relational Methods for Computer Science Applications, pp. 3–13. Physica Verlag (2001)
5. Feng, X., Parnas, D.L., Tse, T.H.: Tabular expression-based testing strategies: A comparison. In: Proc. MUTATION 2007. IEEE Computer Society (2007)
6. Feng, X., Parnas, D.L., Tse, T.H.: Fault propagation in tabular expression-based specifications. In: Proc. COMPSAC 2008, pp. 180–183. IEEE Computer Society (2008)
7. Feng, X., Parnas, D.L., Tse, T.H., O'Callaghan, T.: A comparison of tabular expression-based testing strategies. IEEE Trans. Software Eng. 37(5), 616–634 (2011)
8. Furusawa, H., Kahl, W.: Table algebras: Algebraic structures for tabular notation, including nested headers. Programming science technical report, AIST (2004)
9. ISO/IEC: Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics (2002)
10. Janicki, R.: Towards a formal semantics of parnas tables. In: Proc. ICSE 1995, pp. 231–240. ACM (1995)
11. Janicki, R., Khédry, R.: On a formal semantics of tabular expressions. Sci. Comp. Progr. 39(2-3), 189–213 (2001)
12. Janicki, R., Parnas, D.L., Zucker, J.: Tabular representations in relational documents. In: Relational Methods in Computer Science, pp. 184–196. Springer (1997)
13. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Sci. Comput. Program. 75(11), 980–1000 (2010)
14. Kahl, W.: Compositional syntax and semantics of tables. SQRL Report 15, McMaster University (2003)
15. Lawford, M., Froebel, P., Moum, G.: Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Formal Methods in System Design (2001) (accepted for publication, 2004)
16. Lawford, M., McDougall, J., Froebel, P., Moum, G.: Practical application of functional and relational methods for the specification and verification of safety critical software. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 73–88. Springer, Heidelberg (2000)
17. Liu, Z., Parnas, D.L., Trancón y Widemann, B.: Documenting and verifying systems assembled from components. Frontiers of Computer Science in China 4(2), 151–161 (2010)
18. Pantelic, V., Jin, X., Lawford, M., Parnas, D.L.: Inspection of concurrent systems: Combining tables, theorem proving and model checking. In: Proc. SERP 2006, pp. 629–635 (2006)

19. Pantelic, V.: Combining tables, theorem proving and model checking. Msc thesis, McMaster University (2006)
20. Parnas, D.L., Wang, Y.: The trace assertion method of module interface specification. CIS Report 89-261, Queen's University (1989)
21. Parnas, D.L.: Tabular representation of relations. CLR Report 260, McMaster University (1992)
22. Parnas, D.L.: Predicate logic for software engineering. IEEE Trans. Software Eng. 19(9), 856–862 (1993)
23. Parnas, D.L.: Inspection of safety-critical software using program-function tables. In: IFIP Congress, vol. (3), pp. 270–277 (1994)
24. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Trans. Software Eng. 20(12), 948–976 (1994)
25. Parnas, D.L., Peters, D.K.: An easily extensible toolset for tabular mathematical expressions. In: Cleaveland, W.R. (ed.) TACAS/ETAPS 1999. LNCS, vol. 1579, pp. 345–359. Springer, Heidelberg (1999)
26. Peters, D., Lawford, M., Trancón y Widemann, B.: An ide for software development using tabular expressions. In: Proc. CASCON 2007, pp. 248–251. ACM (2007)
27. Sepehr, S.: Adding Nested Headers and a Proper Gtk-Based GUI to The Haskell Table Tools. Master's thesis, McMaster University (2010)
28. Trancón y Widemann, B., Parnas, D.L.: Tabular expressions and total functional programming. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 219–236. Springer, Heidelberg (2008)
29. Trancón y Widemann, B.: The recursion scheme of the trace function method. In: Proc. ENASE 2012, pp. 146–155 (2012)
30. Trancón y Widemann, B.: State-based simulation of linear course-of-value iteration. In: Proc. CMCS 2012. Tallinn University of Technology (2012); short contribution
31. Uustalu, T., Vene, V.: Primitive (co)recursion and course-of-value (co)iteration, categorically. Informatica, Lith. Acad. Sci. 10(1), 5–26 (1999)
32. Wang, Y., Parnas, D.L.: Simulating the behaviour of software modules by trace rewriting. In: Proc. ICSE 1993, pp. 14–23. IEEE Computer Society (1993)
33. Wang, Y., Parnas, D.L.: Trace rewriting systems. In: Rusinowitch, M., Remy, J.-L. (eds.) CTRS 1992. LNCS, vol. 656, pp. 343–356. Springer, Heidelberg (1993)

Using Rewriting to Synthesize Functional Languages to Digital Circuits

Christiaan Baaij and Jan Kuper

Department of Electrical Engineering, Mathematics, and Computer Science,
University of Twente, Postbus 217, 7500AE Enschede, The Netherlands
{c.p.r.baaij,j.kuper}@utwente.nl

Abstract. A straightforward synthesis from functional languages to digital circuits transforms variables to wires. The types of these variables determine the bit-width of the wires. Assigning a bit-width to polymorphic and function-type variables within this direct synthesis scheme is impossible. Using a term rewrite system, polymorphic and function-type binders can be completely eliminated from a circuit description, given only minor and reasonable restrictions on the input. The presented term rewrite system is used in the compiler for CλaSH: a polymorphic, higher-order, functional hardware description language.

1 Introduction

This paper describes the use of a Term Rewriting System (TRS) in the compiler for CλaSH [1, 2]. CλaSH is a polymorphic, higher-order, functional hardware description language. The purpose of the CλaSH compiler is to transform a description in a functional language to a format from which lithography machines can build an actual chip. The CλaSH compiler actually only provides a part of this transformation. It creates a low-level representation of the hardware, called a netlist; industry-standard tools are used for further processing. The translation from a (functional) description to a netlist is called *synthesis* in hardware literature. And the set of rules/transformations that together describe the conversion procedure from *description* to *netlist* is called a *synthesis scheme*.

The *synthesis scheme* used by the CλaSH compiler produces a specific *normal form* of the description. One aspect of this normal form is that the arguments and results of functions must have a representable type: a type whose values can be encoded by a fixed number of bits. This paper *only* describes the TRS that is used by the CλaSH compiler to eliminate, in a meaning-preserving manner, non-representable values from a functional description. Neither the exact normal form, the simplification TRS used to achieve this normal form, nor the complete synthesis scheme, are however presented. These aspects will be described in a future paper. This paper focuses on the TRS for non-representable value elimination, because it, among other things, transforms higher-order descriptions to first-order descriptions. Because first-order programs are susceptible to a greater range of analysis techniques [3], the described TRS has value in a broader context.

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 17–33, 2014.

The next subsection gives both a definition for netlists, and an introduction to synthesis schemes by describing a specific instance for a small functional language. The definition and introduction are both informal, but hopefully instil an intuition for the process of transforming a functional description to actual hardware.

1.1 Netlists and Synthesis

A netlist is a textual description of a digital circuit [4]. It lists the components that a circuit contains, and the connections between these components. The connection points of a component are called ports, or pins. The ports are annotated with the bit-width of the values that flow through them. A netlist can either be hierarchical or flattened. In a hierarchical netlist, sub-netlists are abstracted to appear as single components, of which multiple instances can be created. By instantiating all of these instances, a flattened netlist can be created.

A synthesis scheme defines the procedure that transforms a (functional) description to a netlist. Synthesis schemes defined for different languages, which nonetheless have common aspects, will be called a synthesis scheme *family*. The CλaSH compiler uses a synthesis scheme, called $\mathcal{T}_{C\lambda}$, that is an instance of the larger synthesis scheme family that will be referred to as $\mathcal{T}$. The following aspects are shared by all instances of $\mathcal{T}$:

- It is completely syntax-directed.
- It creates a hierarchical netlist.
- Function *definitions* are translated to components, where the arguments of the function become the input ports, and the result is connected to the output port.
- Function *application* is translated to an instance of the component that represents the corresponding function. The applied arguments are connected to the input ports of the component instance.

To demonstrate $\mathcal{T}$, a simple functional language, $\mathcal{L}$, is introduced in Fig. 1. $\mathcal{L}$ is a pure, simply-typed, first-order functional language. A program in $\mathcal{L}$ consists of set of function definitions, which always includes the *main* function. Expressions in $\mathcal{L}$ can be: variable references, primitives, or function application. Fig. 3 shows a small example program defined in the presented functional language.

The synthesis scheme for $\mathcal{L}$, called $\mathcal{T}_{\mathcal{L}}$, is defined by two transformations: $[\![\ ]\!]_p$ and $[\![\ ]\!]_e$, in which $[\![\ ]\!]_p$ is initially applied to the *main* function to create the hierarchical netlist. A graphical, informal, definition of the $[\![\ ]\!]_p$ and $[\![\ ]\!]_e$ transformations is depicted in Fig. 2. Again, the purpose of this subsection is to give an intuition for the synthesis process, not to give a formal account of $\mathcal{T}_{\mathcal{L}}$. $[\![\ ]\!]_p$ creates a component definition for a function f, where input ports correspond to the argument of f. $[\![\ ]\!]_p$ also creates an output port for the result of the expression e, which is connected to the outcome of the $[\![\ ]\!]_e$ transformation applied to e.

Fig. 2 shows that $[\![\ ]\!]_e$ transforms an argument reference x to a connection with an input port x. Function application of a function f is transformed to a component instance of f. $[\![\ ]\!]_p$ will be called for the definition of f in case there

$$
\begin{array}{lll}
p ::= f\ \overline{x} = e;\ p & & \text{Function definitions} \\
\quad \mid\ \emptyset & & \\
 & & \\
e ::= x & & \text{Argument reference} \\
\quad \mid\ \otimes\ \overline{e} & & \text{Primitive} \\
\quad \mid\ f\ \overline{e} & & \text{Function application}
\end{array}
$$

Fig. 1. $\mathcal{L}$: a simple functional language

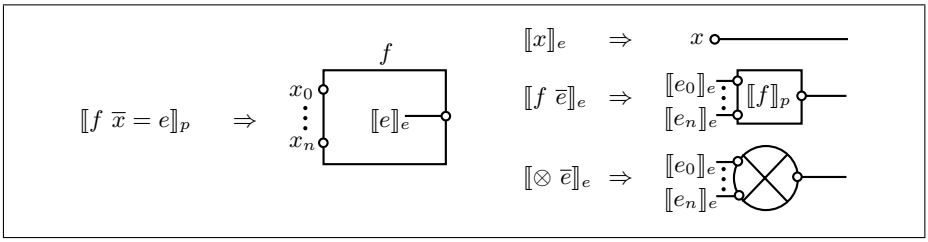


Fig. 2. $\mathcal{T}_{\mathcal{L}}$: A synthesis scheme for $\mathcal{L}$

$$
\begin{array}{l}
double\ x\ = x * x \\
main\ x\ y = (double\ x) + (double\ y)
\end{array}
$$

Fig. 3. Example program in $\mathcal{L}$

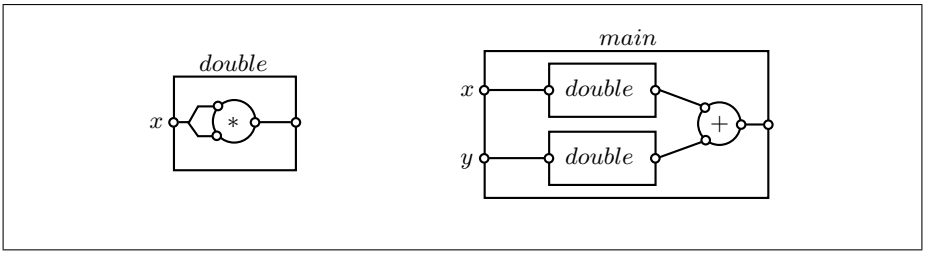


Fig. 4. Netlist of the example program in Fig. 3, created by $\mathcal{T}_{\mathcal{L}}$

is no existing component definition. Arguments to f are recursively transformed by $[\![\ ]\!]_e$, and the outcome of these transformations are connected to the input ports of the component f. The process for the transformations of primitives is analogous to that of functions, except that $[\![\ ]\!]_p$ will not be called for the definitions.

Applying the synthesis scheme $\mathcal{T}_{\mathcal{L}}$ to the example program given in Fig. 3 results in the (graphical representation of the) netlists depicted in Fig. 4. The netlist representation of *main* shows that synthesis schemes belonging to $\mathcal{T}$

exploit the *implicit parallelism* present in (pure) functional languages: as there are no dependencies between the operands of the addition, they are instantiated side-by-side. During the actual operation of the circuit, electricity flows through all parts simultaneously, and the instances of *double* will actually be operating in parallel.

Synthesis of CλaSH. CλaSH has a syntax and a semantics similar to the programming language Haskell [5] including some of language extensions of the Glasgow Haskell Compiler (GHC) [6]. These extensions include higher-rank polymorphism and existential datatypes. CλaSH and Haskell are so similar that every valid CλaSH description is also a valid Haskell program. Because CλaSH uses a synthesis transformation belonging to $\mathcal{T}$, called $\mathcal{T}_{C\lambda}$, the reverse relation does not hold. There are (many) Haskell programs that are not valid CλaSH descriptions. For example, recursive functions are not valid CλaSH descriptions: under $\mathcal{T}_{C\lambda}$, recursive application of a function *f* would lead to a recursive instance of the component *f*. Flattening the netlist would lead to infinitely many instantiations of the component *f*. Because such a netlist cannot be realized, the corresponding recursive function is currently deemed an invalid CλaSH description. Recursively defined (non-function) values are however allowed as they can be synthesized to feedback loops.

CλaSH uses an instance of the $\mathcal{T}$ family of synthesis schemes because it exploits the implicit parallelism of the functional descriptions, as shown earlier in Fig. 4. An important aspect of $\mathcal{T}$ is that the arguments and results of functions become the input and output ports of components. These ports are annotated with a bit-width so that it is known how many wires are needed to make connections between ports. Because CλaSH is a polymorphic, higher-order language, the arguments and results can however contain polymorphic or function-typed values. *It is generally impossible or impractical to represent such values by a fixed number of bits.*

In order to run $\mathcal{T}_{C\lambda}$, all values that cannot be represented by a fixed bit-width, will have to be eliminated from the functional description. The focus of this paper is a TRS that transforms the functional description in a meaning-preserving manner so that all non-representable values are eliminated. The presented TRS achieves its goal using both inlining and specialisation transformations [3].

The remainder of this paper is structured as follows: related work is described in the next section. CλaSH is desugared to a smaller Core language, and it is the Core language on which the TRS operates; Sect. 3 describes this Core language. Section 4 defines the (data)types which are considered non-representable, and the general process required for their meaning-preserving elimination. The rewrite rules of the TRS are described in Sect. 4.1. Properties of the TRS, including its non-termination, and the subsequent measures taken in the CλaSH compiler are discussed in Sect. 5. Conclusions are presented in Sect. 6.

2 Related Work

Functional Hardware Description Languages. SAFL [7] is a first-order hardware description language. As opposed to $\mathcal{T}_{C\lambda}$, which is used by CλaSH, SAFL uses a synthesis scheme that does not create a new component instance for every application of a function f. Instead, a component f is instantiated only once, and additional control and scheduling logic is inferred to safely approximate concurrent access.

BlueSpec SystemVerilog [8] is a hardware description language with a syntax similar to IEEE SystemVerilog standard. It has features also found in functional languages, such as higher-order functions and parametric polymorphism. The compilation from description to netlist is performed in two stages, which corresponds to the static and dynamic semantics of the language:

- A description is partially evaluated according to the static semantics, this includes the elimination/propagation of higher-order functions.
- The resulting description after partial evaluation is actually a set of rewrite rules. The second synthesis transformation instantiates all these rules in parallel, and adds scheduling logic in case there are conflicting preconditions [9].

So where the CλaSH compiler uses a TRS to eliminate non-representable values (such as those with a function type), the BlueSpec compiler uses a partial evaluator. There is however no account of the exact details of then partial evaluation mechanism in the Bluespec compiler, nor is there an exhaustive list of restrictions / requirements on the input programs.

Lava [10, 11] is a domain specific language *embedded in* Haskell. A hardware description in Lava is actually a Haskell program that uses combinators made available by the Lava library. These combinators wrap constructors of a graph datatype that represents a netlist. Synthesis of Lava descriptions is not performed in the traditional sense of transforming a description to a netlist. By *running* the Lava description, a Haskell program, the complete graph representing the netlist is simply calculated/constructed. Consequently, Lava gets the synthesis of higher-order, and recursive functions, *for free*: as long as the function *calculating* the graph terminates, a netlist can be created. Being an embedded language, Lava has disadvantages compared to a compiled language such as CλaSH:

- Because a program calculating the netlist graph cannot *observe* the (application of) individual functions, there can be no intuitive function-to-component mapping. As a result, only flattened netlists can be created.
- The rich set of *choice*-constructs in Haskell (also present in CλaSH), such as pattern-matching, cannot be reflected down to the netlists. Haskell's choice construct can be used to *guide the construction* of the netlist graph, but they cannot be *observed*. Consequently, a developer using Lava only has access to *choice*-functions offered by the Lava library.

Verity is a higher-order functional hardware description language with support for recursion (using a fix-point combinator) and mutable reference-cells.

Verity uses a *semantics*-directed synthesis scheme called *Geometry of Synthesis (GoS)* [12]. GoS assumes a linear type system, that restricts the use of identifiers to *exactly once*. That means that arguments with a function type need to be instantiated only once, an aspect GoS exploits during synthesis. Given a higher-order function f, which has a function-type argument g, the component corresponding to f is given extra input and output ports. The extra output ports correspond to the input ports for g, and the extra input ports correspond to the output ports of g. When f is applied to a function h, an instance of both the f and h component are created, and the components are correctly connected to each other. CλaSH does not have a linear type-system, meaning an identifier with a function type can be applied multiple times. Because of this, the CλaSH compiler cannot use the synthesis approach for function-typed arguments as promoted by GoS.

Higher-Order Removal Methods. Reynolds-style defunctionalisation [13] is a well-known method for generating an equivalent first-order program from a higher-order program. Reynolds' method creates datatypes for arguments with a function-type. Instead of applying a higher-order function to a value with a function-type, it is applied to a constructor for the newly introduced datatype. Application of the functional argument is replaced by the application of a mini-interpreter. Given the following higher-order program:

$$
\begin{aligned}
&\mathit{uncurry}\ f\ (a, b) = f\ a\ b \\
&\mathit{main}\ x = (\mathit{uncurry}\ (+)\ x) + (\mathit{uncurry}\ (-)\ x)
\end{aligned}
$$

Reynolds' method creates the following behaviourally equivalent first-order program:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Function} = \mathit{Plus} \mid \mathit{Sub} \\
&\mathit{apply}\ \mathit{Plus}\ a\ b = (+)\ a\ b \\
&\mathit{apply}\ \mathit{Sub}\ \ a\ b = (-)\ a\ b \\
&\mathit{uncurry}\ f\ (a, b) = \mathit{apply}\ f\ a\ b \\
&\mathit{main}\ x = (\mathit{uncurry}\ \mathit{Plus}\ x) + (\mathit{uncurry}\ \mathit{Min}\ x)
\end{aligned}
$$

Reynolds' method works on all programs, removes all functional arguments, and preserves sharing (a subject that will be discussed later). Although commonly defined and studied in the setting of the simply typed lambda calculus, there are also variants [14,15] of Reynolds' methods that are correct within a polymorphic type system. The disadvantage of Reynolds' method is the introduction of the mini-interpreter (which takes on the form of the *apply* function in the example). Due to the parallel nature of $\mathcal{T}_{C\lambda}$, this interpreter and all of its corresponding functionality will be instantiated at the use sites of the interpreter. For the above example it means that the interpreter will be instantiated twice; and so

will the included functionality: the adder and the subtracter. This method, in combination with $\mathcal{T}_{C\lambda}$, thus creates a lot of redundant hardware; it is this aspect that has precluded the use of Reynolds' method in the CλaSH compiler.

Many of the rewrite rules used by the TRS described in this paper can also be found in optimizing compilers for functional languages, such as GHC [16]. The rewrite rules presented by Peyton Jones and Santos [16] do however not guarantee a first-order normal form, which the TRS presented in this paper does (given certain restrictions on the input program).

Mitchell and Runciman [3] present a defunctionalisation method based on a TRS, which, like the TRS presented in this paper, also uses specialisation and inlining. The presented TRS can thus be seen as an extension to the work of Mitchell and Runciman:

- It provides transformations that additionally perform monomorphisation, which includes the specialisation of: higher-rank polymorphic arguments and existential datatypes.
- It can deal with recursive let-expressions.
- It works on a typed language, and uses this type information to determine when transformations should be applied.

3 Core Language

The syntactically rich CλaSH language is desugared to a smaller Core language, called Core_{HW} (Fig. 5), by the CλaSH compiler. It is a Church-style polymorphic lambda-calculus extended with primitives, algebraic datatypes in combination with case-decomposition, and recursive let-bindings. Case-decompositions are either exhaustive in the constructors of the matched datatype, or include the default pattern. Recursive let-bindings are needed to define values/expressions that are self-referencing and are used to describe feedback loops. Fig. 5 gives the language definition of Core_{HW}, and uses, just like the rest of this paper, the notation described in Fig. 6.

CλaSH supports existential datatypes, and this aspect of the language is reflected in Core_{HW}. A data constructor K, for an existential datatype $\mathbf{T}\ \overline{\alpha}$, is first abstracted over the universally quantified type-variables $\overline{\alpha}$, followed by the existentially quantified type-variables $\overline{\beta}$. The type variables $\overline{\beta}$ brought into scope by a pattern in a case-decomposition correspond to the existentially-quantified type-variables of the datatype.

3.1 Synthesis of Core_{HW} Using $\mathcal{T}_{C\lambda}$

The synthesis scheme $\mathcal{T}_{C\lambda}$ exploits all the implicit parallelism available in the Core_{HW} language. It does this by instantiating all expressions in a let-binding, and all alternatives of a case-decomposition, side-by-side (Fig. 7). $\mathcal{T}_{C\lambda}$ creates anchor points for let-binders so that variable references can be synthesized to connections to these anchor points.

Local variables: x, y, z
Global Variables: f, g
Type Variables: α, β

Data Constructor Types:
$K : \forall\overline{\alpha}.\forall\overline{\beta}.\overline{\tau} \rightarrow \mathbf{T}\ \overline{\alpha}$

Types:

$\tau, \sigma ::=$	
α	Variable reference
$\mid \tau \rightarrow \sigma$	Function Type
$\mid \mathbf{T}$	Datatype
$\mid \tau\ \sigma$	Type application
$\mid \forall\alpha.\sigma$	Polymorphic type

Expressions:

$e, u ::=$	
$x \mid f$	Variable reference
$\mid K \mid \otimes$	Data Constructor / Primitive
$\mid \Lambda\alpha.e \mid e\ \tau$	Type abstraction / application
$\mid \lambda x : \sigma.e \mid e\ u$	Term abstraction / application
$\mid$ **let** $\overline{x : \sigma = e}$ **in** u	Recursive let-binding
$\mid$ **case** e **of** $\overline{p \rightarrow u}$	Case-decomposition

Patterns:

$p ::= \_$	Default case
$\mid K\ \overline{\beta}\ \overline{x : \sigma}$	Match data constructor

Fig. 5. The Core_{HW} calculus

$$\mathbf{T}\ \overline{\sigma} \equiv \mathbf{T}\ \sigma_1\ ...\ \sigma_n \qquad e\ \overline{u} \equiv e\ u_1\ ...\ u_n$$
$$\overline{\tau} \rightarrow \sigma \equiv \tau_1 \rightarrow ... \rightarrow \tau_n \rightarrow \sigma \qquad \lambda\overline{x : \sigma}.e \equiv \lambda x_1 : \sigma_1.\ ...\ \lambda x_n : \sigma_n.e$$
$$\forall\overline{\alpha}.\sigma \equiv \forall\alpha_1.\ ...\ \forall\alpha_n.\sigma \qquad \overline{x : \sigma = e} \equiv \{x_1 : \sigma_1 = e_1,\ ...\ ,\ x_n : \sigma_n = e_n\}$$
$$\overline{p \rightarrow u} \equiv \{p_1 \rightarrow u_1,\ ...\ ,\ p_n \rightarrow u_n\}$$
$$K\ \overline{\beta}\ \overline{x : \sigma} \equiv K\ \beta_1\ ...\ \beta_n\ (x_1 : \sigma_1)\ ...\ (x_m : \sigma_m)$$

Fig. 6. Notation

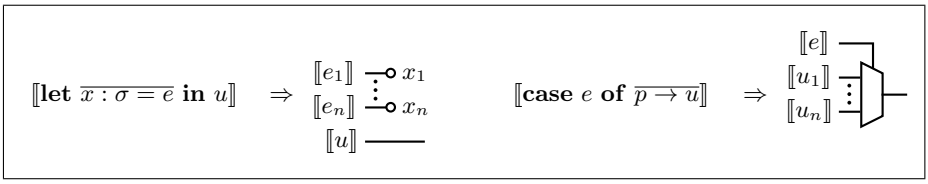


Fig. 7. Synthesis of **let** and **case**

Completely elaborating $\mathcal{T}_{C\lambda}$ falls outside of the scope of this paper. To at least convey an intuition for the synthesis performed by $\mathcal{T}_{C\lambda}$, an example program, and the corresponding netlist are shown in Fig. 8 and 9 respectively. The simultaneous presence of all alternatives in a case-decomposition, and all binders in a let-binding, has consequences for the *sharing* behaviour of expressions.

Sharing is normally defined as the *re-use of the result of a computation* by other expressions. In a digital circuit, sharing means connecting the output port of one component to the input ports of multiple other components. This aspect can be observed in Fig. 9, where the result of the multiplication is shared by the addition and the subtraction. Results that can be shared, instead of recomputed,

```
λx : Bool.λy : Int ∘ let
   z : Int = y ∗ y
in case x of
   True  → z + 1
   False → z + 1
```

Fig. 8. Example program using **let** and **case**

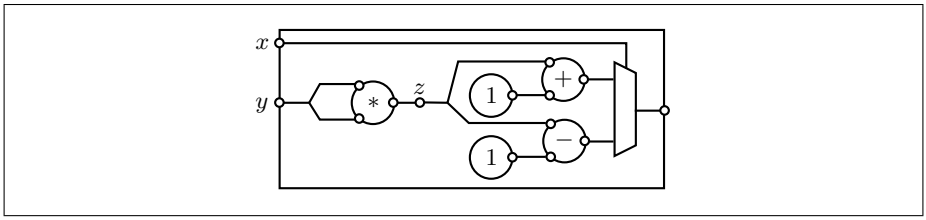


Fig. 9. Netlist of the example program in Fig. 8, created by $\mathcal{T}_{C\lambda}$

will reduce the total size of the circuit. The rewrite rules of the TRS should thus take the effects of sharing under $\mathcal{T}_{C\lambda}$ into account, as any loss in sharing increases the size of the circuit.

4 Eliminating Non-representable Types

$\mathcal{T}_{C\lambda}$ can only synthesize functional descriptions if arguments and results of expressions can be given a fixed bit-encoding. There are straightforward encodings for certain primitive datatypes, and certain algebraic datatypes. Datatypes with a fixed bit-encoding are called *representable*. Deriving a fixed bit-encoding for the following types is either not desired, or not possible:

- Function types
- (Higher-rank) polymorphic types
- Recursively defined datatypes.
- Datatypes that are composed of types that are not representable.

This section shows the TRS that eliminates non-representable values from the function hierarchy. It eliminates such values completely given that the input adheres to the following restriction:

- That both the arguments and the result of the *main* function, and the arguments and result of the used primitives, are representable.

The TRS uses a combination of inlining and specialisation, where specialisation takes on two forms:

- Specialisation of a function on one of its arguments.

– Elimination of a case-decomposition based on a known constructor.

The rewrite rules in this paper are presented using the format depicted in Fig. 10. In all of these rewrite rules, the expression above the horizontal bar is

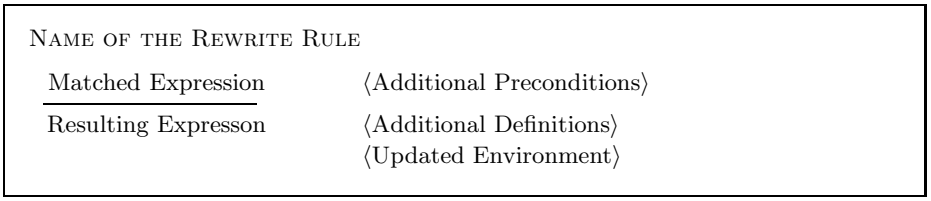


Fig. 10. Format for Rewrite Rules

the expression that has to be matched before performing the rewrite rule, and the expression below the horizontal bar is the result after applying the rewrite rule. Some rewrite rules have additional preconditions, and the rewrite is only applied when these preconditions hold. Other rewrite rules have additional definitions which they use in the resulting expressions. All rewrite rules always have access to the global environment, Γ, which holds all top-level binders. There are some rewrite rules that create new top-level binders, and therefore update the global environment.

The rewrite rules have access to the following functions:

FV e	Calculates the free variables; works for types and terms.
$e\ [x := u]$	A capture-free substitution of a variable reference x, by the expression or type u, in the expression e.
$\Gamma@f$	The expression belonging to a global binder f in the environment Γ.
NONREP τ	Determines if τ is a non-representable type.

Before the TRS starts, all variables are made unique, and all variable references are updated accordingly. Any new variables introduced by the rewrite rules will be unique by construction. Having hygienic expressions prevents accidental free-variable capture, and makes it easier to define meaning-preserving rewrite rules.

4.1 Rewrite Rules

The first three rewrite rules, τ-reduction, LetTyApp, and CaseTyApp, propagate type information downwards into an expression. By either removing type-variables, propagating type-information to a location for specialisation, or propagating type information to a primitive or constructor, these rewrite rules aid in the elimination of polymorphism.

τ-Reduction

$$\frac{(\Lambda\alpha.e)\ \tau}{e\ [\alpha := \tau]}$$

LetTyApp

$$\frac{(\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \tau}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ \tau)}$$

CaseTyApp

$$\frac{(\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ \tau}{\textbf{case}\ e\ \textbf{of}\ \overline{p \to (u\ \tau)}}$$

The next three rewrite rules, LamApp, LetApp, and CaseApp, propagate values, including non-representable ones, downwards into the expression. LamApp is preferred over β-reduction to preserve sharing. CaseApp creates a let-binding, instead of propagating the applied expression towards all alternatives, to preserve sharing. The next rewrite rule, LiftNonRep, removes let-binders introduced by LamApp and CaseApp in case they bind non-representable values.

LamApp

$$\frac{(\lambda x : \sigma.e)\ u}{\textbf{let}\ \{x = u\}\ \textbf{in}\ e}$$

LetApp

$$\frac{(\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ e_0}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ e_0)}$$

CaseApp

$$\frac{(\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ u_0}{\textbf{let}\ \{x = u_0\}\ \textbf{in}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p \to (u\ x)})}$$

LiftNonRep removes a let-binder, $x_i : \sigma_i = e_i$ (with a non-representable type σ_i), and substitutes references to x_i in the rest of the let-binding with an (application of a) variable reference to a *new*, *global*, binder: f. The new global binder, f, binds the original expression e_i which is abstracted over the free local (type) variables of e_i; all references to x_i are substituted with an (application of a) variable reference to f. The LiftNonRep rewrite rule uses the $\cup_\alpha$ operator to indicate that the global environment is only updated with the new binder, f, if an α-equivalent expression is not already present. In case an α-equivalent expression is present in the environment, the transformed expression will refer to that existing global binder instead.

LiftNonRep

$$\frac{\textbf{let } \{b_1; ...; b_{i-1}; x_i : \sigma_i = e_i; b_{i+1}; ...; b_n\} \textbf{ in } u}{(\textbf{let } \{b_1; ...; b_{i-1}; b_{i+1}; ...; b_n\} \textbf{ in } u)\ [x := f\ \overline{\alpha}\ \overline{z}]} \quad \text{Preconditions: NONREP}(\sigma_i)$$

$$\text{Definitions: } (\overline{\alpha}, \overline{y}) = \text{FV}(e_i);\quad \overline{z} = \overline{y} - \{x_i\}$$

$$\text{New Environment: } \Gamma \cup_\alpha \{(f, \Lambda\overline{\alpha}.\lambda\overline{z}.e_i[x_i := f\ \overline{\alpha}\ \overline{z}])\}$$

The previous rewrite rules either propagated non-representable values downwards into the expression, or lifted those values out of the expression. The next two sets of rewrite rules remove non-representable values by specialisation. The TypeSpec and NonRepSpec provide function argument specialisation. CaseLet, CaseCase, InlineNonRep, and CaseCon, together achieve specialisation by eliminating case-decompositions of known constructors (of non-representable datatypes).

The TypeSpec rewrite rule matches on a type application of a variable reference to a global binder, f. The application is replaced by a reference to the *new* global binder f'. The new binder f' is defined in terms of the body of f specialized on the type τ. NonRepSpec behaves similarly to TypeSpec for the application on non-representable arguments. The difference is that the expression of the new binder, f', is abstracted over the free variables of the specialised argument; the transformed expression also takes these free variables into account.

Both TypeSpec and NonRepSpec use the $\cup_\alpha$ operator to indicate that the global environment is only updated with a new binder if an α-equivalent specialization is not already present. In case an α-equivalent specialisation is present in the environment, the transformed expression will refer to that existing global binder instead.

TypeSpec

$$\frac{(f\ \overline{e})\ \tau}{f'\ \overline{e}} \quad \text{Preconditions: FV}(\tau) \equiv \emptyset$$

$$\text{New Environment: } \Gamma \cup_\alpha \{(f', \lambda\overline{x}.(\Gamma@f)\ \overline{x}\ \tau)\}$$

NonRepSpec

$$\frac{(f\ \overline{e})\ (u{:}\sigma)}{f'\ \overline{e}\ \overline{y}} \quad \text{Preconditions: NONREP}(\sigma) \wedge \text{FV}(\sigma) \equiv \emptyset \quad \text{Definitions: } \overline{y} = \text{FV}(u)$$

$$\text{New Environment: } \Gamma \cup_\alpha \{(f', \lambda\overline{x}.\lambda\overline{y}.(\Gamma@f)\ \overline{x}\ u)\}$$

The CaseLet is required in specialising expressions that have a non-representable datatype. Taking the let-binders out of the case-decomposition does not affect the sharing behaviour so can be applied blindly. There is no free variable capture in the alternatives because all variables are made unique before running the TRS.

The CaseCase rewrite rule is only applied if the subject of a case-decomposition has a non-representable datatype. CaseCase is not applied blindly because the alternatives in a case-decomposition are evaluated in parallel in the eventual circuit. So the CaseCase rewrite rule generates a larger number of alternatives than present in the matched expression. A larger number of alternatives results in a larger circuit. Even though CaseCase makes the circuit larger, the intention of CaseCase is to eventually expose the constructor of the non-representable datatype to CaseCon. CaseCon eliminates the case-decomposition, and subsequently amortizes the increase in circuit size induced by CaseCase.

InlineNonRep is only applied if the subject of a case expression is of a non-representable datatype, as inlining breaks down the component hierarchy. All bound variables in the inlined expression are regenerated, and variable references updated accordingly. This preserves the assumptions made by the other rewrite rules that all variables are unique.

The CaseCon rule comes in two variants:

- A case-decomposition with a constructor application as the subject, and a matching constructor pattern.
- A case-decomposition with a constructor application as the subject, with *no* matching constructor pattern.

CaseCon only creates a let-binding if the constructor in the subject exactly matches the constructor of an alternative. When the default pattern is matched, the case-decomposition is simply replaced by the expression belonging to the default alternative. Case-decompositions in Core_{HW} are exhaustive, either by enumerating all the constructors, or by including the default pattern. This means that when a constructor applications is the subject of a case-decomposition, CaseCon will always remove that case-decomposition.

CaseLet

$$\frac{\textbf{case}\ (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ e_1)\ \textbf{of}\ \overline{p \to u}}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\textbf{case}\ e_1\ \textbf{of}\ \overline{p \to u})}$$

CaseCase — Preconditions: NONREP(σ)

$$\frac{\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \{p_1 \to u_1;\ \ldots\ ;\ p_n \to u_n\} : \sigma)\ \textbf{of}\ \overline{p \to u}}{\textbf{case}\ e\ \textbf{of}\{p_1 \to \textbf{case}\ u_1\ \textbf{of}\ \overline{p \to u};\ \ldots\ ;\ p_n \to \textbf{case}\ u_n\ \textbf{of}\ \overline{p \to u}\}}$$

InlineNonRep — Preconditions: NONREP(σ)

$$\frac{\textbf{case}\ (f\ \overline{e}) : \sigma\ \textbf{of}\ \overline{p \to u}}{\textbf{case}\ ((\Gamma @ f)\ \overline{e})\ \textbf{of}\ \overline{p \to u}}$$

CaseCon
$$\frac{\textbf{case}\ K_i\ \overline{\tau_\forall}\ \overline{\tau_\exists}\ \overline{e}\ \textbf{of}\ \{...; K_i\ \overline{\beta}\ \overline{x:\sigma} \rightarrow u_i; ...\}}{(\textbf{let}\ \overline{x:\sigma = e}\ \textbf{in}\ u_i)\ [\overline{\beta} := \overline{\tau_\exists}]}$$

$$\frac{\textbf{case}\ K_i\ \overline{\tau_\forall}\ \overline{\tau_\exists}\ \overline{e}\ \textbf{of}\ \{\overline{p_{j\neq i} \rightarrow u}; \_ \rightarrow u_0\}}{u_0}$$

5 Discussion

5.1 Completeness

The first set of rewrite rules (τ-Reduction - LiftNonRep) propagates or removes non-representable values for those syntactical elements on which the specialisation rewrite rules do not match. The second set of rewrite rules (TypeSpec - CaseCon) remove the non-representable values through specialisation. All rewrite rules together hence remove all non-representable values from the function hierarchy (given the restrictions in Section 4).

The restrictions on primitives are needed because those cannot be specialized on their argument, nor can their definitions be inlined. The restriction that the result type of *main* cannot be a non-representable datatype, ensures that any expression calculating a non-representable datatype is either:

- the subject of a case-decomposition, which will be removed by the TRS,
- or, unreachable, and can be removed by dead-code elimination.

The CλaSH compiler applies the transformations in a specific order: a traversal with TypeSpec, followed by a traversal with NonRepSpec, are applied *after* all the other transformations have been applied exhaustively. Neither the correctness of the individual transformations, nor the guarantee of a first-order normal form, are dependent on this specific ordering of transformations. The argument-specialisation rewrite rules are applied last, so that the fewest number of new functions is introduced, and the original function hierarchy is preserved as much as possible. Because TypeSpec and NonRepSpec do not create expressions on which the other rewrite rules match, all rewrite rules have been applied exhaustively after the traversals with TypeSpec and NonRepSpec.

5.2 Termination

All rewrite rules are exhaustively applied during a (repeated) bottom-up traversal of an expression. InlineNonRep has to be applied using a bottom-up traversal, as a top-down traversal could lead to non-termination when inlining a recursive function. Using a bottom-up traversal for TypeSpec and NonRepSpec introduces the fewest number of lambda-abstraction in the specialized expressions.

There are several (combinations of) rewrite rules that induce non-termination of the unconstrained TRS. The CλaSH compiler has heuristics in place that

constrain the application of certain rewrite rules to ensure termination. When one of the termination measures is triggered, non-representable values remain present in the description. $\mathcal{T}_{C\lambda}$ will not be able to transform the description to a netlist when that happens.

It should be noted that these termination measures are only trigged on functions that contain (mutually) recursive function calls, or have a (mutually) recursive datatype as a result; functions which cannot be synthesized by $\mathcal{T}_{C\lambda}$ anyway. It can hence be said that the unconstrained TRS terminates for all *usefull* programs.

InlineNonRep Restriction. The precondition of InlineNonRep already limits the locations where inlining is applied, exhaustive application of this rewrite rule can however still induce non-termination when dealing with recursive functions. Additionally, although the TRS does not contain β-reduction as one of the rewrite rules, LamApp, LiftNonRep, InlineNonRep, and CaseCon together behave like β-reduction. This means that the typed version of $(\lambda x \rightarrow x\ x)\ (\lambda x \rightarrow x\ x)$:

$$\textbf{data}\ T = C\ (T \rightarrow Int)$$
$$(\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ C\ h \rightarrow h\ x)\ (C\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ C\ h \rightarrow h\ x))$$

induces non-termination. To prevent either situation from happening, a function f can only be inlined *once* at all use sites within a function g, for every pair of f and g.

NonRepSpec Restriction. Specialization performed by NonRepSpec can induce non-termination when a recursive function f has an argument that accumulates non-representable values. To ensure termination, a NonRepSpec is only applied to a function m number of times, where m can be set by the user of the CλaSH compiler.

6 Conclusions

The CλaSH compiler uses a synthesis scheme, $\mathcal{T}_{C\lambda}$, that produces a description that has specific normal from. One aspect of this normal form is that arguments and results of expressions have types for which a fixed bit-encoding exists. For $\mathcal{T}_{C\lambda}$, non-representable values are those values for which no fixed bit-encoding can be determined. The TRS presented in this paper removes all non-representable values from a function hierarchy while preserving the behaviour, given only minor restrictions on this function hierarchy. These restrictions are: that neither the *main* function nor the primitives of CλaSH, can have arguments or results of a non-representable type. These restrictions do however not limit the use polymorphism or higher-order functionality in the rest of the description. We, the authors of this paper, deem these restrictions reasonable for the application domain of CλaSH: creating digital circuits.

Although the CλaSH compiler cannot synthesize recursive function, this limitation is (slightly) amortized by a set of primitives that capture certain recursive patterns. Such functions / primitives include the *map* and *foldl* functions for fixed-length vectors. Using custom rules for these primitives, the CλaSH compiler can correctly synthesize the residual higher-order functionality that is left after normalization. A restriction that still holds for the use of these primitives is that they should *not* have a non-representable result.

Future Work. The complete $\mathcal{T}_{C\lambda}$ synthesis scheme, the normal-form of the Core_{HW} language which $\mathcal{T}_{C\lambda}$ produces, and the simplification TRS of the CλaSH compiler will be described in a future paper. To reduce the number of traversals needed to reach the first-order normal from, the strategy of the presented TRS and its implementation within the CλaSH compiler are also subject to further investigation. Aside from improving the TRS, we are also extending the CλaSH compiler to support the synthesis of recursive functions that can be unrolled at compile-time.

References

1. Baaij, C.P.R., Kooijman, M., Kuper, J., Boeijink, W.A., Gerards, M.E.T.: CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In: Proceedings of the 13th Conference on Digital System Design, USA, pp. 714–721. IEEE Computer Society (September 2010)
2. Gerards, M.E.T., Baaij, C.P.R., Kuper, J., Kooijman, M.: Higher-Order Abstraction in Hardware Descriptions with CλaSH. In: Proceedings of the 14th Conference on Digital System Design, USA, pp. 495–502. IEEE Computer Society (August 2011)
3. Mitchell, N., Runciman, C.: Losing Functions without Gaining Data. In: Proceedings of the Second Symposium on Haskell, pp. 13–24. ACM (September 2009)
4. Frankau, S.: Hardware Synthesis from a Stream-Processing Functional Language. PhD thesis, University of Cambridge (July 2004)
5. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries. Journal of Functional Programming, vol. 13 (2003)
6. The GHC Team: The GHC Compiler, version 7.6.1 (January 2013), `http://haskell.org/ghc`
7. Mycroft, A., Sharp, R.: A Statically Allocated Parallel Functional Language. In: Montanari, U., Rolim, J.D.P., Welzl, E. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 37–48. Springer, Heidelberg (2000)
8. Nikhil, R.S.: Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In: Coussy, P., Morawiec, A. (eds.) High-Level Synthesis - From Algorithm to Digital Circuit, pp. 129–146. Springer, Netherlands (2008)
9. Hoe, J.C., Arvind.: Hardware Synthesis from Term Rewriting Systems. In: Proceedings of the tenth International Conference on VLSI, pp. 595–619 (1999)
10. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware Design in Haskell. In: Proceedings of the Third International Conference on Functional Programming (ICFP), pp. 174–184. ACM (1998)

11. Gill, A.: Type-Safe Observable Sharing in Haskell. In: Proceedings of the Second Haskell Symposium, pp. 117–128. ACM (September 2009)
12. Ghica, D.R.: Geometry of Synthesis: A structured approach to VLSI design. In: Proceedings of the 34th Annual Symposium on Principles of Programming Languages (POPL), pp. 363–375. ACM (2007)
13. Reynolds, J.C.: Definitional Interpreters for Higher-Order Programming Languages. In: Proceedings of the 25th ACM National Conference, pp. 717–740. ACM Press (1972)
14. Pottier, F., Gauthier, N.: Polymorphic Typed Defunctionalization. In: Proceedings of the 31st Symposium on Principles of Programming Languages (POPL), pp. 89–98. ACM (2004)
15. Bell, J.M., Bellegarde, F., Hook, J.: Type-Driven Defunctionalization. In: Proceedings of the Second International Conference on Functional Programming (ICFP), pp. 25–37 (1997)
16. Peyton Jones, S., Santos, A.: Compilation by Transformation in the Glasgow Haskell Compiler. In: Functional Programming Workshops in Computing, pp. 184–204. Springer (1994)

Distributed Places

Kevin Tew[1], James Swaine[2], Matthew Flatt[3], Robert Bruce Findler[2], and Peter Dinda[2]

[1] Brigham Young University
tew@byu.edu
[2] Northwestern University
JamesSwaine2010@u.northwestern.edu, robby@eecs.northwestern.edu,
pdinda@northwestern.edu
[3] University of Utah
mflatt@cs.utah.edu

Abstract. Distributed Places bring new support for distributed, message-passing parallelism to Racket. This paper gives an overview of the programming model and how we had to modify our existing, runtime-system to support distributed places. We show that the freedom to design the programming model helped us to make the implementation tractable. The paper presents an evaluation of the design, examples of higher-level API's that can be built on top of distributed places, and performance results of standard parallel benchmarks.

1 Introduction

Dynamic, functional languages are important as rapid development platforms for solving everyday problems and completing tasks. As programmers embrace parallelism in dynamic programming languages, the need arises to extend multi-core parallelism to multi-node parallelism. Distributed places delivers multi-node parallelism to Racket by building on top of the existing places [18] infrastructure.

The right extensions to dynamic, functional languages enable the introduction of a hierarchy of parallel programming abstractions. Language extension allows these parallel programming abstractions to be concisely mapped to different hardware such as a shared memory node or a distributed memory machine. Distributed places are not an add-on library or a foreign function interface (FFI). Instead, Racket's places and distributed places are language extensions on which higher-level distributed programming frameworks can easily be expressed. An RPC mechanism, map reduce, MPI, and nested-data parallelism are all concisely and easily built on top of distributed places. These higher-level frameworks meld with the Racket language to create extended languages, which describe different types of distributed programming.

The distributed places API allows the user to spawn new execution contexts on remote machines. Distributed places reuse the communication channel API for intra-process parallelism to build a transparent distributed communication system over a underlying sockets layer. Racket's channels for parallel and distributed communication are first-class Racket events. These channels can be waited on concurrently with other Racket event objects such as file ports, sockets, threads, channels, etc. Together, Racket's intra-process and distributed parallelism constructs form a foundation capable of supporting higher-level parallel frameworks.

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 34–57, 2014.

2 Design

Programming with parallelism should avoid the typical interference problems of threads executing in a single address space. Instead, parallel executions contexts should execute in isolation. Communication between execution contexts should use message-passing instead of shared-memory in a common address space. This isolated, message-passing approach positions the programmer to think about the data-placement and communication needs of a parallel program to enable sustained scalability. Distributed places extend our existing implementation of isolated, message-passing parallelism which, until now, was limited to a single node. As a program moves from multi-core parallelism to multi-node parallelism latency increases and bandwidth decreases; data-placement and communication patterns become even more crucial.

Much of a distributed programming API is littered with system administration tasks that impede programmers from focusing on programming and solving problems. First, programmers have to authenticate and launch their programs on each node in the distributed system. Then they have to establish communication links between the nodes in the system, before they can begin working on the problem itself. The work of the distributed places framework is to provide support for handling the problems of program launch and communication link establishment.

Racket's distributed places API design is centered around machine nodes that do computation in places. The user/programmer configures a new distributed system using declarative syntax and callbacks. By specifying a hostname and port number, a programmer can launch a new place on a remote host. In the simplest distributed-places programs, hostnames and port numbers are hard-wired. When programmers need more control, distributed places permits complete programmatic configuration of node launch and communication link parameters.

Distributed Places adopt Erlang's failure model of fail fast. When a place dies or throws an unhandled exception, its execution ends. Parent places' are notified when a spawned place dies, but the user is responsible for recovery from errors.

The hello world example in figure 1 demonstrates the key components of a places program. Appearing first, the `hello-world` procedure is called to create hello-world places. The `main` module follows and contains the code to construct and communicate with a `hello-world` place.

Looking closer at the `main` module, the `hello-world` place is created using `dynamic-place`.

```
(dynamic-place module-path start-proc) → place?
  module-path : module-path?
  start-proc : symbol?
```

The `dynamic-place` procedure creates a place to run the procedure that is identified by *`module-path`* and *`start-proc`*. The result is a place descriptor value that represents the new parallel task; the place descriptor is returned immediately. The place descriptor is also a place channel to initiate communication between the new place and the creating place.

The module indicated by *`module-path`* must export a function with the name *`start-proc`*. The exported function must accept a single argument, which is a place

```
#lang racket/base
(require racket/place
         racket/place/distributed)

(provide hello-world)

(define (hello-world ch)
  (printf/f "hello-world received: ~a\n"
            (place-channel-get ch))
  (place-channel-put ch "Hello World\n")
  (printf/f "hello-world sent: Hello World\n"))

(module+ main
  (define p (dynamic-place
              (quote-module-path "..")
              'hello-world))

  (place-channel-put p "Hello")
  (printf/f "main received: ~a\n"
            (place-channel-get p))
  (place-wait p))
```

Fig. 1. Place's Hello World

channel that corresponds to the other end of communication for the place channel that is returned by `dynamic-place`.

The `(quote-module-path "..")` and `'hello-world` arguments on lines 15 and 16 of figure 1 specify the procedure address of the new place to be launched. In this example, the `(quote-module-path "..")` argument provides the module path to the parent module of `main`, where the `'hello-world` procedure is located.

Places communicate over place channels which allow structured data communication between places. Supported structured data includes booleans, numbers, characters, symbols, byte strings, Unicode strings, filesystem paths, pairs, lists, vectors, and "prefab" structures (i.e., structures that are transparent and whose types are universally named)[1].

```
(place-channel-put ch v) → void?
  ch : place-channel?
  v : place-message-allowed?
(place-channel-get ch) → place-message-allowed?
  ch : place-channel?
```

The `place-channel-put` function asynchronously sends a message `v` on channel `ch` and returns immediately. The `place-channel-get` function waits until a message is available from the place channel `ch`.

```
(place-wait p) → void?
  p : place?
```

Finally the `place-wait` procedure blocks until `p` terminates.

```
13 (module+ main
14   (define n (create-place-node
15               "host2"
16               #:listen-port 6344))
17   (define p (dynamic-place
18               #:at n
19               (quote-module-path "..")
20               'hello-world))
21   ...)
```

Fig. 2. Distributed Hello World

The distributed hello world example in figure 2 shows the two differences between a simple places program and a simple distributed places program. The `create-place-node` procedure uses `ssh` to start a new remote node on `host2` and assumes that ssh is

[1] `http://docs.racket-lang.org/guide/define-struct.html?q=prefab#(tech._prefab)`

configured correctly. Upon launch, the remote node listens on port 6344 for incoming connections. Once the remote node is launched, a TCP connection to port 6344 on the new node is established. The create-place-node returns a node descriptor object, n, which allows for administration of the remote node. The remote place is created using dynamic-place. The new #:at keyword argument specifies the node on which to launch the new place.

Remotely spawned places are private. Only the node that spawned the place can communicate with it through its descriptor object. Named places allow programmers to make a distributed place publicly accessible. Named places are labeled with a name when they are created.

```
(define p (dynamic-place
            #:at n
            #:named 'helloworld1
            (quote-module-path "..")
            'hello-world))
```

Any node can connect to a named place by specifying the destination node and name to connect to. In this example, node is a node descriptor object returned from create-place-node.

```
(connect-to-named-place node 'helloworld1)
```

3 Higher Level APIs

The distributed places implementation is a foundation that can support a variety of higher-level APIs and parallel processing frameworks such as Remote Procedure Calls (RPC), Message Passing Interface (MPI) [13], MapReduce [4], and Nested Data Parallelism [2]. All of these higher-level APIs and frameworks can be built on top of named places.

3.1 RPC via Named Places

Named places make a place's interface public at a well-known address: the host, port, and name of the place. They provide distributed places with a form of computation similar to the actor model [10]. Using named places and the define-named-remote-server form, programmers can build distributed places that act as remote procedure call (RPC) servers. The example in figure 3 demonstrates how to launch a remote Racket node instance, launch a remote procedure call (RPC) tuple server on the new remote node instance, and start a local event loop that interacts with the remote tuple server.

The create-place-node procedure in figure 3 connects to "host2" and starts a distributed place node there that listens on port 6344 for further instructions. The descriptor to the new distributed place node is assigned to the remote-node variable.

```
#lang racket/base
(require racket/place/distributed
         racket/class
         racket/place
         racket/runtime-path
         "tuple.rkt")
(define-runtime-path tuple-path "tuple.rkt")

(module+ main
  (define remote-node (create-place-node
                        "host2"
                        #:listen-port 6344))
  (define tuple-place
    (dynamic-place
       #:at remote-node
       #:named 'tuple-server
       tuple-path
       'make-tuple-server))

  (define c (connect-to-named-place
              remote-node
              'tuple-server))
  (define d (connect-to-named-place
              remote-node
              'tuple-server))
  (tuple-server-hello c)
  (tuple-server-hello d)
  (displayln
    (tuple-server-set c "user0" 100))
  (displayln
    (tuple-server-set d "user2" 200))
  (displayln (tuple-server-get c "user0"))
  (displayln (tuple-server-get d "user2"))
  (displayln (tuple-server-get d "user0"))
  (displayln (tuple-server-get c "user2")))
```

Fig. 3. Tuple RPC Example

Next, the dynamic-place procedure creates a new named place on the remote-node. The named place will be identified in the future by its name symbol 'tuple-server.

The code in figure 4 contains the use of the define-named-remote-server form, which defines a RPC server suitable for invocation by dynamic-place. The RPC tuple-server allows for named tuples to be stored into a server-side hash table and later retrieved. It also demonstrates one-way "cast" procedures, such as hello, that do not return a value to the remote caller.

```
1 #lang racket/base
2 (require racket/match
3          racket/place/define-remote-server)
4
5 (define-named-remote-server tuple-server
6
7   (define-state h (make-hash))
8   (define-rpc (set k v)
9     (hash-set! h k v)
10    v)
11  (define-rpc (get k)
12    (hash-ref h k #f))
13  (define-cast (hello)
14    (printf "Hello from define-cast\n")
15    (flush-output)))
```

Fig. 4. Tuple Server

For the purpose of explaining the tuple-server implementation, figure 5 shows the macro expansion of the RPC tuple server. Typical users of distributed places do not need to understand the expanded code to use the define-named-remote-server macro. The define-named-remote-server form, in figure 5, takes an identifier and a list of custom expressions as its arguments. A place function is created by prepending the make- prefix to the identifier tuple-server. The make-tuple-server identifier is the symbol given to the dynamic-place form in figure 3. The define-state custom form translates into a simple define form, which is closed over by the define-rpc forms.

The define-rpc form is expanded into two parts. The first part is the client stubs that call the RPC functions. The stubs can be seen at the top of figure 5. The client function name is formed by concatenating the define-named-remote-server identifier, tuple-server, with the RPC function name, set, to form tuple-server-set. The RPC client functions take a destination argument which is a remote-connection% descriptor followed by the RPC function's arguments. The RPC client function sends the

RPC function name, `set`, and the RPC arguments to the destination by calling an internal function `named-place-channel-put`. The RPC client then calls `named-place-channel-get` to wait for the RPC response.

The second part of the expansion part of `define-rpc` is the server implementation of the RPC call. The server is implemented by a match expression inside the `make-tuple-server` function. Messages to named places are placed as the first element of a list where the second element is the source or return channel to respond on. For example, in `(list (list 'set k v) src)` the inner list is the message while `src` is the place-channel to send the reply on. The match clause for `tuple-server-set` matches on messages beginning with the `'set` symbol. The server executes the RPC call with the communicated arguments and sends the result back to the RPC client. The `define-cast` form is similar to the `define-rpc` form except there is no reply message from the server to client.

The named place, shown in the tuple server example, follows an actor-like model by receiving messages, modifying state, and sending responses. Racket macros enables the easy construction of RPC functionality on top of named places.

3.2 Racket Message Passing Interface

RMPI is Racket's implementation of the basic MPI operations. A RMPI program begins with the invocation of the `rmpi-launch` procedure, which takes two arguments. The first is a hash from Racket keywords to values of default configuration options. The `rmpi-build-default-config` helper procedure takes a list of Racket keyword arguments and forms the hash of optional configuration values. The second argument is a list of configurations, one for each node in the distributed system. A configuration is made up of a hostname, a port, a unique name, a numerical RMPI process id, and an optional hash of additional configuration options. An example of `rmpi-launch` follows.

```
(rmpi-launch
  (rmpi-build-default-config
    #:racket-path "/tmp/mplt/bin/racket"
    #:distributed-launch-path
      (build-distributed-launch-path
        "/tmp/mplt/collects")
    #:rmpi-module "/tmp/mplt/kmeans.rkt"
    #:rmpi-func 'kmeans-place
    #:rmpi-args
      (list "/tmp/mplt/color100.bin"
            #t 100 9 10 0.0000001))

  (list (list "n1.example.com" 6340 'kmeans_0 0)
        (list "n2.example.com" 6340 'kmeans_1 1)
        (list "n3.example.com" 6340 'kmeans_2 2)
        (list "n4.example.com" 6340 'kmeans_3 3
              (rmpi-build-default-config
                #:racket-path "/bin/racket"))))
```

The `rmpi-launch` procedure spawns the remote nodes first and then spawns the remote places named with the unique name from the config structure. After the nodes

```
(module named-place-expanded racket/base
  (require racket/place racket/match)
  (define/provide
   (tuple-server-set dest k v)
    (named-place-channel-put
      dest
      (list 'set k v))
    (named-place-channel-get dest))
  (define/provide
   (tuple-server-get dest k)
    (named-place-channel-put
      dest
      (list 'get k))
    (named-place-channel-get dest))
  (define/provide
   (tuple-server-hello dest)
    (named-place-channel-put
       dest
       (list 'hello)))
  (define/provide
   (make-tuple-server ch)
    (let ()
      (define h (make-hash))
      (let loop ()
        (define msg (place-channel-get ch))
        (match
         msg
         ((list (list 'set k v) src)
          (define result (let ()
                           (hash-set! h k v)
                           v))
          (place-channel-put src result)
          (loop))
         ((list (list 'get k) src)
          (define result
            (let ()
              (hash-ref h k #f)))
          (place-channel-put src result)
          (loop))
         ((list (list 'hello) src)
          (define result
            (let ()
              (printf
                "Hello from define-cast\n")
              (flush-output)))
          (loop)))
        loop)))
  (void))
```

Fig. 5. Macro Expansion of Tuple Server

and places are spawned, `rmpi-launch` sends each spawned place its RMPI process id, the config information for establishing connections to the other RMPI processes, and the initial arguments for the RMPI program. The last function of `rmpi-launch` is to rendezvous with RMPI process 0 when it calls `rmpi-finish` at the end of the RMPI program.

The `rmpi-init` procedure is the first call that should occur inside the `#:rmpi-func` place procedure. The `rmpi-init` procedure takes one argument `ch`, which is the initial `place-channel` passed to the `#:rmpi-func` procedure. The `rmpi-init` procedure communicates with `rmpi-launch` over this channel to receive its RMPI process id and the initial arguments for the RMPI program.

```
(define (kmeans-place ch)
  (define-values (comm args tc) rmpi-init ch)
  ;;; kmeans rmpi computation ...
  (rmpi-finish comm tc))
```

The `rmpi-init` procedure has three return values: an opaque communication structure which is passed to other RMPI calls, the list of initial arguments to the RMPI program, and a typed channel wrapper for the initial place-channel it was given. The typed channel wrapper allows for the out of order reception of messages. Messages are lists and their type is the first item of the list, which must be a racket symbol. A typed channel returns the first message received on the wrapped channel that has the type requested. Messages of other types that are received are queued for later requests.

The `rmpi-comm` structure, returned by `rmpi-init`, is the communicator descriptor used by all other RMPI procedures. The RMPI informational functions `rmpi-id` and `rmpi-cnt` return the current RMPI process id and the total count of RMPI processes, respectively.

```
> (rmpi-id comm)
3
```

```
> (rmpi-cnt comm)
8
```

The `rmpi-send` and `rmpi-recv` procedures provide point-to-point communication between two RMPI processes.

```
> (rmpi-send comm dest-id '(msg-type1 "Hi"))
```

```
> (rmpi-recv comm src-id)
'(msg-type1 "Hi")
```

With the `rmpi-comm` structure, the programmer can also use any of the RMPI collective procedures: `rmpi-broadcast`, `rmpi-reduce`, `rmpi-allreduce`, or `rmpi-barrier` to communicate values between the nodes in the RMPI system.

The `(rmpi-broadcast comm 1 (list 'a 12 "foo"))` expression broadcasts the list `(list 'a 12 "foo")` from RMPI process `1` to all the other RMPI processes in the `comm` communication group. Processes receiving the broadcast execute `(rmpi-broadcast comm 1)` without specifying the value to send. The `(rmpi-reduce comm 3 + 3.45)` expression does the opposite of broadcast by reducing the local value `3.45` and all the other procesess local values to RMPI process `3` using the `+` procedure to do the reduction. The `rmpi-allreduce` expression is similar to `rmpi-reduce` except that the final reduced value is broadcasted to all processes in the system after the reduction is complete. Synchronization among all the RMPI processes occurs through the use of the `(rmpi-barrier comm)` expression, which is implemented internally using a simple reduction followed by a broadcast.

Distributed places are simply computation resources connected by socket communications. This simple design matches MPI's model and makes RMPI's implementation very natural. The RMPI layer demonstrates how distributed places can provide the foundations of other distributed programming frameworks such as MPI.

3.3 Map Reduce

Our MapReduce implementation is patterned after the Hadoop [1] framework. Key value pairs are the core data structures that pass through the map and reduce stages of the computation. In the following example, the number of word occurrences is counted across a list of text files. The files have been preprocessed so that there is only one word per line.

Figure 6 shows the different actors in the MapReduce paradigm. The program node P creates the MapReduce workers group. When a `map-reduce` call is made, the program node serves as the controller of the worker group. It dispatches mapper tasks to each node and waits for them to respond as finished with the mapping task. Once a node has finished its mapping task, it runs the reduce operation on its local data. Given two nodes in the reduced state, one node can reduce to the other; freeing one node to return to the worker pool for allocation to future tasks. Once all the nodes have reduced to a single node, the `map-reduce` call returns the final list of reduced key values.

The first step in using distributed place's MapReduce implementation is to create a list of worker nodes. This is done by calling the `make-map-reduce-workers` procedure with a list of hostnames and ports to launch nodes at.

```
(define config (list (list "host2" 6430)
                     (list "host3" 6430)))
(define workers (make-map-reduce-workers config))
```

Once a list of worker nodes has been spawned, the programmer can call `map-reduce` supplying the list of worker nodes, the config list, the procedure address of the mapper, the procedure address of the reducer, and a procedure address of an optional result output procedure. Procedure addresses are lists consisting of the quoted-module-path and the symbol name of the procedure being addressed.

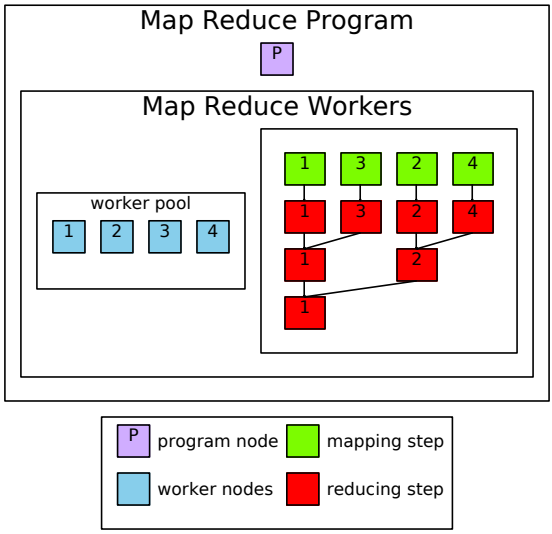


Fig. 6. MapReduce Program

```
(map-reduce
  workers
  config
  tasks
  (list (quote-module-path "..") 'mapper)
  (list (quote-module-path "..") 'reducer)
  #:outputer (list (quote-module-path "..")
                   'outputer))
```

Tasks can be any list of key value pairs. In this example the keys are the task numbers and the values are the input files the mappers should process.

```
(define tasks (list (list (cons 0 "/tmp/w0"))
                    (list (cons 1 "/tmp/w1"))
                    ...))
```

The mapper procedure takes a list of key value pairs as its argument and returns the result of the map operation as a new list of key value pairs. The input to the mapper, in this example, is a list of a single pair containing the task number and the text file to process, `(list (cons 1 "w0.txt"))`. The output of the mapper is a list of each word in the file paired with 1, its initial count. Repeated words in the text are repeated in the mappers output list. Reduction happens in the next step.

```
;;(->
;;  (listof (cons any any))
;;  (listof (cons any any)))
(define/provide (mapper kvs)
  (for/first ([kv kvs])
    (match kv
      [(cons k v)
       (with-input-from-file
         v
         (lambda ()
           (let loop ([result null])
             (define l (read-line))
             (if (eof-object? l)
                 result
                 (loop (cons (cons l 1)
                             result))))))])))
```

After a task has been mapped, the MapReduce framework sorts the output key value pairs by key. The framework also coalesces pairs of key values with the same key into a single pair of the key and the list of values. As an example, the framework transforms the output of the mapper `'(("house" 1) ("car" 1) ("house" 1))` into `'(("car" (1)) ("house" (1 1)))`

The reducer procedure takes, as input, this list of pairs, where each pair consists of a key and a list of values. For each key, the reducer reduces the list of values to a list of a single value. In the word count example, an input pair, `(cons "house" '(1 1 1 1))` will be transformed to `(cons "house" '(4))` by the reduction step.

```
;;(->
;; (listof (cons any (listof any)))
;; (listof (cons any (listof any))))
(define/provide (reducer kvs)
  (for/list ([kv kvs])
    (match kv
      [(cons k v)
       (cons k (list (for/fold ([sum 0])
                               ([x v])
                       (+ sum x))))])))
```

Once each mapped task has been reduced, the outputs of the reduce steps are further reduced until a single list of word counts remains. Finally, an optional output procedure is called which prints out a list of words and their occurrence count and returns the total count of all words.

```
(define/provide (outputer kvs)
  (displayln
    (for/fold ([sum 0]) ([kv kvs])
      (printf "~a - ~a\n" (car kv) (cadr kv))
      (+ sum (cadr kv)))))
```

3.4 Nested Data Parallelism

The last parallel processing paradigm implemented on top of distributed places is nested data parallelism [9]. In this paradigm recursive procedure calls create subproblems that can be parallelized. An implementation of parallel quicksort demonstrates nested data parallelism built on top of distributed places.

The distributed places, nested data parallelism API – `ndp-get-node`, `ndp-sendwork`, `ndp-get-result`, and `ndp-return-node` – is built on top of the RMPI layer. The main program node, depicted as `P` in figure 7, creates the `ndp-group`. The `ndp-group` consists of a coordinating node, `0`, and a pool of worker nodes `1, 2, 3, 4`. The coordinating node receives a sort request from `ndp-sort` and forwards the request to the first available worker node, node `1`. Node `1` divides the input list in half and requests a new node from the coordinator to process the second half of the input. The yellow bars on the right side of figure 7 show the progression as the sort input is subdivided and new nodes are requested from the coordinator node. Once the sort is complete, the result is returned to the coordinator node, which returns the result to the calling program `P`.

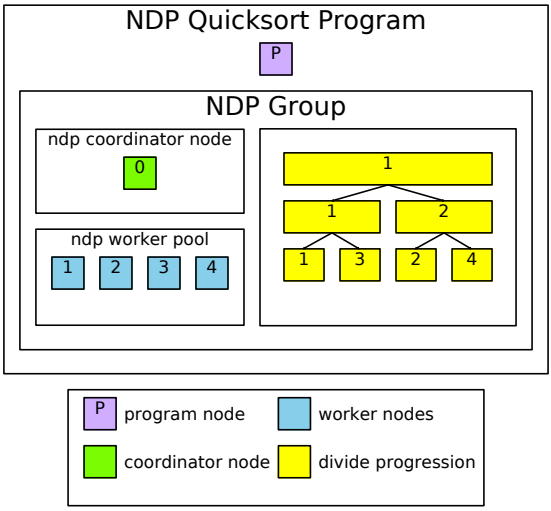


Fig. 7. NDP Program

Like the previous two examples, the nested data parallel quicksort example begins by spawning a group of worker processes.

```
(define config
  (list (list "host2" 6340)
        (list "host3" 6340)
        (list "host4" 6340)
        (list "host5" 6340)
        (list "host6" 6340)))

(define ndp-group (make-ndp-group config))
```

Next the sort is performed by calling `ndp-qsort`.

```
(displayln (ndp-qsort (list 9 1 2 8 3 7 4 6 5 10)
                      ndp-config))
```

The `ndp-qsort` procedure is a stub that sends the procedure address for the `ndp-parallel-qsort` procedure and the list to sort to the `ndp-group`. The work of the parallel sort occurs in the `ndp-parallel-sort` procedure in figure 8. First, the `partit` procedure picks a pivot and partitions the input list into three segments: less than the pivot, equal to the pivot, and greater than the pivot. If a worker node can be obtained from the `ndp-group` by calling `ndp-get-node`, the `gt` partition is sent to the newly obtained worker node to be recursively sorted. If all the worker nodes are taken, the `gt` partition is sorted locally using the `ndp-serial-qsort` procedure. Once the `lt` partition is sorted recursively on the current node, the `gt-part` is checked to see if it was computed locally or dispatched to a remote node. If the part was dispatched to a remote node, its results are retrieved from the remote node by calling `ndp-get-result`. After the results are obtained, the remote node `node` can be returned to the `ndp-group` for later use. Finally, the sorted parts are appended to form the final sorted list result.

4 Implementation

A key part of the distributed place implementation is that distributed places is a layer over places, and parts of the places layer are exposed through the distributed places layer. In particular, each node, in figure 9, begins life with one initial place, the message router. The message router listens on a TCP port for incoming connections from other nodes in the distributed system. The message router serves two primary purposes: it multiplexes place messages and events on TCP connections between nodes and it services remote spawn requests for new places.

There are a variety of distributed places commands which spawn remote nodes and places. These command procedures return descriptor objects for the nodes and places they create. The descriptor objects allow commands and messages to be communicated to the remote controlled objects. In Figure 10, when node A spawns a new node B, A is given a `remote-node%` object with which to control B. Consequently, B is created with a `node%` object that is connected to A's `remote-node%` descriptor via a TCP socket connection. B's `node%` object is the message router for the new node B. A can then use its `remote-node%` descriptor to spawn a new place on node B. Upon successful spawning of the new place on B, A is returned a `remote-place%` descriptor object. On

```
(define (ndp-parallel-qsort l ndp-group)
  (cond
    [(< (length l) 2) l]
    [else
      (define-values (lt eq gt) (partit l))

      ;; spawn off gt partition
      (define gt-ref
         (define node (ndp-get-node ndp-group))
         (cond
           [node
             (cons #t (ndp-send-work
                        ndp-group
                        node
                        (list
                          (quote-module-path)
                          'ndp-parallel-qsort)
                        gt))]
           [else
             (cons #f (ndp-serial-qsort gt))]))

      ;; compute lt partition locally
      (define lt-part
        (ndp-parallel-qsort lt ndp-group))

      ;; retrieve remote results
      (define gt-part
        (match gt-ref
          [(cons #t node-id)
            (begin0
              (ndp-get-result ndp-group node-id)
              (ndp-return-node
                ndp-group
                node-id))]
          [(cons #f part) part]))

      (append lt-part eq gt-part)]))
```

Fig. 8. NDP Parallel Sort

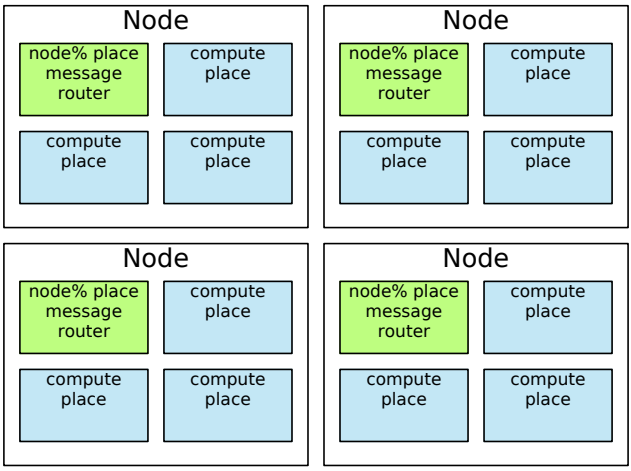


Fig. 9. Distributed Places Nodes

node B, a `place%` object representing the newly spawned place is attached to B's `node%` message-router. The `remote-connection%` descriptor object represents a connection to a named place. At the remote node, B, a `connection%` object intermediates between the `remote-connection%` and its destination named-place.

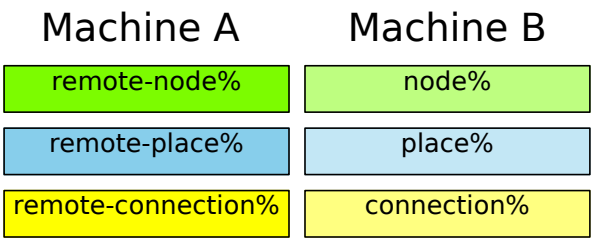


Fig. 10. Descriptor (Controller) - Controlled Pairs

To communicate with remote nodes, a place message must be serializable. As a message-passing implementation, places send a copy of the original message when communicating with other places. Thus, the content of a place message is inherently serializable and transportable between nodes of a distributed system.

To make place channels distributed, `place-socket-bridge%` proxies need to be created under the hood. The `place-socket-bridge%`s listen on local place channels and forward place messages over TCP sockets to remote place channels. Each node in a Racket distributed system must either explicitly pump distributed messages by registering each proxy with `sync` or bulk register the proxies, via the `remote-node%` descriptor, with a message router which can handle the pumping in a background thread.

Figure 11 shows the layout of the internal objects in a simple three node distributed system. The node at the top of the figure is the original node spawned by the user. Early in the instantiation of the top node, two additional nodes are spawned, node 1 and node 2. Then two places are spawned on each of node 1 and node 2. The instantiation code of the top node ends with a call to the `message-router` form. The `message-router` contains the `remote-node%` instances and the `after-seconds` and `every-seconds` event responders. Event responders execute when specific events occur, such as a timer event, or when messages arrive from remote nodes. The message router de-multiplexes events and place messages from remote nodes and dispatches them to the correct event responder.

Finally, function overloading is used to allow `place-` functions, such as `place-channel-get`, `place-channel-put`, and `place-wait`, to operate transparently on both place and distributed place instances. To accomplish this, distributed place descriptor objects are tagged as implementing the `place<%>` interface using a Racket structure property. Then `place-` functions dynamically dispatch to the distributed place version of the function for distributed place instances or execute the original function body for place instances.

5 Distributed Places Performance

Two of the NAS Parallel Benchmarks, IS and CG, are used to test the performance of the Racket distributed places implementation. The Fortran/C MPI version of the benchmarks were ported to Racket's distributed places. Performance testing occurred on 8 quad-core Intel i7 920 machines. Each machine was equipped with at least 4 gigabytes of memory and a 1 gigabit Ethernet connection.

Performance numbers are reported for both Racket and Fortran/C versions of the benchmarks in figure 12. Racket's computational times scaled appropriately as additional nodes were added to the distributed system. Computational times are broken out and graphed in isolation to make computational scaling easier to see.

Racket communication times were larger than expected. There are several factors, stacked on top of one another, that explain the large communication numbers. First, five copies of the message occur during transit from source to destination. In a typical operation, a segment of a large flonum vector needs to be copied to a destination distributed place. The segment is copied (1) out of the large flonum vector into a new flonum vector message. The message vector's length is the length of the segment to be sent. Next, the newly constructed vector message is copied (2) over a place channel from the computational place to the main thread which serializes and copies (3) the message out a TCP socket to its destination. When the message arrives at its destination node, the message is deserialized and copied (4) a fourth time over a place channel to the destination

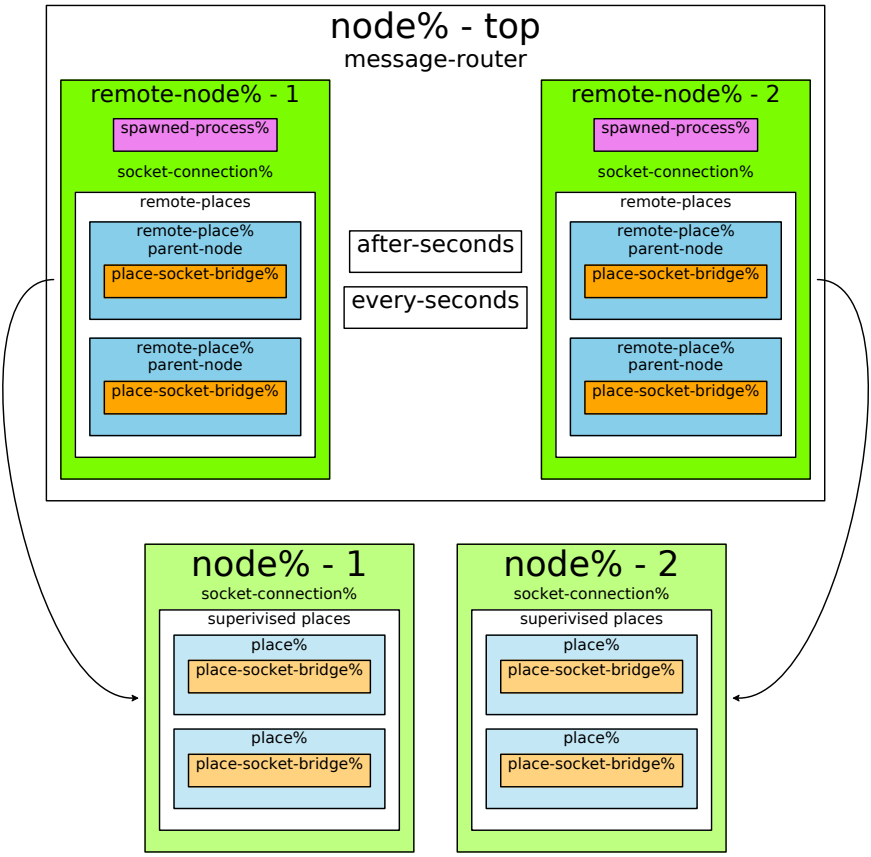


Fig. 11. Three Node Distributed System

computational place. Finally, the elements of the message vector are copied (5) into the mutable destination vector.

Racket's MPI implementation, RMPI, is not as sophisticated as the standard MPICH [14] implementation. MPICH has nonblocking sends and receives that allow messages to flow both directions simultaneously. Both the NAS Parallel Benchmarks used, IS and CG, use non-blocking MPI receives. RMPI on the other hand, always follows the typical protocol design of sending data in one direction and then receiving data from the opposite direction.

The largest contributor to Racket's excessive communication times is the serialization costs of the Racket primitive `write`. On Linux, serialization times are two orders of magnitude larger than the time to write raw buffers. One solution would be to replace distributed place's communication subsystem with FFI calls to an external MPI library. This solution would bypass the expensive `write` calls currently used in distributed places. Another viable solution would be to recognize messages that are vectors of flonums and use a restricted-form of write that could write flonum vectors as efficiently as raw buffers. Finally, it should be noted that using Racket's `write` is advantageous in cases where the message to be sent is a complex object graph instead of a simple raw buffer.

6 Related Work

Erlang [16] Erlang's distributed capabilities are built upon its process concurrency model. Remote Erlang nodes are identified by `name@host` identifiers. New Erlang processes can be started using the `slave:start` procedure or at the command line. Erlang uses a feature called links to implement fault notification. Two processes establish a link between themselves. Links are bidirectional; if either process fails the other process dies also. Erlang also provides monitors which are unidirectional notifications of a process exiting. Distributed Places and Erlang share a lot of similar features. While Erlang's distributed processes are an extension of its process concurrency model, Distributed Places are an extension of Racket's places parallelism strategy. Erlang provides a distributed message passing capability that integrates transparently with its inter-process message passing capability. The Disco project implements map reduce on top of an Erlang core. User level Disco programs, however, are written in Python, not Erlang. In contrast, the implementation and user code of distributed places' map reduce are both expressed as Racket code. Erlang has a good foundation for building higher-level distributed computing frameworks, but instead Erlang programmers seem to build customized distributed solutions for each application.

MapReduce [4] is a specialized functional programming model, where tasks are automatically parallelized and distributed across a large cluster of commodity machines. MapReduce programmers supply a set of input files, a map function and a reduce function. The map function transforms input key/value pairs into a set of intermediate key/value pairs. The reduce function merges all intermediate values with the same key. The framework does all the rest of the work. Google's MapReduce implementation handles partitioning of the input data, scheduling tasks across distributed computers, restarting tasks due to node failure, and transporting intermediate results between com-

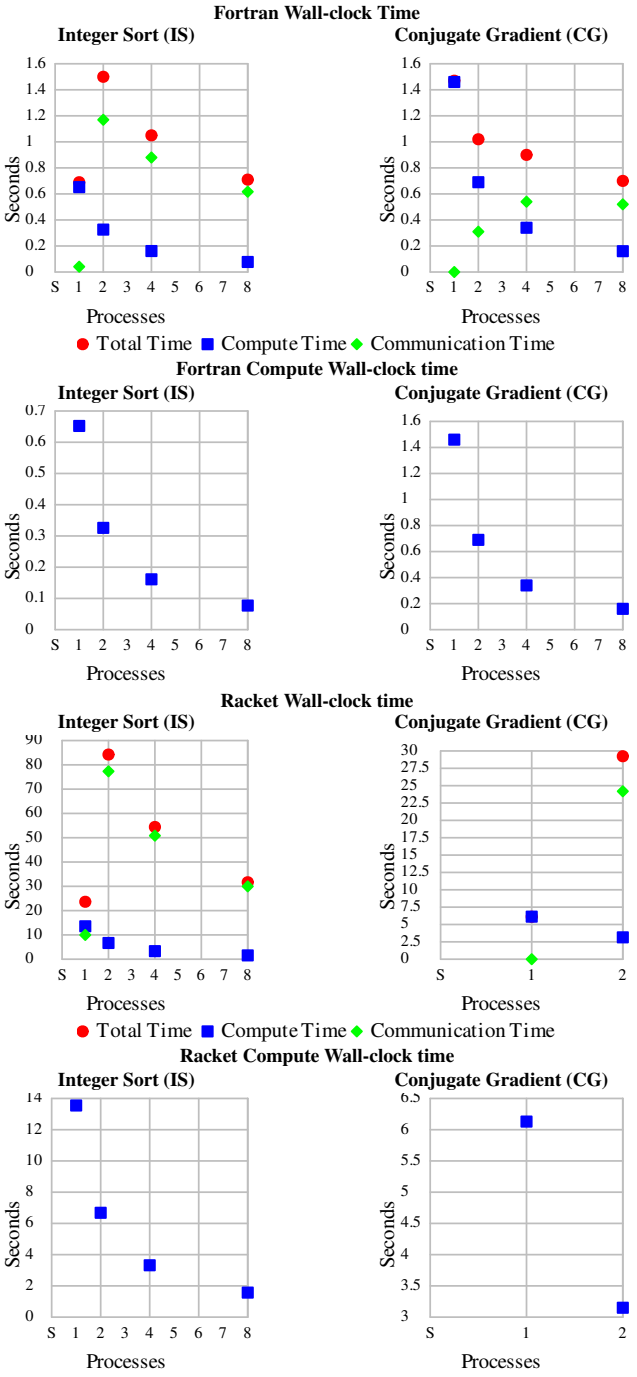


Fig. 12. IS, CG, and MG class A results

pute nodes. The MapReduce model can be applied to problems such as word occurance counting, distributed grep, inverted index creation, and distributed sort.

Termite [8] Termite is a distributed concurrent scheme built on top of Gambit-C Scheme. Direct mutation of variables and data structures is forbidden in Termite. Instead mutation is simulated using messages and suspended, lightweight processes. Lookup in Termite's global environment is a node relative operation and resolves to the value bound to the global variable on the current node. Termite supports process migration via serializable closures and continuations. Termite follows Erlang's style of failing hard and fast. Where Erlang has bidirectional links, Termite has directional links that communicate process failure from one process to another. Failure detection only occurs in one direction from the process being monitored to the monitoring process. Termite also has supervisors which like supervisors in Erlang, restart child processes which have failed. Distributed Places could benefit from Termites superior serialization support, where nearly all Termite VM objects are serializable.

Akka [19] is a concurrency and distributed processing framework for Scala and Java. Like Erlang, Akka is patterned after the Actor model. Akka supports Erlang like supervisors and monitors for failure and exit detection. Like Erlang, Akka leaves the creation of higher-level distributed frameworks to custom application developers.

Kali [3] is a distributed version of Scheme 48 that efficiently communicates procedures and continuations from one compute node to another. Kali's implementation lazily faults continuation frames across the network as they are needed. Kali's proxies are really just address space relative variables. Proxies are identified by a globally unique id. Sending a proxy involves sending only its globally unique id. Retrieving a proxies value returns the value for the current address space. Kali allow for retrieval of the proxy's source node and spawning of new computations at the proxy's source.

Distributed Functional Programming in Scheme (DFPS) [17] uses futures semantics to build a distributed programming platform. DFPS employs the Web Server collection's `serial-lambda` form to serialize closures between machines. Unlike Racket futures, DFPS' `touch` form blocks until remote execution of the future completes. DFPS has a distributed variable construct called a `dbox`. For consistency, a dbox should only be written to once or a reduction function for writes to the `dbox` should be provided. Once a `dbox` has be set, the DFPS implementation propagates the `dbox` value other nodes that reference the `dbox`,

Cloud Haskell [5, 6] is a distributed programming platform built in Haskell. Cloud Haskell has two layers of abstraction. The lowest layer is the process layer, which is a message-passing distributed programming API. Next comes the tasks layer which provides a framework for failure recovery and data locality. Communication of serialized closures requires explicit specification from the user of what parts of environment will be serialized and sent with the code object.

On top of its message-passing process layer, Cloud Haskell implements typed channels that allow only messages of a specific type to be sent down the channel. A Cloud Haskell channel has a SendPort and a ReceivePort. ReceivePorts are not serializable and cannot be shared, which simplifies routing. SendPorts, however, are serializable and can be sent to multiple processes, allowing many to one style communication.

High-level Distributed-Memory Parallel Haskell (HdpH) [12] builds upon Cloud Haskell's work by adding support for polymorphic closures and lazy work stealing. HdpH does not require a special language kernel or any modifications to the vanilla GHC runtime. It simply uses GHC's Concurrent Haskell as a systems language for building a distributed memory Haskell.

Dryad [11] is an infrastructure for writing coarse-grain data-parallel distributed programs on the Microsoft platform. Distributed programs are structured as a directed graph. Sequential programs are the graph vertices and one-way channels are the graph edges. Unlike Distributed Places, Dryad is not a programming language. Instead it provides a execution engine for running sequential programs on partitioned data at computational vertices. Although Dryad is not a parallel database, the relational algebra can be mapped on top of a Dryad distributed compute graph. Unlike distributed places which is language centric, Dryad is infrastructure piece, which doesn't extend the expressiveness of any particular programming language.

Jade [15] is a implicitly parallel language. Implemented as a extension to C, Jade is intended to exploit task-level concurrency. Like OpenMP, Jade consists of annotations that programmers add to their sequential code. Jade uses data access and task granularity annotations to automatically extract concurrency and parallelize the program. A Jade front end then compiles the annotated code and outputs C. Programs parallelized with Jade continue to execute deterministically after parallelization. Jade's data model can interact badly with the programs that write to disjoint portions of a single aggregate data structure. In contrast, Distributed Places is an explicitly parallel language where the programmer must explicitly spawn tasks and explicitly handle communication between tasks.

Dreme [7] is a distributed Scheme. All first-class language objects in Dreme are mobile in the network. Dreme describes the communication network between nodes using lexical scope and first class closures. Dreme has a network-wide distributed memory and a distributed garbage collector. By default, Dreme sends objects by reference across the network, which can lead to large quantities of hidden remote operations. In contrast, distributed places copies all objects sent across the network and leaves the programmer responsible for communication invocations and their associated costs.

7 Conclusion

Building distributed places as a language extension allows the compact and clean construction of higher-level abstractions such as RPC, MPI, map reduce, and nested data parallelism. Distributed places programs are more compact and easier to write than traditional C MPI programs. A Racket MPI implementation of parallel k-means was written with distributed places using less than half the lines of code of the original C and MPI version. With distributed places, messages can be heterogeneous and serialization is handled automatically by the language.

In addition to distributed parallel computing, Racket has many features that make it a great coordination and control language. Racket provides a rich FFI (foreign function interface) for invoking legacy C code. Racket also includes extensive process exec capabilities for launching external programs and communicating with them over standard IO pipes. Racket's FFI, process exec capabilities, and distributed places gives programmers a powerful distributed coordination and workflow language.

With distributed places, programmers can quickly develop parallel and distributed solutions to everyday problems. Developers can also build new distributed computing frameworks using distributed places as a common foundation. Distributed places extension of places augments the Racket programmer's toolbox and provides a road map other language implementers to follow.

References

[1] Apache Software Foundation. Hadoop (2012), http://hadoop.apache.org

[2] Blelloch, G.E.: Programming Parallel Algorithms. Communications of the ACM (1996)

[3] Cejtin, H., Jagannathan, S., Kelsey, R.: Higher-Order Distributed Objects. ACM Transactions on Programming Languages and Systems, TOPLAS (1995)

[4] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI 2004: Sixth Symposium on Operating System Design and Implementation (2004)

[5] Epstein, J., Black, A.P., Peyton-Jones, S.: Haskell for the Cloud. In: Proceedings of the 4th ACM Symposium on Haskell, Haskell 2011 (2011)

[6] Epstein, J.: Functional programming for the data centre. MS thesis, University of Cambridge (2011)

[7] Fuchs, M.: Dreme: for Life in the Net. PhD dissertation, New York University (1995)

[8] Germain, G., Feeley, M., Monnier, S.: Concurrency Oriented Programming in Termite Scheme. In: Proc. Scheme and Functional Programming (2006)

[9] Blelloch, G.E., Hardwick, J.C., Chatterjee, S., Sipelstein, J., Zagha, M.: Implementation of a portable nested data-parallel lang. In: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 1993 (1993)

[10] Hewitt, C., Bishop, P., Steiger, R.: A Universal Modular ACTOR Formalism for Artificial Intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI 1973 (1973)

[11] Isard, M., Budiur, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In: European Conference on Computer Systems, EuroSys (2007)

[12] Maier, P., Trinder, P., Loidl, H.-W.: High-level Distributed-Memory Parallel Haskell in Haskell. In: Symposium on Implementation and Application of Functional Languages (2011)

[13] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface (2003), http://www.mpi-forum.org/docs/mpi2-report.pdf

[14] MPICH. MPICH (2013), http://www.mcs.anl.gov/mpich2

[15] Rinard, M.C., Lam, M.S.: The Design, Implementation, and Evaluation of Jade. ACM Transactions on Programming Languages and Systems 20(1), 1–63 (1998)

[16] Sagonas, K., Wilhelmsson, J.: Efficient Memory Management for Concurrent Programs that use Message Passing. Science of Computer Programming 62(2), 98–121 (2006)

[17] Schwendner, A.: Distributed Functional Programming in Scheme. MS thesis, Massachusetts Institute of Technology (2010), http://groups.csail.mit.edu/commit/papers/2010/alexrs-meng-thesis.pdf

[18] Tew, K., Swaine, J., Flatt, M., Findler, R.B., Dinda, P.: Places: Adding Message-Passing Parallelism to Racket. In: Dynamic Language Symposium (2011)

[19] Typesafe Inc. Akka (2012), http://akka.io

Bytecode and Memoized Closure Performance

Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu

Abstract. This article describes a new project to study the memory performance of different closure-implementation strategies in terms of memory allocation and runtime performance. At the heart of the project are four new implementation strategies for closures: three bytecode closures and memoized flat closures. The project proposes to compare the new implementation strategies to the classical strategy that dynamically allocates flat closures as heap data structures. The new bytecode closure representations are based on dynamically creating specialized bytecode instead of allocating a data structure. The first new strategy creates specialized functions by inlining the bindings of free variables. The second uses memoization to reduce the number of dynamically created functions. The third dynamically creates memoized specialized functions that treat free variables as parameters at runtime. The fourth memoizes flat closures. Empirical results from a preliminary byetcode-closure case-study using three small benchmarks are presented as a proof-of-concept. The data suggests that dynamically created bytecode closures in conjunction with memoization can allocate significantly less memory, as much as three orders of magnitude less memory in the presented benchmarks, than a flat closure implementation. In addition to studying the memory footprint of the different closure representations, the project will also compare runtime efficiency of these new strategies with traditional flat closures and flat closures that are unpacked onto the stack.

1 Introduction

In functional languages functions are first-class. This means that functions can be passed as arguments to functions and can be returned as the result of evaluating a function. One of the consequences of first-class functions that programming language implementors must resolve is how to represent functions that may be applied outside of their lexical scope. Care must be taken to represent functions, because they may contain references to *free* variables[1]. For example, consider the function in Figure 1. The function mk-mapper declares the variable f which is free in the returned function. Notice that the returned function can only be applied outside the lexical scope of f. Therefore, f must be "remembered" by the returned function.

[1] A variable, x, is free in a function, f, if f references x, f does not declare x, and x is declared by an ancestor of f in the program's parse tree.

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 58–75, 2014.

```
(define (mk-mapper f)
     (define (mapper L)
         (cond [(null? L) L]
               [else (cons (f (car L)) (mapper (rest L)))]))
   mapper)
```

Fig. 1. A function that returns a function

In the λ-calculus [1], β-reduction is used as the mechanism for remembering the bindings of free variables. The β-rule

$$(\lambda x.e)x_0 \rightarrow e\{x_0/x\}$$

states that all free occurrences of x in e are replaced by x_0. Typical implementations of functional languages, however, do not perform actual substitutions in e and, instead, use an *environment* to track what should have been substituted [2]. Thus, to represent a function, with references to free variables, that may be applied outside its lexical scope, the creation of a closed package, called a *closure* [19], is required. The closure captures the bindings of the free variables by storing (a pointer to) an environment. For example, in Figure 1 a closure is created to retain the binding of `f` for the returned function `mapper`.

Closures, in this context, are functions that are represented using a data structure in order to avoid performing actual substitutions. Part of the data structure represents the function itself (i.e., the code to be evaluated) and part of the data structure represents the environment that gives meaning to the function. This representation facilitates the compilation of functions given that the structure of the function remains constant at runtime (i.e., the bindings of the free variables do not change the compiled function). In contrast, substitution changes the structure of a compiled function every time the bindings of the free variables are different requiring the creation of a new function specialized for the bindings of its free variables.

An alternative to using a data-structure closure to represent a function, of course, is to perform actual substitutions to create a specialized function. Such an approach has been investigated in the past, but implementations to date have led to excessive memory allocation [6,10]. The project described in this article aims to study three strategies for implementing dynamically created byetcode closures instead of dynamically allocating data-structure closures. The memory-efficient strategies are expected to come from a controlled form of actual substitutions employing memoization [17]. This article introduces these implementation strategies using a small, pure, and strict functional language and compares the strategies using small benchmarks–as a preliminary proof-of-concept. The presented empirical data suggests that memory-wise memoized dynamically allocated bytecode closures can be a viable alternative to flat closures. The project also aims to compare the memory allocation of byetcode closures with memoized flat closures. In addition to studying the memory footprint of the different closure representations, the project aims to compare the runtime efficiency

of these new strategies with traditional flat closures and with flat closures that are unpacked onto the stack. Finally, the article ends with other, longer term, interesting lines of research to be pursued.

2 Closures: Representation and Issues

Typically, closures are heap-allocated data structures that are created every time a function with free variables needs to be represented. Historically, closures have been implemented in a number of ways. Early implementations of functional languages, like Henderson's Lisp [11] using a SECD machine [19], used *linked* closures (a.k.a deep closures). In a linked closure, a list of frames (i.e., the existing environment) is used to store the bindings of the free variables. The attractive feature of this approach is that closure creation is done in constant time. Accessing the binding of a free variable, however, requires an $O(n)$ traversal of the list of frames, where n is the lexical offset of the free-variable reference. The space required to store a closure is proportional to the size of the environment–amortized over all the closures that share the environment. This closure representation makes closure creation fast at the expense of making resolving variable references slower [23]. In addition, bindings that are no longer relevant to a computation are unnecessarily kept alive (i.e., not garbage collected) by storing (a pointer to) the entire existing environment as part of the closure.

An alternative to linked closures, used for example by the Functional Abstract Machine (FAM) [3,13], are flat closures (a.k.a. display closures [23]). A flat closure employs an array to store the bindings of free variables. Free variables are accessed by a fixed displacement within the array in constant time. Closure creation requires copying the bindings of free variables into the closure. Therefore, flat-closure creation is $O(v_f)$, where v_f is the number of free variables the function depends on. The space required to store a closure is proportional to the number of free variables a function depends on which is always less than or equal to the size of the environment. This closure representation makes the resolution of references to free variables faster at the expense of closure creation time. In addition, this representation only stores the part of the environment that is relevant to the remaining computation and, thus, allows a garbage collector to be more effective by allowing the recycling of memory space used by bindings that are known to no longer be relevant to the computation.

Shao and Appel observed that flat-closure creation may require many values to be copied repeatedly from closure to closure [25]. To avoid this copying, they developed *safely linked* closures that allow for bindings to be shared between closures. Free variables referenced by more than one function are grouped together into a shareable record. Their representation strategy guarantees that the nesting of safely linked closures never exceeds two. The space required to store a closure is proportional to the number of free variables a function depends on, but when multiple functions have free variables in common the space required is reduced by $(f - 1) * n$, where f is the number of functions that share free

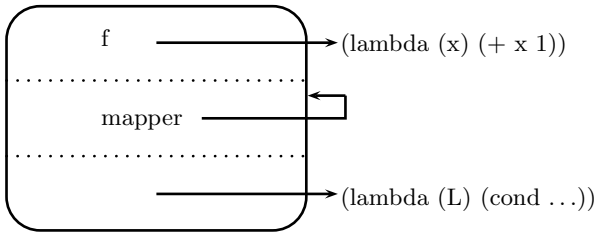


Fig. 2. Conceptual View of the Flat Closure for (mk-mapper (lambda (x) (+ x 1)))

variables and n is the number of free variables the functions share. This closure representation makes closure creation faster than flat-closure creation at the expense of adding overhead to the resolution of references to free variables. In addition, bindings are only kept alive while they may still be relevant to the computation allowing the space they occupy to be recycled as soon as possible by a garbage collector.

Always allocating a closure data structure to represent a function with free variables can lead to excessive memory allocation. This is why many modern implementations of functional languages attempt to eliminate closure allocations whenever possible. For instance, MzScheme [8], which uses flat closures [7], inlines functions and adds free variables as arguments to functions whenever all applications of a function are visible [21]. In addition to reducing memory allocation, providing fast access to free variables is another goal of modern implementations. This has led to a variety of methods to access the bindings of free variables. In MzScheme, for example, the bindings of free variables are not accessed directly from the closure. Instead, the bindings are unpacked onto the stack whenever the closure is applied [7]. Some language implementations make free variables explicit by performing program transformations such as lambda lifting [12,15] and closure conversion [14]. Lambda lifting explicitly adds free variables as parameters to functions. Accesses to free variables in the source program are turned into parameter accesses as done in MzScheme. Closure conversion explicitly adds an environment parameter to functions. The bindings of free variables in the source program are accessed through the environment parameter.

3 Intuitive Data-Structure Closure Elimination

When closures are implemented as data structures, heap memory is allocated every time a function with free variables needs to be represented. For example, consider the code in Figure 1 and the evaluation of:

```
(mk-mapper (lambda (x) (+ x 1))).
```

This expression returns the closure displayed in Figure 2. This conceptual view of the flat closure has f bound to the representation of the combinator that adds 1 to its input. In addition, it has mapper bound to the closure itself, thus, enabling the self-reference (i.e., recursive application) in the body of the function the closure represents.

Instead of allocating and returning a data structure closure, a specialized version of the returned function, based on the binding of f, can be dynamically created. Specifically, if substitutions were performed the function returned would be semantically equivalent to this new function:

```
(define (mapper-f-x-x+1 L)
  (cond [(null? L) L]
        [else (cons ((lambda (x) (+ x 1)) (car L))
                    (mapper-f-x-x+1 (rest L)))])).
```

In this example, the returned function, mapper-f-x-x+1, has the same structure as, mapper, the function specialized. In general, however, the returned specialized function does not require the same structure[2]. The important point is that the returned function is a combinator. That is, it lacks references to free variables and, as such, does not require a data-structure closure to store the bindings of free variables.

Studying variations of three basic strategies to dynamically create such a combinator as a bytecode function, coined a bytecode closure, is a primary focus of the project. The three strategies are outlined as follows:

- The first strategy inlines the bindings of the free variables into the returned function as suggested by β-reduction. The advantage of this implementation strategy is that the resolution of free variables is transformed to accessing constants in specialized functions. A potential disadvantage is that inlining may lead to code explosion and require more memory allocation than flat closures when the functions being specialized are large relative to the number of free variables referenced.
- The second strategy attempts to reduce memory consumption by memoizing inlined functions. That is, the dynamically-created specialized functions of the first strategy are memoized and reused.
- The third strategy breaks away from inlining functions in the source code. Instead, references to free variables in the source code are transformed to parameter references. Specialized bytecode closures inlined with the bindings of free variables that push these bindings onto the stack are memoized. When compared to using flat closures or inlined source functions, the advantages of this implementation strategy are that the resolution of free variables is faster than using flat closures by treating free variables as parameters, that the memory dynamically allocated for specialized functions is proportional to the number of free variables (not the size of the specialized function) as it is for flat closures, and that the structure of compiled λ-terms does not

[2] This can be the result, for example, of performing δ-reductions.

$$
\begin{aligned}
\mathit{program} &\rightarrow \mathit{def}^{*} \\
\mathit{def} &\rightarrow (\textbf{define}\ (\mathit{symbol}^{+})\ \mathit{def}^{*}\ \mathit{expr}) \\
\mathit{expr} &\rightarrow \mathit{number} \\
&\rightarrow \mathit{symbol} \\
&\rightarrow \mathit{boolean} \\
&\rightarrow (\textbf{if}\ \mathit{expr}\ \mathit{expr}\ \mathit{expr}) \\
&\rightarrow (\mathit{expr}^{+}) \\
&\rightarrow (\textbf{lambda}\ (\mathit{symbol}^{*})\ \mathit{expr})
\end{aligned}
$$

Fig. 3. The BNF Grammar of the Source Core Language

change. A potential disadvantage, unlike inlining, is that a jump is required to transfer control from the function that pushes the bindings of the free variables to the function that utilizes these bindings

Comparing the performance of bytecode closures with memoized flat closures and the unpacking of flat (both memoized and not memoized) is part of the project. Memoized flat closures eliminate the need for the jump required by the third strategy at the expense of increasing the access time to free variables. Unpacking a closure may make access to free variables faster when amortized over a relatively large number of references. Empirical data will be collected to determine when, if ever, one implementation strategy is superior to the others.

4 Illustrating the Compilation Process

The BNF grammar for a small core language is displayed in Figure 3. This core language is used for the preliminary results presented in this article. A program consists of zero or more definitions. A definition consists of a header which contains the function name and the parameters, zero or more local definitions, and a body which is an expression. An expression is a number, a symbol, a boolean, an *if* expression, an *application* expression, or a *lambda* expression.

The architecture of the proof-of-concept compiler is displayed in Figure 4. A source program is first parsed. The parse tree is given as input to a *δ-reducer*. The *δ-reducer* replaces a primitive function applied to its required known arguments by a result. This transformation reduces the size of the resulting program by evaluating primitive application expressions and by eliminating dead code (e.g., when the condition of an *if*-expression can be evaluated at compile time). The δ-reduced parse tree is given as input to a lambda lifting function (e.g., [15]). Lambda lifting makes the free variables of a function explicit and, thus, the variables by which to specialize functions at runtime. Given that functions are not curried, the resulting lambda lifted program may contain functions with a nested lambda expression. The parameters of a lifted function are the original function's free variables in the source program and the parameters of the nested lambda expression are the parameters of the original source function. The resulting lambda lifted parse tree is passed to the code generator to produce bytecode.

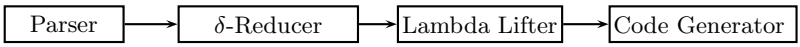


Fig. 4. The General Architecture of the Proof-of-Concept Compiler

To illustrate the process, consider a function common in environment-passing interpreters to evaluate the arguments of an application expression. The function takes as input a list of expressions to be evaluated and the environment (implemented as a list of frames) in which to evaluate the expressions. It returns a list containing the results of evaluating each expression. Using the syntax of Figure 3, the function is implemented as follows:

```
(define (eval-operands rands env)
  (map (lambda (e) (eval-expr e env)) rands))
```

After parsing, the δ-reducer discovers that there are no primitive application expressions that can be evaluated and produces as output the original parse tree. Lambda lifting hoists the lambda expression to the global level. Since env is the only free variable in this function, env is the only parameter in the lifted function. The body of the lifted function is itself the original lambda expression. In the body of eval-operands, the lambda-expression is substituted with an application expression that applies the lifted function to its free variable. After lambda lifting, the parse tree represents the following program:

```
(define (eval-rands rands env) (map (lifted1 env) rands))
(define (lifted1 env) (lambda (e) (eval-expr e env))).
```

The lambda lifted parse tree is passed to the code generator to produce the bytecode displayed in Figure 5. The displayed code is generated assuming the use of flat closures and, to aid readability, Figure 5 omits the proper handling of tail calls. Bytecode is generated for 3 functions: eval-rands, lifted1, and the nested lambda expression in lifted1 (i.e., FN19 in the bytecode). The compiled code for eval-rands sets up an activation record on the stack for the call to map and another for the call to lifted1. For lifted1, env (i.e., PACC 2) is pushed onto the stack and control-flow registers are updated before the call is made. After returning from lifted1, the bytecode pushes rands, the first parameter, onto the stack (i.e., PACC 1), updates control-flow registers, and calls map. The bytecode generated for lifted1 allocates a closure of size 1 for FN19 (i.e., the nested lambda expression), populates the closure with the binding of the first parameter, and returns this closure after popping off its activation record with 1 parameter (i.e. FRETURN 1). The code for FN19 sets up an activation record for the call to eval-expr, pushes e, its parameter, and the free variable env onto the stack (i.e., FVACC 1), updates control-flow registers, and makes the call to eval-expr.

```
eval-rands              lifted1               FN19
  FCALL                   ACLOSURE FN19 1       FCALL
  FCALL                   COPY2CLOSURE 1 1      PACC 1
  PACC 2                  FRETURN 1             FVACC 1
  <update registers>                            <update registers>
  GOTO lifted1                                  GOTO eval-expr
  PACC 1
  <update registers>
  GOTO map
```

Fig. 5. Compiled Code for the MT Virtual Machine

5 Bytecode Closures Implementation Strategies

This section describes the three strategies to dynamically create bytecode closures. It is important to remember that the code generator expects lambda lifted programs in which a lambda expression only exists as the body of a global function and in which lambda expressions contain at least one reference to each of the parameters of its enclosing global function. It is these anonymous functions that are specialized at runtime.

5.1 Strategy I: Inlined Functions

In strategy I, anonymous functions (i.e., compiled λ-terms) are treated as templates with holes. These templates are never executed at runtime and are only used to generate specialized versions of the anonymous function. The holes are the instructions to access free variables (i.e., FVACC instructions in the bytecode).

To generate a specialized function from a template, the bytecode of the template is copied. The holes of the template, however, are filled with instructions to push a constant onto the stack based on the binding of the free variable referenced. That is, specialized functions are inlined with the bindings of the free variables wherever free variables are referenced. Care must be taken to handle jumps to labels correctly (e.g., in the compiled code of an if-expression). Branch instructions that refer to labels can not simply be copied, because that would mean branching into the template instead of a location in the specialized function. The fact that the template and the specialized function have the same number of instructions means that simple address arithmetic solves the problem at runtime.

This strategy uses the *blind* policy of always generating a specialized function whenever a flat closure would be generated. Dynamic function creation using this implementation strategy is $O(n)$, where n is the size of the function being specialized. That is, the time it takes to create a specialized function is proportional to the number of bytecode instructions in the function and not the number of free variables the function references. In general when the size of specialized

```
(define (eval-expr expr env)
   (if (literal? expr)
       expr
       ...
       (if (app-expr? expr)
           (apply-proc (eval-expr (proc-expr expr) env)
                       (cons (eval-operands (ops-expr expr) env) env))
       ...)))
(define (eval-operands rands env)
   (map (lambda (e) (eval-expr e env)) rands))
```

Fig. 6. Program Fragment of an Environment-Passing Interpreter

```
(define (lifted1-1 e) (eval-expr e '(((x 2) (y 2)))))
(define (lifted1-2 e) (eval-expr e '(((x 2) (y 2)))))
(define (lifted1-3 e) (eval-expr e '((b (f 2)) ((x 2) (y 2)))))
(define (lifted1-4 e) (eval-expr e '((b (f 2)) ((x 2) (y 2)))))
(define (lifted1-5 e)
  (eval-expr e '(((i (g (f 2))) (j (g (f 2)))) ((x 2) (y 2)))))
```

Fig. 7. Five Dynamically Created Inlined Functions

functions is large relative to the number of free variables referenced, it is expected for such an implementation to be inefficient when compared to using flat closures for two reasons. The first is that programs allocate more memory. The second is that specialized function creation takes longer than closure creation.

To illustrate this strategy, consider the program fragment in Figure 6 for an environment-passing interpreter and the evaluation of

(eval-expr '(h (g (f x)) (g (f y))) '(((x 2) (y 2)))),

where f, g, and h are user-defined functions, and the environment binds x and y to 2. The function eval-operands[3] is called 5 times: once for h, twice for g, and twice for f. The evaluation of both applications of f is done with the same environment (i.e., the displayed environment). Likewise, the evaluation of both applications of g is done with the same environment (i.e., value-wise). The result at runtime is the generation of the 5 functions[4] displayed in Figure 7. Notice that the generated functions for f, lifted-1 and lifted-2, and the generated functions for g, lifted-3 and lifted-4, are, respectively, semantically equivalent. This means that three specialized functions can be generated, instead of five, to evaluate the expression. Generated functions that are semantically equivalent to needed functions can be re-used to reduce memory allocation.

[3] This function is lambda lifted as described in Section 4.

[4] In the interest of readability, source syntax is used in this example.

5.2 Strategy II: Memoized Inlined Functions

The second implementation strategy does not blindly create specialized functions. Instead of always generating a function when a closure would be allocated, specialized functions are memoized and functions are only dynamically created when needed. Specialized function memoization requires a cache of specialized functions to be maintained. If a specialized function is needed and is found in this cache, then the previously generated specialized function is re-used. Otherwise, a new specialized function is generated and this new function is added to the cache of specialized functions. Determining function equality is achieved by exploiting the naming convention used for specialized functions. Instead of simply generating a fresh identifier, the fresh identifier is a linear combination of the name of the function being specialized and of the types and the bindings of the free variables.

To illustrate how memoized function specialization works, once again, consider the program fragment in Figure 6 for an environment-passing interpreter and the evaluation of:

(eval-expr '(h (g (f x)) (g (f y))) '(((x 2) (y 2)))).

As before, the function eval-operands is called five times, but only three specialized functions are generated[5]:

```
(define (lifted1-list-100-2400 e)
  (eval-expr e '(((x 2) (y 2)))))
(define (lifted1-list-3000-5000 e)
  (eval-expr e '(((b (f 2))) ((x 2) (y 2)))))
(define (lifted1-list-7500-8150 e)
  (eval-expr e '(((i ((g (f 2)))) (j ((g (f 2)))))
                 ((x 2) (y 2))))).
```

Notice that in this example we have a 40% reduction in the number of dynamically generated functions when compared to using strategy I.

5.3 Strategy III: Memoized Auxiliary Inlined Functions

In strategy III, λ-terms are not treated as templates with holes. Instead, these anonymous functions are executable and are converted to combinators. Free-variable references become parameter references. In this context, a specialized byetcode function has two roles. The first is to push the bindings of free variables needed by an anonymous function onto the stack (akin to unpacking a flat closure). The second is to transfer control to the anonymous function for which it was created. A specialized function, in other words, completes the construction of the front rib of the environment for an anonymous function.

[5] The function names are based on the linear combination convention mentioned above.

```
lifted1
  GENF FN19 1
  FRETURN 1
FN19
  FCALL
  PACC 1
  PACC 2
  <update registers>
  GOTO eval-expr
```

Fig. 8. Strategy III Lambda Expression

```
lifted1-list-100-2400
          PUSHLIST 100 2400
          GOTO FN19
lifted1-list-3000-5000
          PUSHLIST 3000 5000
          GOTO FN19
lifted1-list-7500-8150
          PUSHLIST 7500 8150
          GOTO FN19
```

Fig. 9. Strategy III Specialized Functions

The third implementation strategy makes dynamic function creation memory efficient by making the size of specialized functions proportional to the size of the flat closures they substitute. A specialized function is a collection of instructions to push constants onto the stack followed by a jump instruction. At compile time, the order in which constants are to be pushed onto the stack by a specialized function is determined. This order corresponds to the order of the parameters of a lambda-lifted function that has an anonymous function in its body. Every parameter of such a function must be referenced by the λ-expression in its body. Therefore, to create a specialized function, the runtime system only needs to examine the first rib of the environment to assemble the instructions to push constants onto the stack in the right order and to add a jump to the anonymous function.

To illustrate how function specialization works using strategy III, once again, consider the program fragment in Figure 6 and the evaluation of

(eval-expr '(h (g (f x)) (g (f y))) '(((x 2) (y 2)))).

The compiled code for lifted1 and its nested anonymous function (i.e., FN19) are displayed in Figure 8. Observe that the compiled code for FN19 is almost the same as the compiled code in Figure 5. The only difference is that the reference to the first free variable (i.e., FVACC 1) is now a reference to the second parameter (i.e., PACC 2). The compiled code for lifted1 generates a specialized function for FN19 that adds one parameter to FN19's activation record (i.e., GENF FN19 1). As for strategy II, the function eval-operands is called 5 times and only 3 specialized functions are generated which are displayed using bytecode in Figure 9. One function is generated for each of the different bindings for env. Each generated function pushes the binding of env onto the stack and transfers control to FN19. It is straightforward to see that the specialized versions of lifted1 are smaller than their counterparts using strategies I or II and are proportional in size to flat closures.

6 Preliminary Empirical Results

This section presents preliminary memory allocation empirical results obtained from three small benchmarks. These results are intended solely as an indication

that there is fertile ground for exploration using larger benchmarks. First, the benchmarks are briefly described. Second, the performance measurements are presented. The benchmarks naively use higher-order functions to test extreme ends of the memory allocation spectrum. The lambda lifted versions of the benchmarks are found in the appendix in section 8.

6.1 Benchmarks

AP. This benchmark traverses a list of pairs of integers to produce a list that contains the sums of each pair. The presented measurements are for a list of 9,999 pairs with each pair containing two randomly generated integers in [0..9999].

ST. This benchmark traverses a binary tree of integers and scales each integer in the tree by its depth in the tree. The presented measurements are for the scaling of a full binary tree of depth 15.

TK. This is the triply recursive integer function related to the Takeuchi function, one of Gabriel's benchmarks [9], that has been modified to maximize the use of anonymous functions. The presented measurements are for (tak 18 12 6).

Each benchmark was executed 4 times for a total of 12 experiments on a non-distributed version of the MT virtual machine [16]. The benchmarks were executed using flat closures and each of the three strategies for bytecode closures described in the previous section, denoted by *Strategy I*, *Strategy II*, and *Strategy III*.

6.2 Measurements and Analysis

For each benchmark, Figure 10 displays the relative difference, $\frac{fca-bsa}{bsa}$, in memory allocation between flat closures allocations (fca) and each bytecode strategy allocation (bsa). A negative relative difference means that the flat-closure-based implementation allocated less memory.

For each benchmark, strategy I incurs the maximum number of allocations. The total excess memory allocation ranges from about 20% to about 90% when compared to the flat-closure-based implementation. This occurs, as expected, because the number of dynamically created functions is the same as the number of closures allocated and the size of a specialized function is larger than the size of a flat closure. These numbers clearly suggest that such a naive implementation of bytecode closures is neither efficient nor feasible for industrial-strength implementations.

Strategies II and III significantly outperform the flat-closure-based implementation (as well as Strategy I). For the AP benchmark, the flat closure based implementation allocates about 33% more memory than either of these strategies. The savings in memory allocation are due to memoization exploiting the modest amount of repetition in the generation of random numbers in [0..9999].

	Relative Difference
AP Strategy I	-0.6190
AP Strategy II	0.3325
AP Strategy III	0.3332
TK Strategy I	-0.8889
TK Strategy II	79.13
TK Strategy III	794.3
ST Strategy I	-0.1998
ST Strategy II	0.2497
ST Strategy III	0.2900

Fig. 10. Relative Difference with Flat Closures

Strategies II and III virtually exhibit the same performance with strategy III displaying slightly less memory allocation. The observed performance is so close, because the function being specialized is small and a specialized inlined function generated with strategy II is only one bytecode instruction larger than a specialized function generated with strategy III. This benchmark clearly suggests that memoization of dynamic functions may significantly reduce memory allocation when compared to flat closures and that further study is justified.

For the ST benchmark, the flat closure base implementation allocates about 25% more memory than strategy II and 29% more memory than strategy III. For this benchmark, memory allocation is dominated by allocation to build a list-based structure (i.e., a full binary tree). For strategies II and III, only a small number of functions, 16, are dynamically created. In essence, only one specialized function is created per tree level due to memoization. The savings observed are attributed to the large number of flat closures allocated by the the classical implementation (one for each node in the full binary tree). The difference between strategy II and strategy III is due to the smaller functions generated by the latter. This benchmark clearly suggests that even for programs in which first-class functions only play a small role, memoized bytecode closures may significantly reduce memory allocation and that further study is warranted.

For the TK benchmark, we observe the largest gain in performance over the flat-closure-based implementation. For strategy II the flat-closure-based implementation allocates about 7,913% more memory (i.e., two orders of magnitude more memory) while for strategy III the excess allocation by the flat-closure-based implementation reaches 79,430% (i.e., three orders of magnitude more memory). The difference is quite significant and occurs because the Takeuchi triply recursive function makes many recursive calls with the same arguments. This benchmark presents the ideal conditions under which memoization is most effective. Strategy III significantly outperforms strategy II by about 1 order of magnitude. This difference occurs, because the inlined specialized functions generated using strategy II are significantly larger than the specialized functions generated by strategy III. The TK benchmark clearly suggests that memoized

dynamically generated functions may lead to significantly less memory allocation than flat closures and that such performance potential deserves further study.

The preliminary empirical data clearly suggests that the thesis that memoized bytecode closures can exhibit significantly better memory performance than flat closures and deserve further study. Furthermore, the data also suggests that keeping the size of dynamically created bytecode closures proportional to the number of free variables is important.

7 Related Work

Feeley and Lapalme first suggested generating code instead of allocating data structure closures [6]. In their work, a specialized function only pushes the bindings of free variables onto the stack and these bindings are accessed like parameters by a compiled lambda expression akin to strategy III described in this article. Their performance measurements indicate that for their implementation strategy memory allocated for specialized functions increases by up to 25% when compared to a closure-based implementation. The major differences between with the approach described in this article and their work is the use of memoization and lambda lifting. Memoization can significantly reduce memory allocation as argued in the previous section. Lambda lifting reduces the complexity of compiling for specialization, but at runtime exactly the same specialized functions are created by both approaches. Finally, Feeley and Lapalme also address the problems introduced by assignment and propose pushing the address to a mutable box instead of a binding to extend the technique to support assignment. A similar approach would work with the memoization-based strategies described in this article.

More recently, Grabmüller developed a prototype system to implement closures using runtime code generation for a strict (and pure) functional language [10]. Instead of compiled code, this approach uses abstract syntax trees at runtime to create specialized functions. The runtime code generator inlines functions with the bindings of their free variables akin to strategy I described in this article. The use of abstract syntax trees is intended to simplify common optimizations (e.g., reduction to normal form and dead code elimination) that can be performed by the runtime code generator once the bindings of free variables are known. It is unclear, based on the preliminary work done with their prototype, if any runtime analysis of an abstract syntax tree can not be done a priori to indicate to the code generator what optimizations to perform. Grabmüller's performance evaluation indicates that the system runs out of memory space for some benchmarks, but provides no other indication on memory allocation performance.

There have been several approaches, far too many to reference here, to runtime code generation that have not focused on eliminating data-structure closures. Lee and Leone's FABIUS compiler specialize curried functions by inlining the bindings of arguments as they are received [20]. Consel and Noël have used runtime specialization for C programs that uses templates with holes to inline

and partially evaluate functions [4]. Poletto et al. have also used dynamic code generation to improve the performance of a superset of C called 'C that requires programmers to annotate their code [22]. The work on 'C has been extended to a dialect of Java called DynJava to generate type safe specialized classes [18].

8 Concluding Remarks

This article describes a new project to study the memory performance of representing closures as dynamically allocated bytecode functions and as memoized flat closures. The preliminary empirical data presented suggests that memoized bytecode closures can significantly reduce memory allocation. The magnitude of the savings increases for programs in which first-class functions play a significant role at runtime reaching up to three orders of magnitude less allocation than flat closures in the presented benchmarks. The inescapable conclusion is that memoized bytecode closures is a technology worthy of future study. In addition, note that the memoization strategy described in this article does not break the high-level of abstraction provided by functional languages. That is, it does not require the programmer to be aware of the memoization process nor to annotate programs for function specialization to occur.

In addition to studying the memory performance of bytecode closures, this work will pursue several other interesting lines of research such as:

Memoized Flat Closures. What impact do memoized flat closures have on performance? Clearly, the number of flat closures will be the same as for Strategy III bytecode closures. Their memory footprint and their allocation time will also be similar given that both are proportional to the number of free variables. The difference, if any, will be marked by free-variable access time.

Runtime Performance. How do the different closure representations impact running time? It is a generally accepted that a smaller memory footprint is better. This project will collect empirical evidence to quantify the impact. Furthermore, we will compare unpacking closures onto the stack with the representation used in Strategy III.

Inflation of Parameters. The bytecode closures presented in this article are based on a compiler that performs lambda lifting. Danvy and Schultz showed that lambda lifting may present efficiency difficulties due to parameter inflation which led them to propose lambda dropping[5]. A fundamental line of research is to determine if bytecode closures overcome the efficiency problems raised by the inflation of parameters.

Continuations. It is common for functional programs to be transformed to continuation-passing style (CPS) [24,26,27]. A continuation can be represented as a function that knows how to complete the rest of the computation. Many implementations, however, transform continuations to a data structure representation. Another fundamental line of research is whether or not bytecode closures eliminate the need for this change.

Garbage Collection. The performance of memoized bytecode closures hinges on their reuse. Performance, however, also hinges on the recycling of memory by a garbage collector. How should memoized closures be garbage collected (whether of the bytecode nature or the data structure nature)? What rules or heuristics can be used to prevent premature recycling of memoized closures?

Acknowledgements. The author thanks Olivier Danvy for his thoughtful comments over the years on the research questions posed by what is now this new project.

References

1. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics, Revised edn. Studies in Logic and the Foundations of Mathematics. North-Holland (1984)
2. Biernacka, M., Danvy, O.: A Concrete Framework for Environment Machines. ACM Trans. Comput. Logic 9(1) (December 2007)
3. Cardelli, L.: Compiling a Functional Language. In: Proceedings of the 1984 ACM Conference on LISP and Functional Programming, pp. 208–217. ACM Press, New York (1984)
4. Consel, C., NoëL, F.: A General Approach for Run-time Specialization and its Application to C. In: 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, pp. 145–156. ACM Press (1996)
5. Danvy, O., Schultz, U.P.: Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. Theoretical Computer Science 248(1-2), 243–287 (2000)
6. Feeley, M., Lapalme, G.: Closure Generation Based on Viewing Lambda as Epsilon Plus Compile. Journal of Computer Languages 17(4), 251–267 (1992)
7. Matthew Flatt. Private Communication (May 2007)
8. Flatt, M.: PLT MzScheme: Language Manual. Technical Report PLT-TR2008-1-v4.1, PLT Scheme Inc. (2008), `http://www.plt-scheme.org/techreports/`
9. Gabriel, R.P.: Performance and Evaluation of Lisp Systems. MIT Press, Cambridge (1985)
10. Grabmüller, M.: Implementing Closures Using Run-time Code Generation. Research report 2006-02 in Forschungsberichte Fakultät IV – Elektrotechnik und Informatik, Technische Universität Berlin (February 2006)
11. Henderson, P.: Functional Programming: Application and Implementation. Prentice-Hall International, Englewood (1980)
12. Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations. In: Proc. of a Conf. on Functional Prog. Lang. and Comp. Arch., pp. 190–203. Springer-Verlag New York, Inc. (1985)
13. Cardelli, L.: The Functional Abstract Machine. Technical Report No.107, Bell Laboratories (April 1983)
14. Minamide, Y., Morrisett, G., Harper, R.: Typed Closure Conversion. In: Proc. of he 23rd ACM Symp. on Principles of Progr. Lang., pp. 271–283. ACM Press (1996)
15. Morazán, M.T., Schultz, U.P.: Optimal Lambda Lifting in Quadratic Time. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 37–56. Springer, Heidelberg (2008)

16. Morazán, M.T., Troeger, D.R.: The MT Architecture and Allocation Algorithm. In: Michaelson, G., Trinder, P., Loidl, H.-W. (eds.) Trends in Functional Programming, Bristol, UK, vol. 1, pp. 97–104. Intellect (2000)
17. Norvig, P.: Techniques for Automatic Memoization with Applications to Context-Free Parsing. Comput. Linguist. 17(1), 91–98 (1991)
18. Oiwa, Y., Masuhara, H., Yonezawa, A.: DynJava: Type Safe Dynamic Code Generation in Java. In: Third JSSST Work. on Progr. and Progr. Lang. (March 2001)
19. Landin, P.J.: The Mechanical Evaluation of Expressions. The Computer Journal 6(4), 308–320 (1964)
20. Lee, P., Leone, M.: Optimizing ML with Run-Time Code Generation. In: Proc. of the ACM SIGPLAN Conf. on Progr. Lang. Design and Impl., pp. 137–148. ACM Press (May 1996)
21. PLT Scheme Inc. Guide: PLT Scheme (2008), `http://docs.plt-scheme.org/guide/index.html`
22. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, M.F.: 'C and tcc: A Language and Compiler for Dynamic Code Generation. ACM Transactions on Programming Languages and Systems 21(2), 324–369 (1999)
23. Kent Dybvig, R.: The Development of Chez Scheme. In: Proc. of the Eleventh ACM SIGPLAN Int. Conf. on Funct. Prog., pp. 1–12 (September 2006)
24. Reynolds, J.C.: The Dicoveries of Continuations. Lisp and Symbolic Computation 6(3/4) (1993)
25. Shao, Z., Appel, A.W.: Space Efficient Closure Representations. In: Proc. of the 1994 ACM Conf. on LISP and Funct. Prog., pp. 150–161. ACM Press, New York (1994) ISBN 0-89791-643-3
26. Strachey, C., Wadsworth, C.P.: Continuations: A Mathematical Semantics for Handling Full Jumps. Higher-Order and Symbolic Computation 13(1/2) (2000)
27. Sussman, G.J., Steele Jr., G.L.: Scheme: An Interpreter for Extended Lambda Calculus. In: MEMO 349, MIT AI LAB (1975)

A Appendix

A.1 The AP Benchmark

```
(define (g x) (lambda (y) (+ x y)))
(define (f x) ((g (car x)) (cdr x)))
(define (mklist len modus)
    (if (= len 0) '()
        (cons (cons (random modus) (random modus))
              (mklist (- len 1) modus))))
(define (benchmark n modus) (map f (mklist n modus)))
```

A.2 The TK Benchmark

```
(define (tak-y x) (lambda (y) (tak-z y x)))
(define (tak-z y x)
  (lambda (z) (if (not (< y x)) z
                  (tak
                    (tak (- x 1) y z)
                    (tak (- y 1) z x)
                    (tak (- z 1) x y)))))
(define (tak x y z) (((tak-y x) y) z))
```

A.3 The ST Benchmark

```
(define (scaleT-by-depth T) (scale 0 T))
(define (scale d T) (map (scale-function d) T))
(define (scale-function d)
  (lambda (t) (if (number? t) (* d t) (scale (+ d 1) t))))
(define (mkbt d)
   (if (= d 0) '()
       (cons d (cons (mkbt (- d 1))
                     (cons (mkbt (- d 1)) '())))))
(define (benchmark x) (scaleT-by-depth (mkbt x)))
```

Towards Efficient Abstractions for Concurrent Consensus*

Carlo Spaccasassi** and Vasileios Koutavas***

Trinity College Dublin, Ireland
{spaccasc,Vasileios.Koutavas}@scss.tcd.ie

Abstract. Consensus is an often occurring problem in concurrent and distributed programming. We present a programming language with simple semantics and build-in support for consensus in the form of communicating transactions. We motivate the need for such a construct with a characteristic example of generalized consensus which can be naturally encoded in our language. We then focus on the challenges in achieving an implementation that can efficiently run such programs. We setup an architecture to evaluate different implementation alternatives and use it to experimentally evaluate runtime heuristics. This is the basis for a research project on realistic programming language support for consensus.

Keywords: Concurrent programming, consensus, communicating transactions.

1 Introduction

Achieving consensus between concurrent processes is a ubiquitous problem in multicore and distributed programming [8, 6]. Among the classic instances of consensus is leader election and synchronous multi-process communication. Programming language support for consensus, however, has been limited. For example, CML's first-class communication primitives provide a programming language abstraction to implement two-party consensus. However, they cannot be used to abstractly implement consensus between three or more processes [11, Thm. 6.1]—this needs to be implemented in a case-by-case basis.

Let us consider a hypothetical scenario of generalized consensus, which we will call the *Saturday Night Out* (SNO) problem. In this scenario a number of friends are seeking partners for various activities on Saturday night. Each has a list of desired activities to attend in a certain order, and will only agree for a night out if there is a partner for each activity. Alice, for example, is looking for company to go out for dinner and then a movie (not necessarily with the same person). To find partners for these events in this order she may attempt to synchronize on the "handshake" channels `dinner` and `movie`:

* Student project paper (primarily the work of the first author).
** Supported by MSR (MRL 2011-039)
*** Supported by SFI project SFI 06 IN.1 1898.

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 76–90, 2014.

Alice $\stackrel{\text{def}}{=}$ **sync** dinner; **sync** movie

Here **sync** is a two-party synchronization operator, similar to CSP synchronization. Bob, on the other hand, wants to go for dinner and then for dancing:

Bob $\stackrel{\text{def}}{=}$ **sync** dinner; **sync** dancing

Alice and Bob can agree on dinner but they need partners for a movie and dancing, respectively, to commit to the night out. Their agreement is *tentative*.

Let Carol be another friend in this group who is only interested in dancing:

Carol $\stackrel{\text{def}}{=}$ **sync** dancing

Once Bob and Carol agree on dancing they are both happy to commit to going out. However, Alice has no movie partner and she can still cancel her agreement with Bob. If this happens, Bob and Carol need to be notified to cancel their agreement and everyone starts over their search of partners. An implementation of the SNO scenario between concurrent processes would need to have a specialized way of reversing the effect of this synchronization. Suppose David is also a participant in this set of friends.

David $\stackrel{\text{def}}{=}$ **sync** dancing; **sync** movie

After the partial agreement between Alice, Bob, and Carol is canceled, David together with the first two can synchronize on dinner, dancing, and movie and agree to go out (leaving Carol at home).

Notice that when Alice raised an objection to the partial agreement between her, Bob, and Carol, all three participants had to restart. However, if Carol was taken out of the agreement (even after she and Bob were happy to commit their plans), David would have been able to take Carol's place and the work of Alice and Bob until the point when Carol joined in would not need to be repeated.

Programming SNO between an arbitrary number of processes (which can form multiple agreement groups) in CML is complicated. Especially if we consider that the participants are allowed to perform arbitrary computations between synchronizations affecting control flow, and can communicate with other parties not directly involved in the SNO. For example, Bob may want to go dancing only if he can agree with the babysitter to stay late:

Bob $\stackrel{\text{def}}{=}$ **sync** dinner; **if** babysitter() **then sync** dancing

In this case Bob's computation has side-effects outside of the SNO group of processes. To implement this would require code for dealing with the SNO protocol to be written in the Babysitter (or any other) process, breaking modularity.

This paper shows that *communicating transactions*, a recently proposed mechanism for automatic error recovery in CCS processes [13], is a useful mechanism for modularly implementing the SNO and other generalized consensus scenarios. They provide a construct for *non-isolated* (communicating), *all-or-nothing* (transactional) computation, with which we can give implementations of the

T	$::= \mathbf{unit} \mid \mathbf{bool} \mid \mathbf{int} \mid T \times T \mid T \to T \mid T\,\mathbf{chan}$	Types
v	$::= x \mid () \mid \mathbf{true} \mid \mathbf{false} \mid n \mid (v, v) \mid \mathbf{fun}\, f(x) = e \mid c$	Values
e	$::= v \mid (e, e) \mid e\ e \mid op\ e \mid \mathbf{let}\, x = e\, \mathbf{in}\, e \mid \mathbf{if}\, e\, \mathbf{then}\, e\, \mathbf{else}\, e$ $\mid \mathbf{send}\, e\ e \mid \mathbf{recv}\, e \mid \mathbf{newChan}_T \mid \mathbf{spawn}\, e$ $\mid \mathbf{atomic}\, [\![e \rhd_k e]\!] \mid \mathbf{commit}\, k$	Expressions
P	$::= e \mid P \parallel P \mid \nu c.P \mid [\![P \rhd_k P]\!] \mid \mathbf{co}\, k$	Processes
op	$::= \mathbf{fst} \mid \mathbf{snd} \mid \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{leq}$	Operators
E	$::= [] \mid (E, e) \mid (v, E) \mid E\ e \mid v\ E \mid op\ E \mid \mathbf{let}\, x = E\, \mathbf{in}\, e$ $\mid \mathbf{if}\, E\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid \mathbf{send}\, E\ e \mid \mathbf{send}\, v\ E \mid \mathbf{recv}\, E \mid \mathbf{spawn}\, E$	Eval. Contexts
where $n \in \mathbb{N}$, $x \in \mathit{Var}$, $c \in \mathit{Chan}$, $k \in \mathcal{K}$		

Fig. 1. TCML syntax

SNO participants that resemble the above pseudocode. Previous work on communicating transactions focused on behavioral theory with respect to *safety* and *liveness* [13, 14]. However, the effectiveness of this construct in a pragmatic programming language has yet to be proven. One of the main milestones to achieve on this direction is the invention of efficient runtime implementations of communicating transactions. Here we describe the challenges and our first results in a recently started project to investigate this direction.

In particular, we equip a simple concurrent functional language with communicating transactions and use it to discuss the challenges in making an efficient implementation of such languages (Sect. 2). This language contains a novel combination of sequential evaluation and communicating transactions, making it more appropriate for programming compared to the CCS-based calculus of previous work [13, 14]. In this language we give a modular implementation of consensus scenarios such as the SNO example, where participants are oblivious of their environment and can communicate with arbitrary processes (such as the Babysitter process) without the need to add code for the SNO protocol in those processes. Moreover, the above more efficient, partially aborting strategy is captured in this semantics.

Our semantics of this language is non-deterministic, allowing different runtime scheduling strategies of processes, some more efficient than others. To study their relative efficiency we have developed a skeleton implementation of the language which allows us to plug in and evaluate such runtime strategies (Sect. 3). We describe several such strategies (Sect. 4) and report the results of our evaluations (Sect. 5). Finally, we summarize related work in this area and the future directions of this project (Sect. 6).

2 The TCML Language

We study TCML, a language combining a simply-typed λ-calculus with π-calculus and communicating transactions. For this language we use the abstract syntax shown in Fig. 1 and the usual abbreviations from the λ- and π-calculus.

IF-TRUE	$\textbf{if}\,\textbf{true}\,\textbf{then}\,e_1\,\textbf{else}\,e_2$	$\hookrightarrow e_1$	
IF-FALSE	$\textbf{if}\,\textbf{false}\,\textbf{then}\,e_1\,\textbf{else}\,e_2$	$\hookrightarrow e_2$	
LET	$\textbf{let}\,x = v\,\textbf{in}\,e$	$\hookrightarrow e[v/x]$	
OP	$op\,v$	$\hookrightarrow \delta(op, v)$	
APP	$\textbf{fun}\,f(x) = e\;v_2$	$\hookrightarrow e[\textbf{fun}\,f(x) = e/f][v_2/x]$	
STEP	$E[e]$	$\longrightarrow E[e']$	if $e \hookrightarrow e'$
SPAWN	$E[\textbf{spawn}\,v]$	$\longrightarrow v\;() \parallel E[()]$	
NEWCHAN	$E[\textbf{newChan}_T]$	$\longrightarrow \nu c.E[c]$	if $c \notin \mathrm{fc}(E[()])$
ATOMIC	$E[\textbf{atomic}\,[\![\, e_1 \rhd_k e_2 \,]\!]]$	$\longrightarrow [\![\, E[e_1] \rhd_k E[e_2] \,]\!]$	
COMMIT	$E[\textbf{commit}\,k]$	$\longrightarrow \textbf{co}\,k \parallel E[()]$	

Fig. 2. Sequential reductions

Values in TCML are either constants of base type (**unit**, **bool**, and **int**), pairs of values (of type $T \times T$), recursive functions ($T \to T$), and channels carrying values of type T ($T\,\textbf{chan}$). A simple type system (with appropriate progress and preservation theorems) can be found in an accompanying technical report [12].

Source TCML programs are expressions in the functional core of the language, ranged over by e, whereas running programs are processes derived from the syntax of P. Besides standard lambda calculus expressions, the functional core contains the constructs $\textbf{send}\,c\;e$ and $\textbf{recv}\,c$ to synchronously send and receive a value on channel c, respectively, and $\textbf{newChan}_T$ to create a new channel of type $\textbf{chan}\,T$. The constructs **spawn** and **atomic**, when executed, respectively spawn a new process and transaction; $\textbf{commit}\,k$ commits transaction k—we will shortly describe these constructs in detail.

A simple running process can be just an expression e. It can also be constructed by the parallel composition of P and Q ($P \parallel Q$). We treat free channels as in the π-calculus, considering them to be *global*. Thus if a channel c is free in both P and Q, it can be used for communication between these processes. The construct $\nu c.P$ encodes π-calculus restriction of the scope of c to process P. We use the Barendregt convention for bound variables and channels and identify terms up to alpha conversion. We also write $\mathrm{fc}(P)$ for the free channels in P.

Process $[\![\, P_1 \rhd_k P_2 \,]\!]$ encodes a communicating transaction. This can be thought of as the process P_1, the *default* of the transaction, which runs until the transaction *commits*. If, however, the transaction *aborts* then P_1 is discarded and the entire transaction is replaced by its *alternative* process P_2. Intuitively, P_2 is the continuation of the transaction in the case of an abort. TCML provides a mechanism for P_1 to communicate with its environment. This mechanism guarantees that P_1 has an all-or-nothing behavioral semantics (see [14]). Hence the name communicating transactions. As we will see, commits are asynchronous, requiring the process $\textbf{co}\,k$ in the language. The name k of the transaction is bound in P_1. Thus only the default of the transaction can potentially spawn a $\textbf{co}\,k$. The meta-function $\mathrm{ftn}(P)$ gives us the free transaction names in P.

Processes with no free variables can reduce using transitions of the form $P \longrightarrow Q$. These transitions for the functional part of the language are shown in Fig. 2 and are defined in terms of reductions $e \hookrightarrow e'$ (where e is a *redex*) and

eager, left-to-right evaluation contexts E whose grammar is given in Fig. 1. Due to a unique decomposition lemma, an expression e can be decomposed to an evaluation context and a redex expression in only one way. Here we use $e[u/x]$ for the standard capture-avoiding substitution, and $\delta(op, v)$ for a meta-function returning the result of the operator op on v, when this is defined.

Rule Step lifts functional reductions to process reductions. The rest of the reduction rules of Fig. 2 deal with the concurrent and transactional side-effects of expressions. Rule Spawn reduces a $\mathbf{spawn}\, v$ expression at evaluation position to the unit value, creating a new process running the application $v\ ()$. The type system of the language guarantees that value v here is a thunk. With this rule we can derive the reductions:

$$\mathbf{spawn}(\lambda().\, \mathbf{send}\, c\ 1); \mathbf{recv}\, c \longrightarrow (\lambda().\, \mathbf{send}\, c\ 1)\ () \parallel \mathbf{recv}\, c \longrightarrow \mathbf{send}\, c\ 1 \parallel \mathbf{recv}\, c$$

The resulting processes of these reductions can then communicate on channel c. As we previously mentioned, the free channel c can also be used to communicate with any other parallel process. Rule NewChan gives processes the ability to create new, locally scoped channels. Thus, the following expression will result in an input and an output process that can *only* communicate with each other:

$$\begin{array}{l}\mathbf{let}\, x = \mathbf{newChan_{int}}\, \mathbf{in}\, (\mathbf{spawn}\, (\lambda().\, \mathbf{send}\, x\ 1); \mathbf{recv}\, x) \\ \longrightarrow \nu c.\, (\mathbf{spawn}\, (\lambda().\, \mathbf{send}\, c\ 1); \mathbf{recv}\, c) \longrightarrow^{*} \nu c.\, (\mathbf{send}\, c\ 1 \parallel \mathbf{recv}\, c)\end{array}$$

Rule Atomic is a novel rule that deals with the combination of communicating transactions and sequential computations. This rule applies when a new transaction is started from within the current (expression-only) process, engulfing the entire process in it, and storing the abort continuation in the alternative of the transaction. Rule Commit spawns an asynchronous commit. Transactions can be arbitrarily nested, thus we can write:

$$\begin{array}{l}\mathbf{atomic}\ [\![\, \mathbf{spawn}(\lambda().\, \mathbf{recv}\, c; \mathbf{commit}\, k) \rhd_k ()\,]\!]; \\ \mathbf{atomic}\ [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!] \\ \longrightarrow [\![\, \mathbf{spawn}(\lambda().\, \mathbf{recv}\, c; \mathbf{commit}\, k);\ \ \mathbf{atomic}\ [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!] \\ \quad \rhd_k ();\ \ \mathbf{atomic}\ [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!]\,]\!] \\ \longrightarrow^{*} [\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \parallel [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!] \\ \quad \rhd_k (); \mathbf{atomic}\ [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!]\,]\!]\end{array}$$

This process will commit the k-transaction after an input on channel c and the inner l-transaction after an input on d. As we will see, if the k transaction aborts then the inner l-transaction will be discarded (even if it has performed the input on d) and the resulting process (the alternative of k) will restart l: $(); \mathbf{atomic}\ [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!]$. The effect of this abort will be the rollback of the communication on d reverting the program to a consistent state.

Process and transactional reductions are handled by the rules of Fig. 3. The first four rules (Sync, Eq, Par, and Chan) are direct adaptations of the reduction rules of the π-calculus, which allow parallel processes to communicate, and propagate reductions over parallel and restriction. These rules use an omitted

$$\textsc{Sync}\quad \frac{}{E_1[\mathbf{recv}\, c] \parallel E_2[\mathbf{send}\, c\, v] \longrightarrow E_1[v] \parallel E_2[()]} \qquad \textsc{Eq}\quad \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

$$\textsc{Par}\quad \frac{P_1 \longrightarrow P_1'}{P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2} \qquad \textsc{Chan}\quad \frac{P \longrightarrow P'}{\nu c.P \longrightarrow \nu c.P'}$$

$$\textsc{Emb}\quad \frac{}{P_1 \parallel [\![\, P_2 \rhd_k P_3 \,]\!] \longrightarrow [\![(P_1 \parallel P_2) \rhd_k (P_1 \parallel P_3)]\!]} \qquad \textsc{Step}\quad \frac{P \longrightarrow P'}{[\![\, P \rhd_k P_2 \,]\!] \longrightarrow [\![\, P' \rhd_k P_2 \,]\!]}$$

$$\textsc{Co}\quad \frac{P_1 \equiv \mathbf{co}\, k \parallel P_1'}{[\![\, P_1 \rhd_k P_2 \,]\!] \longrightarrow P_1'/k} \qquad \textsc{Abort}\quad \frac{}{[\![\, P_1 \rhd_k P_2 \,]\!] \longrightarrow P_2}$$

Fig. 3. Concurrent and Transactional reductions (omitting symmetric rules)

structural equivalence ($\equiv$) to identify terms up to the reordering of parallel processes and the extrusion of the scope of restricted channels, in the spirit of the π-calculus semantics. Rule Step propagates reductions of default processes over their respective transactions. The remaining rules are taken from TransCCS [13].

Rule Emb encodes the *embedding* of a process P_1 in a parallel transaction $[\![\, P_2 \rhd_k P_3 \,]\!]$. This enables the communication of P_1 with P_2, the default of k. It also keeps the current continuation of P_1 in the alternative of k in case it aborts. To illustrate the mechanics of the embed rule, let us consider the above nested transaction running in parallel with the process $P = \mathbf{send}\, d\, (); \mathbf{send}\, c\, ()$:

$$[\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \parallel [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l () \,]\!] \rhd_k (); \mathbf{atomic}\, [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l () \,]\!]]\!] \quad \parallel \quad P$$

After two embedding transitions we will have

$$[\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \quad \parallel \quad [\![\, P \parallel \mathbf{recv}\, d; \mathbf{commit}\, l \quad \rhd_l \quad P \parallel () \,]\!] \quad \rhd_k \quad P \parallel \ldots]\!]$$

Now P can communicate on d with the inner transaction:

$$[\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \quad \parallel \quad [\![\, \mathbf{send}\, c\, () \parallel \mathbf{commit}\, l \quad \rhd_l \quad P \parallel () \,]\!] \quad \rhd_k \quad P \parallel \ldots]\!]$$

Next, there are (at least) two options: either $\mathbf{commit}\, l$ spawns a $\mathbf{co}\, l$ process which causes the commit of the l-transaction, or the input on d is embedded in the l-transaction. Let us assume that the latter occurs:

$$[\![\ [\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \ \parallel \ \mathbf{send}\, c\, () \ \parallel \ \mathbf{commit}\, l \ \rhd_l (\mathbf{recv}\, c; \mathbf{commit}\, k) \ \parallel \ P \ \parallel \ ()]\!] \rhd_k P \parallel \ldots]\!] \longrightarrow^* [\![\ [\![\ [\![\, \mathbf{co}\, k \parallel \mathbf{co}\, l \rhd_l \ldots]\!] \rhd_k \ldots]\!]$$

The transactions are now ready to commit from the inner-most to the outer-most using rule Commit. Inner-to-outer commits are necessary to guarantee that all transactions that have communicated have reached an agreement to commit.

This also has the important consequence of making the following three processes behaviorally indistinguishable:

$$[\![P_1 \rhd_k P_2]\!] \parallel [\![Q_1 \rhd_l Q_2]\!]$$
$$[\![P_1 \parallel [\![Q_1 \rhd_l Q_2]\!] \rhd_k P_2 \parallel [\![Q_1 \rhd_l Q_2]\!]]\!]$$
$$[\![[\![P_1 \rhd_k P_2]\!] \parallel Q_1 \rhd_l [\![P_1 \rhd_k P_2]\!] \parallel Q_2]\!]$$

Therefore, an implementation of TCML, when dealing with the first of the three processes can pick any of the alternative, non-deterministic mutual embeddings of the k and l transactions without affecting the observable outcomes of the program. In fact, when one of the transactions has no possibility of committing or when the two transactions never communicate, an implementation can decide *never* to embed the two transactions in each-other. This is crucial in creating implementations that will only embed processes (and other transactions) only when necessary for communication, and pick the most *efficient* of the available embeddings. The development of implementations with efficient embedding strategies is one of the main challenges of our project for scaling communicating transactions to pragmatic programming languages.

Similarly, aborts are entirely non-deterministic (ABORT) and are left to the discretion of the underlying implementation. Thus in the above example any transaction can abort at any stage, discarding part of the computation. In such examples there is usually a multitude of transactions that can be aborted, and in cases where a "forward" reduction is not possible (due to deadlock) aborts are necessary. Making the TCML programmer in charge of aborts (as we do with commits) is not desirable since the purpose of communicating transactions is to lift the burden of manual error prediction and handling. Minimizing the number aborts and picking aborts that rewind the program minimally but sufficiently to reach a successful outcome is another major challenge in our project.

The SNO scenario can be simply implemented in TCML using *restarting transactions*. A restarting transaction uses recursion to re-initiate an identical transaction in the case of an abort:

$$\mathbf{atomic}_{rec\ k}\ [\![e]\!] \quad \stackrel{\text{def}}{=} \quad \mathbf{fun}\, r() = \mathbf{atomic}\, [\![e \rhd_k r\ ()]\!]$$

A transactional implementation of the SNO participants we discussed in the introduction simply wraps their code in restating transactions:

$$\begin{array}{l}
\mathbf{let}\ \mathit{alice} \ = \mathbf{atomic}_{rec\ k}\ [\![\mathit{sync\ dinner};\ \mathit{sync\ movie};\ \mathbf{commit}\, k]\!]\ \mathbf{in} \\
\mathbf{let}\ \mathit{bob} \ = \mathbf{atomic}_{rec\ k}\ [\![\mathit{sync\ dinner};\ \mathit{sync\ dancing};\ \mathbf{commit}\, k]\!]\ \mathbf{in} \\
\mathbf{let}\ \mathit{carol} \ = \mathbf{atomic}_{rec\ k}\ [\![\mathit{sync\ dancing};\ \mathbf{commit}\, k]\!]\ \mathbf{in} \\
\mathbf{let}\ \mathit{david} \ = \mathbf{atomic}_{rec\ k}\ [\![\mathit{sync\ dancing};\ \mathit{sync\ movie};\ \mathbf{commit}\, k]\!]\ \mathbf{in} \\
\mathbf{spawn}\ \mathit{alice};\ \mathbf{spawn}\ \mathit{bob};\ \mathbf{spawn}\ \mathit{carol};\ \mathbf{spawn}\ \mathit{david}
\end{array}$$

Here *dinner*, *dancing*, and *movie* are implementations of CSP synchronization channels and *sync* a function to synchronize on these channels. Compared to a potential ad-hoc implementation of SNO in CML the simplicity of the above

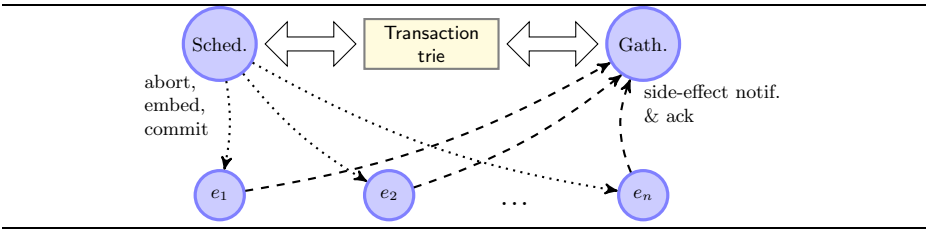


Fig. 4. TCML runtime architecture

code is evident (the version of Bob communicating with the Babysitter is just as simple). However, as we discuss in Sect. 5, this simplicity comes with a severe performance penalty, at least for straightforward implementations of TCML. In essence, the above code asks from the underlying transactional implementation to solve an NP-complete satisfiability problem. Leveraging existing useful heuristics for such problems is something we intend to pursue in future work.

In the following we describe an implementation where these transactional scheduling decisions can be plugged in, and a number of heuristic transactional schedulers we have developed and evaluated. Our work shows that advanced heuristics bring measurable performance benefits but the exponential number of runtime choices require innovative compilation and execution techniques to make communicating transactions a realistic solution for programmers.

3 An Extensible Implementation Architecture

We have developed an interpreter for the TCML semantics in Concurrent Haskell [7, 10] to which we can plug-in different decisions about the non-deterministic transitions of our semantics with the runtime architecture in Fig. 4.

The main Haskell threads are shown as round nodes in the figure. Each concurrent functional expression e_i is interpreted in its own thread according to the sequential reduction rules in Fig. 2 of the previous section. Side-effects in an expression are handled by the interpreting thread, creating new channels, spawning new threads, and starting new transactions. Our implementation of synchronous, dynamically created channels is on top of Haskell's `MVar`s, and guarantees that only processes within the same transactions can communicate.

Except for channel creation, the evaluation of all other side-effects in an expression will cause a *notification* (shown as dashed arrows in Fig. 2) to be sent to the *gatherer* process (Gath.). This process is responsible for maintaining a global view of the state of the running program in a *Trie* data-structure. This data-structure essentially represents the transactional structure of the program; i.e., the logical nesting of transactions and processes inside running transactions:

```
data TTrie = TTrie { threads   :: Set ThreadID,
                     children  :: Map TransactionID TTrie, ... }
```

A `TTrie` node represents a transaction, or the top-level of the program. The main information stored in such a node is the set of threads (`threads`) and transactions (`children`) running in that transactional level. Each child transaction

has its own associated `TTrie` node. An invariant of the data-structure is that each thread and transaction identifier appears only once in it. For example the complex program we saw on Fig. 3:

$$\begin{array}{l}[\![(\mathbf{recv}\, c; \mathbf{commit}\, k)^{\mathtt{tid}_1} \parallel [\![(\mathbf{recv}\, d; \mathbf{commit}\, l)^{\mathtt{tid}_2} \rhd_l ()]\!] \\ \rhd_k (); \mathbf{atomic}\, [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()]\!]]\!] \qquad\qquad \parallel P^{\mathtt{tid}_P}\end{array}$$

will have an associated trie:

```
TTrie{threads  = {tid_P},
      children = {k ↦ TTrie{threads = {tid_1},
                            children = {l ↦ TTrie{threads = {tid_2},
                                                  children = ∅}}}}}
```

The last ingredient of the runtime implementation is the *scheduler* thread (Sched. in Fig. 4). This makes decisions about the commit, embed and abort transitions to be performed by the expression threads, based on the information in the trie. Once such a decision is made by the scheduler, appropriate signals (implemented using Haskell asynchronous exceptions [10]) are sent to the running threads, shown as dotted lines in Fig. 4. Our implementation is parametric to the precise algorithm that makes scheduler decisions, and in the following section we describe a number of such algorithms we have tried and evaluated.

A scheduler signal received by a thread will cause the update of the *local transactional state* of the thread, affecting the future execution of the thread. The local state of a thread is an object of the `TProcess` data-type:

```
data TProcess = TP {
  expr :: Expression,
  ctx  :: Context,
  tr   :: [Alternative] }

data Alternative = A {
  tname :: TransactionID,
  pr    :: TProcess }
```

The local state maintains the expression (`expr`) and evaluation context (`ctx`) currently interpreted by the thread and a list of *alternative* processes (represented by objects of the `Alternative` data-type). This list contains the continuations stored when the thread was embedded in transactions. The nesting of transactions in this list mirrors the transactional nesting in the global trie and is thus compatible with the transactional nesting of other expression threads. Let us go back to the example of Fig. 3:

$$\begin{array}{l}[\![(\mathbf{recv}\, c; \mathbf{commit}\, k)^{\mathtt{tid}_1} \parallel [\![(\mathbf{recv}\, d; \mathbf{commit}\, l)^{\mathtt{tid}_2} \rhd_l ()]\!] \\ \rhd_k (); \mathbf{atomic}\, [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()]\!]]\!] \qquad\qquad \parallel P^{\mathtt{tid}_P}\end{array}$$

where $P = \mathbf{send}\, d\, (); \mathbf{send}\, c\, ()$. When P is embedded in both k and l, the thread evaluating P will have the local state object

```
TP{expr = P, tr = [A{tname = l, pr = P}, A{tname = k, pr = P}]}
```

recording the fact that the thread running P is part of the l-transaction, which in turn is inside the k-transaction. If either of these transactions aborts then

the thread will rollback to P, and the list of alternatives will be appropriately updated (the aborted transaction will be removed).

Once a transactional reconfiguration is performed by a thread, an acknowledgment is sent back to the gatherer, who, as we discussed, is responsible for updating the global transactional structure in the trie. This closes a cycle of transactional reconfigurations initiated from the process (by starting a new transaction or thread) or the scheduler (by issuing a commit, embed, or abort).

What we described so far is a simple prototype architecture for an interpreter of TCML. Improvements are possible; for example, the gatherer is a message bottleneck, and together with the scheduler they are single points of failure in a potential distributed setting. But such concerns are beyond the scope of this paper. In the following section we discuss various policies for the scheduler which we then evaluate experimentally.

4 Transactional Scheduling Policies

Our goal here is to investigate schedulers that make decisions on transactional reconfiguration based only on runtime heuristics. We are currently working on more advanced schedulers, including schedulers that take advantage of static information extracted from the program, which we leave for future work.

An important consideration when designing a scheduler is *adequacy* [15, Sec. 11.4]. For a given program, an adequate scheduler can produce *all outcomes* that the non-deterministic operational semantics give for that program. However, this does not mean that the scheduler should be able to produce *all traces* of the non-deterministic semantics. Many of these traces will unnecessarily abort and restart the computations. Previous work on the behavioral theory of communicating transactions has shown that all program outcomes can be reached with traces that *never* restart a computation [13]. Thus a goal for schedulers is to minimize re-computations by minimizing aborts.

Moreover, as we discussed at the end of Sect. 2, many of the exponential number of embeddings can be avoided without altering the observable behavior of a program. This can be done by embedding a process inside a transaction only when this embedding is necessary to enable communication between the process and the transaction. We take advantage of this in a *communication-driven* scheduler we describe in this section.

Even after reducing the number of possible non-deterministic choices faced by the scheduler, in most cases we are still left with a multitude of alternative transactional reconfiguration options. Some of these are more likely to lead to efficient traces than other. However, to preserve adequacy we cannot exclude any of these options since the scheduler has no way to foresee their outcomes. In these cases use heuristics to assign different, non-zero probabilities to available choices, which leads to measurable performance improvements without violating adequacy. Of course some program outcomes might be more likely to appear than others. This approach trades quantitative fairness for performance improvement.

However, the probabilistic approach is *theoretically fair*. Every finite trace leading to a program outcome has a non-zero probability. Diverging traces due

to sequential reductions also have non-zero probability to occur. The only traces with zero probability are those in the reduction semantics that have an infinite number of non-deterministic reductions. Intuitively, these are unfair traces that abort and restart transactions *ad infinitum*, even if other options are possible.

Random Scheduler (R). The first scheduler we consider is the random scheduler, whose policy at each point is to simply select one of all available non-deterministic choices with equal probability, with no exception. Any available abort, embed, or commit actions are equally likely to happen. For example, this scheduler might decide at any time to embed Bob into Carol's transaction, or abort David. As one would expect, this is not particularly efficient; it is, however, obviously adequate and fair according to the discussion above. If a reduction transition is available infinitely often, scheduler R will eventually select it.

There is much room for improvement. Suppose transaction k can commit: $[\![P \parallel \mathbf{co}\, k \rhd_k Q]\!]$. Since R makes no distinction between the choices of committing and aborting k, it will often unnecessarily abort k. All processes embedded in this transaction will have to roll back and re-execute; if k was a transaction that restarts, the transaction will also re-execute. This results to a considerable performance penalty. Similarly, scheduler R might preemptively abort a long-running transaction that could have committed, given enough time and embeddings.

Staged Scheduler (S). The staged scheduler partially addresses these issues by prioritizing its available choices. Whenever a transaction is ready to commit, scheduler S will always decide to send a commit signal to that transaction before aborting it or embedding another process in it. This does not violate adequacy; before continuing with the algorithm of S, let us examine the adequacy of prioritizing commits over other transactional actions with an example.

Example 1. *Consider the following program in which k is ready to commit: $[\![P \parallel \mathbf{co}\, k \rhd_k Q]\!] \parallel R$. If embedding R in k leads to a program outcome, then that outcome can also be reached after committing k from the residual $P \parallel R$.*

Alternatively, a program outcome could be reachable by aborting k (from the process $Q \parallel R$). However, the $\mathbf{co}\, k$ was spawned from one of the previous states of the program in the current trace. In that state, transaction k necessarily had the form: $[\![P' \parallel E[\mathbf{commit}\, k] \rhd_k Q]\!]$, and the abort of k was enabled. Therefore, the staged interpreter indeed allows a trace leading to the program state $Q \parallel R$ from which the outcome in question is reachable. □

If a transaction T cannot commit, S prioritizes embeddings into T over abort of T. This decision is adequate because transactions that take an abort reduction before an embed step have an equivalent abort reduction after that step. When no commit nor embed options are available, the staged interpreter lets the transaction run with probability 0.95 to progress more in the current trace, and aborts it with probability 0.05—these numbers have been fine-tuned experimentally.

This heuristic greatly improves performance by minimizing unnecessary aborts. Its drawback is that it does not abort transactions often, thus program outcomes

reachable only from transactional alternatives are less likely to appear. Moreover, this scheduler does not avoid *unnecessary embeddings*.

Communication-Driven Scheduler (CD). To avoid spurious embeddings, scheduler CD improves over R by performing an embed transition only if it is *necessary* for an imminent communication. For example, at the very start of the SNO example the CD scheduler can only choose to embed Alice into Bob's transaction or vice versa, because they are the only processes ready to synchronize on *dinner*. Because of the equivalence $[\![P \rhd_k Q]\!] \parallel R \equiv_{\text{cxt}} [\![P \parallel R \rhd_k Q \parallel R]\!]$ which we previously discussed, this scheduler is adequate.

For the implementation of this scheduler we augment the information in the trie data-structure (Sect. 3) with channels with a pending communication operation (if any). In Sect. 5 we show that this heuristic noticeably boosts performance because it greatly reduces the exponential number of embedding choices.

Delayed-Aborts Scheduler (DA). The final scheduler we report is DA, which adds a minor improvement upon scheduler CD. This scheduler keeps a timer for each running transaction k in the trie, and resets it whenever a non-sequential operation happens inside k. Transaction k can be aborted only when its timer expires. This strategy benefits transactions that perform multiple communications before committing. The CD scheduler is adequate because it only adds time delays.

5 Evaluation of the Interpreters

We now report the experimental evaluation of interpreters using the preceding scheduling policies. The interpreters were compiled with GHC 7.0.3, and the experiments were performed on a Windows 7 machine with Intel® Core™i5-2520M (2.50 GHz) and 8GB of RAM. We run several versions of two programs:

1. The three-way rendezvous (3WR) in which a number of processes compete to synchronize on a channel with *two* other processes, forming groups of three which then exchange values. This is a standard example of multi-party agreement [11, 3, 5]. In the TCML implementation of this example each process nondeterministically chooses between being a *leader* or *follower* within a communicating transaction. If a leader and two followers communicate, they can all exchange values and commit; any other situation leads to deadlock and eventually to an abort of some of the transactions involved.
2. The SNO example of the introduction, as implemented in Sect. 2, with multiple instances of the Alice, Bob, Carol, and David processes.

To test scheduler scalability, we tested versions of the above programs with a different number of competing parallel processes. Each process in these programs continuously performs 3WR or SNO cycles and our interpreters are instrumented to measure the number of operations in a given period, from which we compute the *mean throughput* of successful operations. The results are shown in Fig. 5.

Each graph in the figure contains the mean throughput of operations (in logarithmic scale) as a function of the number of competing concurrent TCML

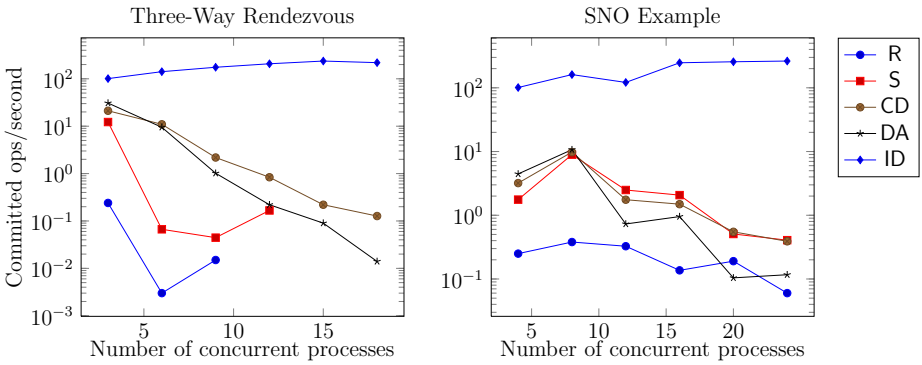


Fig. 5. Experimental Results

processes. The graphs contain runs with each scheduler we discussed (random R, staged S, communication-driven, CD, and communication-driven with delayed aborts DA) as well as with an *ideal* non-transactional program (ID). The ideal program in the case of 3WR is similar to the TCML, non-abstract implementation [11]. The ideal version of the SNO is running a simpler instance of the scenario, without any Carol processes—this instance has no deadlocks and therefore needs no error handling. Ideal programs give us a performance upper bound.

As predictable, the random scheduler (R)'s performance is the worst; in many cases R could not perform any operations in the window of measurements (30sec).

The other schedulers perform better than R by an order of magnitude. Even just prioritizing the transactional reconfiguration choices significantly cuts down the exponential number of inefficient traces. However, none of the schedulers scale to programs with more processes; their performance deteriorates exponentially. In fact, when we go from the communication-driven (CD) to the delayed aborts (DA) scheduler we see worst throughput in larger process pools. This is because with many competing processes there is more possibility to enter a path to deadlock; in these cases the results suggest that it is better to abort early.

The upper bound in the performance, as shown by the throughput of ID is one order of magnitude above that of the best interpreter, when there are few concurrent processes, and (within the range of our experiments) two orders when there are many concurrent processes. The performance of ID is increasing with more processes due to better utilization of the processor cores.

It is clear that in order to achieve a pragmatic implementation of TCML we need to address the exponential nature in consensus scenarios as the ones we tested here. Our exploration of purely runtime heuristics shows that performance can be improved, but we need to turn to a different approach to close the gap between ideal ad-hoc implementations and abstract TCML implementations.

6 Related Work and Conclusions

Consensus is common problem in concurrent and distributed programming. The need for developing programming language support for consensus has already been identified in previous work on *transactional events* (TE) [3], *communicating memory transactions* (CMT) [9], *transactors* [4] and *cJoin* [1]. These approaches propose forms of communicating transactions, similar to those described in Sect. 2. All approaches can be used to an extent to implement generalized consensus scenarios, such as the instance of the Saturday Night Out (SNO) example in this paper. Without such constructs the programmer needs to devise and implement complex low-level protocols for consensus. Stabilizers [16] add transactional support for fault-tolerance in the presence of transient faults but do not directly address consensus scenarios such as the SNO example. Our work here is based on *communicating transactions* which is the only construct to date with a provably intuitive behavioral theory [13, 14].

TE extends CML events with a transactional sequencing operator; transactional communication is resolved at runtime by search threads which exhaustively explore all possibilities of synchronization. CMT extends STM with asynchronous communication, maintaining a directed dependency graph mirroring communication between transactions; STM abort triggers cascading aborts to transactions that have received values from aborting transactions. Transactors extend actor semantics with fault-tolerance primitives, enabling the composition of systems with consistent distributed state via distributed checkpointing. The cJoin calculus extends the Join calculus with isolated transactions which can merge at runtime; merging and aborting are managed by the programmer, offering a manual alternative to TCML's nondeterministic transactional operations.

Reference implementations have been developed for TE, CMT, and cJoin (in JoCaml). The discovery of efficient implementations for communicating transactions can be equally beneficial for all approaches.

This paper presented TCML, a simple functional language with build-in support for consensus via communicating transactions. This is a construct with a robust behavioral theory supporting its use as a programming language abstraction for automatic error recovery [13, 14]. TCML has a simple operational semantics and can simplify the programming of advanced consensus scenarios; we introduced such an example (SNO) which has a natural encoding in TCML. We have motivated this construct as a programming language solution to the problem of programming consensus. To our knowledge, this is the most intricate and general application of such constructs in a concurrent and distributed setting. However, communicating transactions could address challenges in other application domains, such as *speculative computing* [2].

The usefulness of communicating transactions in real-world applications, however, depends on the invention of efficient implementations. This paper described the obstacles to overcome and our first results in a recently started project. We gave a framework and a modular implementation to develop and evaluate current and future schedulers of communicating transactions, and used it to examine schedulers based solely on runtime heuristics. We have found that some of them

improve upon the performance of a naive randomized implementation but do not scale to programs with significant contention, where exponential numbers of computation paths lead to necessary rollbacks. It is clear that purely dynamic strategies do not lead to sustainable performance improvements.

In future work we intend to explore the extraction of information from the source code to guide the language runtime. This information can include an abstract model of the communication behavior of processes in order to predict their future communication pattern. A promising approach is the development of technology in type and effect systems and static analysis. Although scheduling communicating transactions is theoretically computationally expensive, realistic performance in many programming scenarios could be achievable.

References

[1] Bruni, R., Melgratti, H., Montanari, U.: Nested commits for mobile calculi: Extending join. In: Levy, J.-J., Mayr, E.W., Mitchell, J.C. (eds.) Expl. New Frontiers of Theor. Informatics. IFIP, vol. 155, pp. 563–576. Springer, Heidelberg (2004)
[2] Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: POPL, pp. 209–220. ACM, NY (2005)
[3] Donnelly, K., Fluet, M.: Transactional Events. In: ICFP, pp. 124–135. ACM, NY (2006)
[4] Field, J., Varela, C.A.: Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In: POPL, pp. 195–208. ACM, NY (2005)
[5] Harris, T., Marlow, S., Jones, S.P.L., Herlihy, M.: Composable memory transactions. Commun. ACM, 91–100 (2008)
[6] Herlihy, M., Shavit, N.: The art of multiprocessor programming. Kaufmann (2008)
[7] Jones, S.P.L., Gordon, A.D., Finne, S.: Concurrent Haskell. In: POPL, pp. 295–308. ACM, NY (1996)
[8] Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press (2008)
[9] Lesani, M., Palsberg, J.: Communicating memory transactions. In: PPoPP 2011, pp. 157–168. ACM, NY (2011)
[10] Marlow, S., Jones, S.P.L., Moran, A., Reppy, J.H.: Asynchronous exceptions in Haskell. In: PLDI 2001, pp. 274–285. ACM, NY (2001)
[11] Reppy, J.H.: Concurrent programming in ML. Cambridge University Press (1999)
[12] Spaccasassi, C.: Transactional Concurrent ML. Tech. Rep. TCD-CS-2013-01, Trinity College Dublin (2013)
[13] de Vries, E., Koutavas, V., Hennessy, M.: Communicating Transactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 569–583. Springer, Heidelberg (2010)
[14] de Vries, E., Koutavas, V., Hennessy, M.: Liveness of Communicating Transactions. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 392–407. Springer, Heidelberg (2010)
[15] Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. Foundations of computing. MIT Press (1993)
[16] Ziarek, L., Schatz, P., Jagannathan, S.: Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In: Reppy, J.H., Lawall, J.L. (eds.) ICFP, pp. 136–147. ACM, NY (2006)

Blame Prediction

Dries Harnie*, Christophe Scholliers, and Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel, Belgium
{dharnie,cfscholl,wdmeuter}@vub.ac.be

Abstract. Static type systems are usually conservative. Therefore, many interesting programs are rejected by the static type system, even though they may execute without errors. Dynamic type systems allow such ill-typed programs to run, at the cost of run-time errors. The cause of runtime errors is often far removed from the place where the type errors are raised, making the program hard to debug. We present a novel typing discipline called *blame prediction* which transforms programs in order to detect runtime type errors as soon as they are guaranteed to happen. These type errors relate the future type error with its cause, aiding in debugging. As a proof of concept, we have applied blame prediction to a functional Scheme-like language and evaluated our system against soft typing.

Keywords: type systems, dynamic typing, blame prediction.

1 Introduction

In recent years there has been a surge in the use of dynamically typed programming languages. Developers are building large systems for all kinds of purposes, taking advantage of the fast prototyping and short edit-run-debug cycles offered by these languages. These advantages are attributed to the fact that all dynamically typed programs are allowed to run. The interpreter detects and reports errors at runtime. When a program halts on a runtime error, the developer has to figure out the expression that caused the error solely from the error description and a stack trace.

In the meantime, researchers have developed expressive type systems that attempt to ascribe types to ever-increasing subsets of dynamically typed programming languages. This research focused on Scheme-like languages in the nineties [1,2,3], but recent research has focused on other languages like Ruby and Javascript [4,5].

Static and dynamic approaches aim to achieve different goals: static typing approaches are conservative and report type errors up front, but they *only* allow programs to run if no type errors can be found. By contrast, dynamic typing approaches allow every program to run, but report runtime errors when the type tests in primitive operations fail. This means that statically identifiable type errors will only be reported long after they *could* have been reported.

* Funded by the Prospective Research For Brussels program of Innoviris, Brussels.

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 91–106, 2014.

In this paper we present *blame prediction*: a hybrid technique that makes primitive type tests explicit and performs them as early as possible. The key insight of blame prediction is that primitive operations can be decoupled from the type tests they need to perform, which allows these type tests to be performed *much* earlier. When a type test fails and *all* code paths after the test require it to succeed, a blame prediction error is raised. This error references both the faulty expression and the primitive operations that depend on the type test. Using these blame prediction errors, programmers can more easily debug their code. We have implemented a proof-of-concept blame prediction transformation for a functional core of Scheme. We are convinced that blame prediction can be also applied to a whole class of dynamic programming languages and type systems.

The rest of this paper is structured as follows: we describe the motivation and the idea behind blame prediction in sections 2 and 3. Section 4 defines the blame prediction transformation. Related work is examined in section 6. Finally, we perform a first evaluation of blame prediction in section 5.

2 Motivation

In this section we explain the tension between dynamic and static typing by exploring a small example: a number guessing game. This game consists of the user trying to guess a hidden target number, for every guess the program gives feedback like "too high" or "too low". If the user guesses the number, the game is over. Consider the following Scheme implementation of the number guessing game:

```
(define (guess target)
  (let ((input (read)))
    (cond ((eq? input 'quit) 'quit)
          ((< input target)
            (display "Too low!\n") (guess target))
          ((> input target)
            (display "Too high!\n") (guess target))
          (else
            (display "Correct!\n") 'done))))
```

Listing 1.1. Simple number guessing game in Scheme

This program uses the `read` primitive function to read input from the user and parse it as a Scheme value. If this input is the symbol `'quit`, the game is immediately stopped. Otherwise the input is assumed to be a guess, so it is compared to the `target` number. If the guess was correct, the game ends, otherwise a feedback message is printed and the game continues.

The following steps will occur when applying a static type discipline to the program: first of all, because the `read` primitive can return a value of *any* type, the type of `input` is inferred as $\top$ (which represents all Scheme types). In the first branch of the **`cond`**, the `eq?` check narrows the type of `input` to symbols, as the `eq?` primitive requires two input expressions of the same type. Then,

in the second branch, `input` is numerically compared to `target`, which needs two numeric arguments. As this requires `input` to have a numeric type while it already has type symbol, a type error is signaled and the program is rejected. Since the program is rejected, we cannot run the program.

A dynamic typing discipline has the opposite result: the program runs normally, but type tests are performed at runtime and errors are raised if they fail. In this case, after the `read` primitive returns a value, it is compared to the symbol `quit`. This does not require a runtime type test, only a pointer equality test. The calls to `<`, `>` and `=` all verify that both `input` and `target` are numbers before they execute the actual comparison code. If these type tests fail, an error is signaled and evaluation is aborted.

Both approaches fall short: static typing simply rejects the program, while dynamic typing accepts the program but it will crash if given the wrong input. We require a hybrid approach: one that applies static typing for the most part, while shielding expressions that might raise errors with type tests. One way of making this program safer is by inserting dynamic type checks in the right places. For example, the code assumes that the `input` variable is either a number or a symbol, as evidenced by the `eq?` and `<` tests. The programmer could therefore insert a type test into the code, right below the **`let`**:

```
(define (guess target)
  (let ((input (read)))
    (if (not (or (symbol? input) (number? input)))
      (error "Invalid type for input")
      (cond ...))))
```

Listing 1.2. Guessing game with an extra type test

If the user now enters an unexpected kind of input, an error is raised immediately. This program has properties of both the static and dynamic approaches: it allows the program to run as long as no runtime type errors are raised, but an error is raised as soon as possible if errors can be predicted. We discuss how to mechanize this transformation in the next section.

3 Blame Prediction

In this section we specify blame prediction for a functional subset of Scheme called Scheme$_\beta$. Its syntax and semantics are described in Figure 1, in the style of Felleisen and Hieb [6]. Note that the syntax is assumed to be in administrative normal form (ANF)[7,8], which makes the order of evaluation — and that of type tests — explicit.

The most important evaluation rule here is E-Apply, which is responsible for applying both primitive operations and user-defined functions. Evaluation of an application happens as follows: both the function and its arguments are evaluated and then passed to the δ function. If the function being applied is a primitive operation like `+`, the types of the arguments are tested and a `err-not-int` error is raised if one of the arguments is not an integer. A second case is when

a user-defined function is invoked, in which case the number of arguments must match the arity of the function. Otherwise an err-args-λ error is raised. Finally, application of a non-function value results in a err-not-λ error. Note that — just like in Scheme — these errors are raised at the time of application and propagate outwards, halting the evaluation process.

$$
\begin{array}{rcll}
\mathrm{e} \in \mathrm{Exp} & ::= & s & \text{simple expressions} \\
 & & (s\ s_1 \ldots s_n) & \text{application} \\
 & & (\mathsf{if}\ s\ e\ e) & \text{conditional} \\
 & & (\mathsf{let}\ (x\ e)\ e) & \text{let} \\
\mathrm{s} \in \mathrm{Simp} & ::= & x & \text{variables} \\
 & & \#f \mid \#t \mid \mathsf{n} & \text{constants and literals} \\
 & & (\mathsf{lambda}\ (x_1 \ldots x_n)\ e) & \text{lambda expressions} \\
 & & & \\
v & ::= & \#f \mid \#t \mid \mathsf{n} \mid \lambda x_1 \ldots x_n.e & \text{Runtime values} \\
E & ::= & \square \mid (\mathsf{let}\ (x\ E)\ e) & \text{Evaluation contexts}
\end{array}
$$

$$
\begin{array}{rlll}
\text{(E-If-False)} & E\langle(\mathsf{if}\ \#f\ e_1\ e_2)\rangle & \rightarrow E\langle e_2\rangle & \\
\text{(E-If-True)} & E\langle(\mathsf{if}\ v\ e_1\ e_2)\rangle & \rightarrow E\langle e_1\rangle & \text{if } v \neq \#f \\
\text{(E-Let)} & E\langle(\mathsf{let}\ (x\ v)\ e)\rangle & \rightarrow E\langle e[v/x]\rangle & \\
\text{(E-Lambda)} & E\langle(\mathsf{lambda}\ (x_1 \ldots x_n)\ e)\rangle & \rightarrow E\langle\lambda x_1 \ldots x_n.e\rangle & \\
\text{(E-Apply)} & E\langle(v_f\ v_1 \ldots v_n)\rangle & \rightarrow E\langle\delta(v_f, v_1, \ldots, v_n)\rangle & \\
\text{(E-Error)} & E\langle\text{err-}\omega(v)\rangle & \rightarrow \text{err-}\omega(v) &
\end{array}
$$

$$
\begin{array}{rcll}
\delta(+, v_1, v_2) & = & v_1 + v_2 & \text{if } \mathsf{int?}(v_1) \wedge \mathsf{int?}(v_2) \\
\delta(+, v_1, v_2) & = & \text{err-not-int}(v_1) & \text{if } \neg\mathsf{int?}(v_1) \\
\delta(+, v_1, v_2) & = & \text{err-not-int}(v_2) & \text{if } \neg\mathsf{int?}(v_2) \\
\delta(\lambda x_1 \ldots x_m.e, v_1, \ldots, v_n) & = & e[v_1 \ldots v_n / x_1 \ldots x_m] & \text{if } m = n \\
\delta(\lambda x_1 \ldots x_m.e, v_1, \ldots, v_n) & = & \text{err-args-}\lambda(\lambda x_1 \ldots x_m.e) & \text{if } m \neq n \\
\delta(v, \ldots) & = & \text{err-not-}\lambda(v) & \text{if } \neg\mathsf{function?}(v)
\end{array}
$$

Fig. 1. Syntax and evaluation rules of Scheme$_\beta$

Throughout this and the next section we will use a synthetic example (Listing 1.3) to demonstrate blame prediction. This function switches its operation (and thus its return type) according to the truth value of the `mode` parameter: it either adds one to a number, or prepends a string with "Hello". Note that this functions happens to be in ANF already.

```
(define (inc-or-greet mode y)
  (if mode
    (+ y 1)
    (string-append "Hello " y)))
```

Listing 1.3. Running example

Consider the expression `(inc-or-greet #t (read))`. The `read` primitive reads input from the user and passes it to the `inc-or-greet` function, along with the boolean value `#t`. Inside this function, the true branch is taken and the `+` primitive is applied to y. If y is not a number, the interpreter raises an error along the lines of "non-numeric value passed to `+`: y".

One observation is that despite the branch inside `inc-or-greet`, the type of the `y` parameter must be `int` or `string`. If this is not the case, the invocation of `inc-or-greet` *always* produces an error. Since the type tests for `+` and `string-append` only happen right before they are invoked, users of `inc-or-greet` receive a type error too late. Therefore, the program can perform a check (see Listing 1.4) on `y` upon entering the function and error out if the test returns false:

```
(define (inc-or-greet mode y)
  (check (or (string? y) (number? y))
    (if mode
      (+ y 1)
      (string-append "Hello " y))))
```

Listing 1.4. `inc-or-greet`, transformed

This `check` macro raises an error if its first argument is $\#f$, otherwise it is equivalent to its second parameter. Note that there is no check on `mode`, as the if-expression in Scheme$_\beta$ accepts any type of value.

A second observation is that the return type of `inc-or-greet` depends on 1) the path taken through the function (based on the runtime value of `mode`); and 2) the type of the `y` parameter.While the first cannot be predicted, an analysis can deduce that *if* `y` is of type `int` or of type `string` *and* the evaluation of the body does not result in an error, its return type is the same as that of `y`. We can write down the type of `inc-or-greet` as a disjunction of two *conditional* types, where $(\tau \sim \mathbf{var}) \cdot \tau_r$ means "if **var** has type τ, the result type is τ_r".

$$\texttt{inc-or-greet}(\mathbf{mode},\ \mathbf{y}) :: ((\mathsf{int} \sim \mathbf{y}) \cdot \mathsf{int}) \vee ((\mathsf{string} \sim \mathbf{y}) \cdot \mathsf{string})$$

This formulation allows *users* of `inc-or-greet` to check types up front:

```
(let ((input (read)))
  (check (or (string? input) (number? input))
         ... other code here ...
    (inc-or-greet #t input)))
```

Listing 1.5. transformed application of `inc-or-greet`

4 The Blame Prediction Transformation

In this section we describe the transformation that enables blame prediction. This transformation makes the primitive type tests performed by the runtime system explicit, and aims to perform them as soon as possible. This transformation has two important properties:

Blame May Only Be Signaled If All Possible Paths Result in an Error. This property forms the main distinction between blame prediction and a type system: a type system performs all its type tests at verification time and rejects (parts of) programs if they *might* cause a runtime error. By

contrast, blame prediction allows programs to run up until the point where all execution paths *must* result in an error. For example, upon entering `(inc-or-greet #t "hi")`, some paths could still succeed so no blame is predicted. Once execution reaches the true branch of the if-expression, blame is predicted as all possible paths (namely `(+ y 1)`) result in an error. This property ensures that blame-predicted programs only raise errors if their original versions do.

Blame Prediction May Resolve Type Tests Statically.
Blame predicted programs are allowed to elide type tests when the inferred type of a variable is exactly the requested type. Likewise, blame prediction is allowed to replace expressions with type tests that will *always* fail by a static error message. For example, `(inc-or-greet #t #t)` will always fail with a type error, so blame can be predicted without entering the function.

The blame prediction transformation has three stages: type inference (subsection 4.1), insertion of type tests (subsection 4.2), and moving type tests upwards (subsection 4.3). The rest of this section assumes that the input is free of variable shadowing.

4.1 Type System

The transformation first analyzes the program and associates each expression with a type. Such a type encodes a primitive type like other type systems do, but also records all type tests that lead up to that expression. The types returned by this analysis are listed in Figure 2. Union types combine the types returned by the branches of if-expressions. The most unconventional part of this type system are the conditional types $(\tau_1 \sim \tau_2)^{l_b}_{l_c} \cdot \tau_t$: they represent a value that only exists if τ_1 is equivalent to τ_2. Note that the τ_1 will always be either a ground type (int or string) or a function type.

The blame label l_b and cause label l_c respectively identify the source locations of a function application and one of its operands. At a later step in the transformation, type tests will be performed earlier in the program, predicting blame to the expression at l_b if the tests fail. The cause label l_c associates the operand to be tested with the conditional type. The Label and Expr functions map expressions to labels and vice versa.

There are also function types that represent the different paths through a function along with the type tests they make and their return types. Their arguments are represented by type variables, which get embedded in type tests and return types. Applying these functions yields a type-level application, which substitutes argument types for type variables. If the function type being applied is a type variable, the type-level application is left unresolved until the function type is known. We will point out how each kind of type can be generated by the type rules, shown in Figure 3.

- Rules T-CONST and T-VAR are defined as usual.
- Rule T-IF enables if-expressions to combine the types of both branches using a union type.

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{int} \mid \mathsf{string} & \text{ground types} \\
 & | & \tau \vee \tau & \text{union types} \\
 & | & \alpha & \text{type variable} \\
 & | & \Pi \alpha_{1\ldots n}.\tau & \text{function type} \\
 & | & (\alpha\ \tau_1 \ldots \tau_n) & \text{function type application} \\
 & | & (\tau \sim \tau)^l_l \cdot \tau & \text{conditional type} \\
\alpha, \beta & \subseteq & \text{TVar} & \\
l & \in & Lab & \\
Label & : & Exp \mapsto Lab & \text{labels of expressions} \\
Expr & : & Lab \mapsto Exp & \text{expressions at labels}
\end{array}
$$

Fig. 2. Types

- Rule T-LET is different from normal type systems because it needs to ensure that the type of the whole let-expression is guarded by the type tests made by e_x[1]. For example, the type of `(let (x (+ y 1)) #t)` should record the fact that `y` is used as a number, even though it is not used in the let body. In order to record this, this type rule uses the Leaves helper function to extract the types at the leaves (τ_L) of e_x. The rationale behind this is that all type tests in e_x have happened by the time the body is evaluated. The let body is then inferred with the x bound to the type leaves τ_L, yielding a body type τ. Finally, the type of the let-expression is attached to the type tests in the type of e_x using the Chain function. This preserving the type tests made by e_x in the larger let type.
- Rule T-LAMBDA infers the type of the body with the arguments bound to type variables $\alpha_1 \ldots \alpha_n$. This type is then wrapped in a type function with these type variables as arguments. Any type tests performed on the arguments are recorded in the type of the body as well. When this type function is eventually applied (see rule T-APPLY below), the type tests are propagated to the application site and can be eliminated or moved up.
- Rule T-APPLY analyzes function application. This rule infers the type of the called function and its arguments and tries to apply the type function to the arguments using the Apply type function discussed in the next section. This rule does not use the Leaves and Chain functions of T-LET, as the entire type of the function application is returned.
  For bookkeeping purposes, each type τ_i passed into the Apply function is associated with the label of its expression l_{τ_i}. The label of the function application itself is l_f. This enables the type tests generated by Apply to assign blame to the correct part of the program if they fail.

The Apply function. The various cases of the Apply function are considered from top to bottom, and the first eligible case is executed. In the simplest case, the Apply function is invoked with a τ_f that is statically known to be a function. The return type is then constructed by substituting the actual types $\tau_1 \ldots \tau_n$

[1] Note that the equality between (let $(x\ e_x)\ e$) and ((lambda $(x)\ e)\ e_x$) holds because the latter must always be ANF-converted into the former.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{c \in \{\#t, \#f, \mathsf{n}\}}{\Gamma \vdash c : \text{Typeof}(c)} \quad \text{(T-CONST)} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{if}\ s\ e_1\ e_2) : \tau_1 \vee \tau_2} \quad \text{(T-IF)}$$

$$\frac{\Gamma \vdash e_x : \tau_1 \qquad \Gamma, x : \tau_L \vdash e : \tau \qquad \tau_L = \text{Leaves}(\tau_1)}{\Gamma \vdash (\mathsf{let}\ (x\ e_x)\ e) : \text{Chain}(\tau_1, \tau)} \quad \text{(T-LET)}$$

$$\frac{\Gamma, x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e : \tau}{\Gamma \vdash (\mathsf{lambda}\ (x_1 \ldots x_n)\ e) : \Pi\alpha_{1\ldots n}.\tau} \quad \text{(T-LAMBDA)}$$

$$\frac{\Gamma \vdash s_f : \tau_f \qquad \Gamma \vdash s_i : \tau_i \quad \forall i \in [1 \ldots n] \qquad l_f = \text{Label}((s_f\ s_1 \ldots s_n))}{\Gamma \vdash (s_f\ s_1 \ldots s_n) : \text{Apply}(\tau_f, l_f, \tau_1 \ldots \tau_n)} \quad \text{(T-APPLY)}$$

Auxiliary definitions

$$\begin{array}{lll}
\text{Apply}(\Pi\alpha_{1\ldots m}.\tau_f, l_f, \tau_1 \ldots \tau_n) & = \tau_f[\tau_1 \ldots \tau_n/\alpha_1 \ldots \alpha_m][l_{\tau_1} \ldots l_{\tau_m}/l_{\alpha_1} \ldots l_{\alpha_m}] & \text{if } m = n \\
\text{Apply}(\Pi\alpha_{1\ldots m}.\tau_f, l_f, \tau_1 \ldots \tau_n) & = \mathsf{error}(l_f) & \text{if } m \neq n \\
\text{Apply}(\tau_\alpha \vee \tau_\beta, l_f, \tau_1 \ldots \tau_n) & = \text{NoError}(\text{Apply}(\tau_\alpha, l_f, \tau_1 \ldots \tau_n),\ \text{Apply}(\tau_\beta, l_f, \tau_1 \ldots \tau_n)) & \\
\text{Apply}(\alpha, l_f, \tau_1 \ldots \tau_n) & = ((\Pi\alpha_{1\ldots n}._) \sim \alpha)^{l_f}_{l_\alpha} \cdot (\alpha\ \tau_1 \ldots \tau_n) & \\
\text{Apply}(\tau, l_f, \tau_1 \ldots \tau_n) & = \mathsf{error}(l_f) & \\
 & & \\
\text{Chain}(\tau_1 \vee \tau_2, \tau_c) & = \text{Chain}(\tau_1, \tau_c) \vee \text{Chain}(\tau_2, \tau_c) & \\
\text{Chain}((\tau_t \sim \tau_1)^{l_b}_{l_c} \cdot \tau, \tau_c) & = (\tau_t \sim \tau_1)^{l_b}_{l_c} \cdot \text{Chain}(\tau, \tau_c) & \\
\text{Chain}(\tau, \tau_c) & = \tau_c & \\
 & & \\
\text{Leaves}(\tau_1 \vee \tau_2) & = \text{Leaves}(\tau_1) \vee \text{Leaves}(\tau_2) & \\
\text{Leaves}((\tau_t \sim \tau) \cdot \tau_1) & = \text{Leaves}(\tau_1) & \\
\text{Leaves}(\tau) & = \tau & \\
 & & \\
\text{NoError}(\mathsf{error}(l_f), \tau_\beta) & = \tau_\beta & \\
\text{NoError}(\tau_\alpha, \mathsf{error}(l_f)) & = \tau_\alpha & \\
\text{NoError}(\tau_\alpha, \tau_\beta) & = \tau_\alpha \vee \tau_\beta &
\end{array}$$

Fig. 3. Blame prediction: type inference rules and auxiliary definitions

for the type variables $\alpha_1 \ldots \alpha_n$ in the function type. Additionally, the labels attached to these types are replaced by the labels of the arguments. Below are some examples of Apply:

$$\begin{aligned}
\text{Apply}(\Pi\alpha.((\mathsf{string} \sim \alpha)^{-}_{l_\alpha} \cdot \mathsf{string}), l_f, \mathsf{string}) &= ((\mathsf{string} \sim \mathsf{string})^{l_f}_{l_{string}} \cdot \mathsf{string}) && (1)\\
\text{Apply}(\mathsf{string} \vee \Pi\alpha.\alpha, l_f, \mathsf{int}) &= \text{NoError}(\mathsf{error}(l_f),\ \mathsf{int}) = \mathsf{int} && (2)\\
\text{Apply}(\alpha, l_f, \mathsf{boolean}, (\mathsf{string} \vee \mathsf{int})) &= ((\Pi\alpha_1, \alpha_2._) \sim \alpha)^{l_f}_{l_\alpha} \cdot \\
&\qquad (\alpha\ \mathsf{boolean}\ (\mathsf{string} \vee \mathsf{int})) && (3)
\end{aligned}$$

If there are too few or too many arguments given to a a function, Apply returns an error type. If the first argument to Apply is a union type, Apply is called recursively while preserving the original structure of the type. This can result in invalid function applications — for example in example 2 above — which are replaced with error types. After invoking Apply on a union type, only the non-error parts are retained by the NoError function. If all parts of a union type result in error, the entire type is error. Normal type systems would reject such programs immediately, but blame prediction must continue as such an expression might be buried in a function that is never called or only on some paths. Expressions that are assigned the type error will be guarded by a runtime test that always fails.

Finally, a type variable might be the type function being applied to several type arguments. This can happen as a result of the user creating higher-order functions, which are common in functional languages. The type of such an application is a *type-level* function application, as in example 3 above. Type-level function applications remain in the type until their type variable is replaced by a concrete type, at which point Apply is called anew. This mechanism enables blame prediction to deal with higher-order functions.

After assigning types to all expressions of the program, a round of simplification is performed on the types:

- Elimination of trivially true type tests such as $\mathsf{string} \sim \mathsf{string}$ in example 1;
- Elimination of repeated type tests :
  $((\mathsf{string} \sim n) \cdot (\mathsf{int} \vee (\mathsf{string} \sim n) \cdot \mathsf{string})$ becomes $(\mathsf{string} \sim n) \cdot (\mathsf{int} \vee \mathsf{string}))$
- Merging of union types with equivalent branches ($\mathsf{int} \vee \mathsf{int}$ becomes just int);

4.2 Separation of Code and Type Tests

After inferring types for the program, blame prediction then makes type tests explicit by separating type tests from applications of both primitive- and user-defined functions. Every type test is annotated with two labels: a *blame label* that references the function whose preconditions are being checked, and a *cause label* that points to the expression being tested. For example, in a type test for the expression `(f x)`, the blame label points to the definition of `f` and the cause label points to `x`.

Figure 4 extends the syntax presented earlier with a check construct, which evaluates its second argument if the checks in the first argument are true. As a

$$
\begin{array}{rcll}
\mathrm{e} \in \mathrm{Exp} & ::= & \ldots & \text{as in Figure 1} \\
 & & (\mathsf{check}\ \mathsf{C}\ e) & \text{check expression} \\
\mathsf{C} & ::= & (\tau?\ {}^{x}\!/_{c})^{\mathsf{L}} & \text{type test} \\
 & & \#t & \text{empty test} \\
 & & \mathsf{C} \vee \mathsf{C} & \text{disjunction} \\
 & & \mathsf{C} \wedge \mathsf{C} & \text{conjunction} \\
 & & \mathsf{blame}(\mathsf{l}) & \text{static blame} \\
{}^{x}\!/_{c} & ::= & x \mid c & \\
\mathsf{L} & \subset & \mathit{Lab} &
\end{array}
$$

Shorthand notation

$$[\mathsf{C}]e \equiv (\mathsf{check}\ \mathsf{C}\ e)$$

Fig. 4. Expression syntax with explicit checks

shorthand, we write $[\mathsf{C}]e$ for $(\mathsf{check}\ \mathsf{C}\ e)$. Checks can either be a type test on a variable or constant ${}^{x}\!/_{c}$ with a set of blame labels L, the "always-true" test $\#t$ and a conjunction or disjunction of checks. When an expression has type error, a corresponding "static blame" check is generated which always assigns blame if it is reached. The residual type at each application site is converted to a check C as follows:

$$
\begin{array}{ll}
\mathrm{convert}((\tau \sim \_)^{l_b}_{l_c} \cdot \tau_f) & = (\tau?\ s_c)^{\{l_b\}} \wedge \mathrm{convert}(\tau_f) \quad (\text{where } (s_c) = \mathrm{Expr}(l_c)) \\
\mathrm{convert}(\tau_1 \vee \tau_2) & = \mathrm{convert}(\tau_1) \vee \mathrm{convert}(\tau_2) \\
\mathrm{convert}(\mathsf{error}(\mathsf{l})) & = \mathsf{blame}(\mathsf{l}) \\
\mathrm{convert}(\tau) & = \#t
\end{array}
$$

Other elements of the syntax tree receive the check $\#t$. Remember from subsection 4.1 that each type test is associated with a cause label l_c and a blame label l_b, both referring to source locations. To construct an actual check, the cause label l_c is looked up using Expr — the inverse of Label — yields the expression $\mathsf{e_c}$ the label refers to. This expression is always a variable or a constant, as cause labels point to the simple arguments of function applications. The blame label l_b becomes the sole element of the blame labels associated with the type test.

After conversion, every check is simplified according to the normal logic formulas for conjunctions and disjunctions: $\#t$ is removed from conjunctions, and disjunctions with $\#t$ in one of the branches are entirely replaced by $\#t$.
After type test introduction, the `inc-or-greet` example becomes as shown in Listing 1.6.

```
(define (inc-or-greet mode y)
  (if mode
      [(int? y)^{l+}] (+ y 1)
      [(string? y)^{l_string-append}]
    (string-append "Hello " y)))
```

Listing 1.6. `inc-or-greet` example after separating code from type tests

4.3 Moving Type Tests Upwards

The last step of the blame prediction transformation attempts to move the `check` nodes as far up the evaluation tree as possible. This process (called "flotation") is applied to the expression tree in a bottom-up fashion. Flotation rules are of the form $e \mapsto e' \uparrow \mathsf{C}$, meaning that expression e can be rewritten to expression e', floating check C upwards. Checks cannot float past the top of the program. The flotation rules — listed in Figure 5 — are described as follows:

- Rules F-CONST and F-VAR are for completeness: as we only introduce `check` nodes at function application sites, constants and variables will never give rise to a type test.
- The F-APPLY rule simply floats its checks upwards.
- Rule F-IF floats the disjunction of the checks performed by both nodes, while still performing these checks in the branches themselves. This enables the `inc-or-greet` example to predict blame if `y` variable contains something not of type `int` or `string`.
- Rule F-LET ensures that let-expressions only float checks upwards which do not involve the bound variable. Any checks that *do* involve the variable x are replaced by $\#\mathrm{t}$. A copy of the original checks C remains in the let body.
- Rule F-LAMBDA stops checks from floating past lambda-expressions, as these type tests would only be performed when the function is actually applied.

$$c \mapsto c \uparrow \#t \quad \text{(F-CONST)} \qquad x \mapsto x \uparrow \#t \quad \text{(F-VAR)}$$

$$[\mathsf{C}](s\ s_1 \ldots s_n) \mapsto (s\ s_1 \ldots s_n) \uparrow \mathsf{C} \quad \text{(F-APPLY)}$$

$$\frac{e_1 \mapsto e_1' \uparrow \mathsf{C}_1 \qquad e_2 \mapsto e_2' \uparrow \mathsf{C}_2}{(\mathsf{if}\ s\ e_1\ e_2) \mapsto (\mathsf{if}\ s\ [\mathsf{C}_1]e_1'\ [\mathsf{C}_2]e_2') \uparrow \mathsf{C}_1 \vee \mathsf{C}_2} \quad \text{(F-IF)}$$

$$\frac{e_x \mapsto e_x' \uparrow \mathsf{C}_x \qquad e \mapsto e' \uparrow \mathsf{C} \qquad \mathsf{C}' = [\#t/(\tau?\ x)]\mathsf{C}}{(\mathsf{let}\ (x\ e_x)\ e) \mapsto (\mathsf{let}\ (x\ e_x')\ [\mathsf{C}]e') \uparrow \mathsf{C}_x \wedge \mathsf{C}'} \quad \text{(F-LET)}$$

$$\frac{e \mapsto e' \uparrow \mathsf{C}}{(\mathsf{lambda}\ (x_1 \ldots x_n)\ e) \mapsto (\mathsf{lambda}\ (x_1 \ldots x_n)\ [\mathsf{C}]e') \uparrow \#t} \quad \text{(F-LAMBDA)}$$

Fig. 5. Rules for floating checks up the evaluation tree

Floating type tests in the `inc-or-greet` example finally yields the following program, which is what we wanted to accomplish in section 3.

```
(define (inc-or-greet mode y)
  (check (or (int? y)^{l+} (string? y)^{l_string-append})
    (if mode
      (check (int? y)^{l+} (+ y 1))
      (check (string? y)^{l_string-append}
        (string-append "Hello " y)))))
```

Listing 1.7. `inc-or-greet` example after floating type tests up

The algorithm as presented generates a lot of redundant tests: in general, *each use* of a variable results in a type test regardless of whether it has been performed before. Therefore, a number of simplifications are performed:

1. Repeated tests in the same precondition are merged into the first top-level occurrence of that test. Their blame labels are added to the first test's blame labels and the duplicate tests are replaced by $\#t$. Finally, the preconditions are simplified.
2. Preconditions of the form $\#t \vee \mathsf{C}$ or $\mathsf{C} \vee \#t$ are replaced by $\#t$. These kinds of tests are formed when one branch of an `if` expression returns a constant or variable and the other branch is more complex. They do not contribute anything to the blame prediction as they are always true.
3. Finally, the program is walked from top to bottom in order to remove repeated tests on the same variable across nested `check` expressions. As with the first step, the blame labels of the lower tests are merged with those of the tests higher up.

Remember from subsection 4.1 that trivially true type tests (e.g. `int? 5`) are eliminated as part of the simplification, while leaving alone type tests that always fail (e.g. `int? "hello"`). These last type tests are introduced and floated upwards together with the other constraints. It is tempting to introduce a simplification that replaces these expressions by $\#f$ and subsequently prunes preconditions, but this causes misleading predictions.

Applying the full blame prediction transformation to the `guess` example from the introduction results in Listing 1.8.

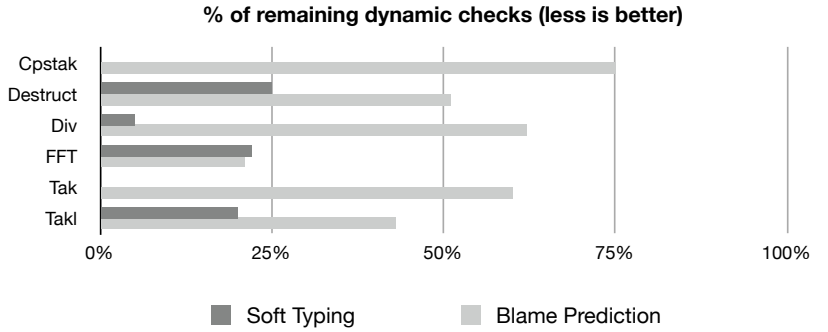
```
(define (guess target) ; type is Πα.symbol ∨ ((int ∼ α) · symbol)
  (let ((input (read)))
    (if (eq? input 'quit)
        'quit
        (check
          (and (number? input)^{<,>} (number? target)^{<,>})
          (if (< input target)
              (begin (display "Too low!\n")
                     (guess target))
              (if (> input target)
                  (begin (display "Too high!\n")
                         (guess target))
                  (begin (display "Correct!\n")
                         'done)))))))
```

Listing 1.8. `guess` after blame prediction

5 Evaluation

To support our claim of moving type tests up, we have compared the output of soft typing [9] to that of blame prediction for selected examples from the Gabriel benchmarks [10]. This comparison is done along two dimensions (Figure 6):

1. The number of dynamic type tests remaining in the program, with the untransformed program as baseline (100%).
2. The "computational distance" between a type test and the primitive application that needs it, counted as steps in the spine of the ANF-transformed program. Since variables can be tested multiple times, we show the minimum, median and maximum distance encountered for select variables in the program.



Name	Computational distance (min/med/max)
Cpstak	y(2/3/8), x(2/3/5), z(1/1/3), k(0/0/0)
Destruct	l1(3/6/10), l(3/5/13), l2(4/5/7), a(1/4/7), b(3/4/6)
Div	l(1/4/6), n(1/2/7), i(1/2/6)
FFT	ar(19/56/60), i(1/10/13), ai(3/7/11), l(1/5/6), j(1/2/6)
Tak	z(4/4/5), x(2/3/5), y(2/3/6), n(1/1/3)
Takl	z(4/4/5), n(2/3/5), y(2/3/4), x(1/2/3)

Fig. 6. Comparison of blame prediction to soft typing with respect to type tests

As the graph in Figure 6 shows, soft typing is a more advanced type system that can eliminate type tests statically. However, the basic type system presented in subsection 4.1 is also capable of removing about forty percent of the type tests in an average program. A majority of the remaining type tests can be attributed to lack of precision when dealing with data structures: in the fft example, just over half of the remaining type tests verify the types of vector elements. Of the tests that remain most can be moved up three to four levels, which already makes a difference in these small programs.
Our implementation can be downloaded from `https://github.com/botje/crystal`; blame prediction can be tried online at `http://bit.ly/blame-sandbox`.

6 Related Work

As mentioned in the introduction, there exists a huge body of work [4,5] on ever more expressive type systems for dynamically typed programming languages, with the ultimate goal of being able to statically type all dynamic programs.

These approaches limit themselves to a subset of the language they are studying, resulting in an inability to type whole classes of programs that will never raise a runtime error. The outcome of this body of work can be used to improve the static capabilities of blame prediction, at least for programs that can be typed successfully.

Soft typing [9] was among the first attempts at inserting type tests into dynamically typed programs. Their type system featured guarded primitives like `CHECK-car` and also marked subexpressions as "will always fail". The primary motivation for inserting these type tests is to allow type inference to proceed, but errors are only reported if faulty expressions are evaluated. The type system of soft typing is much more powerful than ours: it supports arbitrary recursive types and negative types to represent information about the absence of types in a union type. However, blame prediction floats the remaining type tests up as much as possible, reporting errors earlier.

Recently, gradual typing [11] has acknowledged that programmers may want to gradually convert their programs to static typing. In the cases where normal type systems cannot reason over the entire program, gradual typing can be used to statically type check parts of the program. Parts that cannot be typechecked are assigned the unknown type $\star$ and interactions with statically typed code is guarded by a type conversion such as $\langle x \Leftarrow \tau \rangle$. Gradual typing was a big inspiration for this research, with the observation that the type tests performed by a gradually typed program can be performed earlier in the control flow. A similar idea was raised by [12], which uses contracts to reconcile typed and untyped code. We plan to build a more elaborate version of blame prediction on top of gradual and/or soft typing, equipping these type systems with better error prediction.

Contract systems are closely related to blame prediction. In higher-order contracts [13] it is often the case that when a contract violation is detected at runtime, the point in the program where the violation is detected (e.g. line number) is not related to the point in the program that caused the problem. Blame assignment [13] helps the programmer in relating the point where the error is detected to its cause. The big difference to blame prediction is that contract systems point out errors *back in time*, while blame prediction assigns blame *ahead of time*.

Aside from primitive operations, explicit type tests can also be used to deduce type information in a dynamically typed language. The work in [4] sidesteps the "one variable, one type" restriction that is present in many type systems, instead opting to make types flow-sensitive. We plan to incorporate manual type tests in later versions of blame prediction to better estimate types, thus eliminating more type tests and floating tests up even further.

A recent addition to the Glasgow Haskell Compiler has lead to deferred type errors [14]. Rather than halting compilation when a type error is discovered, the program is compiled as normal but the wrongly-typed expressions are replaced by "holes". When such a hole is accessed by the runtime system, an error is raised

as before. These errors are only raised as they are accessed however, even if the compiler can predict that the hole must be entered.

7 Conclusion

In this paper, we presented a novel typing discipline called blame prediction. The key insight of this research is that dynamically typed programs perform type tests on expressions only at the call site of primitive operations, while these tests could be performed considerably earlier. We have developed this insight into a typing discipline that makes type tests an integral part of expressions' types. These types are used to steer a program transformation that makes type tests explicit in code. The end result is a program that performs dynamic type tests well before they can raise errors, with pointers to the failing expression and the code that needs it. This in turn helps developers debug their applications faster.

Acknowledgements. The authors would like to thank Matthias Felleisen for input on an early draft of this paper. We are also grateful to the anonymous reviewers for input on the second draft.

References

1. Cartwright, R., Fagan, M.: Soft typing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 278–292 (1991)
2. Aiken, A., Wimmers, E.L., Lakshman, T.K.: Soft typing with conditional types. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 163–173 (1994)
3. Flanagan, C., Felleisen, M.: A new way of debugging lisp programs. 40th Anniversary of Lisp (Lisp in the Mainstream) (1998)
4. Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing Local Control and State Using Flow Analysis. In: Proceedings of the 20th European Symposium on Programming, pp. 256–275 (2011)
5. Furr, M., An, J., Foster, J., Hicks, M.: Static type inference for Ruby. In: Proceedings of the ACM Symposium on Applied Computing, pp. 1859–1866 (2009)
6. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science 103(2), 235–271 (1992)
7. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. LISP and Symbolic Computation 6(3-4), 289–360 (1993)
8. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 237–247 (1993)
9. Wright, A.K., Cartwright, R.: A practical soft type system for Scheme. In: Proceedings of the ACM Conference on LISP and Functional Programming, pp. 250–262 (July 1994)
10. Gabriel, R.P., Masinter, L.M.: Performance of Lisp systems. In: Proceedings of the ACM Symposium on LISP and Functional Programming, pp. 123–142 (1982)

11. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Proceedings of the Workshop on Scheme and Functional Programming, pp. 81–92 (2006)
12. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: Proceedings of the 21st ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 964–974 (2006)
13. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the 7th International Conference on Functional Programming, pp. 48–59 (2002)
14. Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Equality proofs and deferred type errors: A compiler pearl. In: Proceedings of the 17th International Conference on Functional Programming, pp. 341–352 (2012)

Model-Based Shrinking for State-Based Testing

Pieter Koopman, Peter Achten, and Rinus Plasmeijer

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
{pieter,p.achten,rinus}@cs.ru.nl

Abstract. Issues found in model-based testing of state-based systems are traces produced by the system under test that are not allowed by the model used as specification. It is usually easier to determine the error behind the reported issue when there is a short trace revealing the issue. Our model-based test system treats the system under test as a black box. Hence the test system cannot use internal information from the system under test to find short traces. This paper shows how the model can be used to systematically search for shorter traces producing an issue based on some trace revealing an issue.

1 Introduction

In model-based testing of state-based systems there is a model that specifies the allowed transitions between states. Each transition in an extended state machine is labeled with an input and the corresponding output. The conformance relation restricts the allowed outputs by the system under test to output covered by the model for the inputs specified for each reachable state [9, 11].

In a state-based system the reaction of the system under test and the model on some input depends on the current state and hence on the history. The list of previous transitions, the trace, determines the current state. When the observed output for some transition of the system under test is not covered by the model we have determined nonconformance. In the testing jargon this is called an *issue*. In an ideal test world such an issue indicates an error in the system under test. In the real world issues can also indicate errors in the model or interfacing problems between the system under test and the test system.

In simple cases the error indicated by a discovered issue is obvious. This happens for instance when it is clear that the output that is generated by the system under test is not allowed in the reached state. Often it is less obvious what caused the illegal transition. In such situations we have to take the trace in consideration since it determines how we arrived in the current state. It is obvious that analysing such issues is in general much easier when we have a short trace indicating the issue.

In an unguided test situation the test system has no idea where to search potential nonconformance and takes transitions in a pseudorandom order. This can result in fairly long traces of several thousands of transitions. In this paper we discuss strategies for finding smaller traces indicating an issue based on such

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 107–124, 2014.

a long trace. The technique to find smaller counterexamples based on an already found counterexample in model-based testing is called *shrinking*.

The shrinking algorithm used for traces showing an issue acknowledges that the state of the system under test can be quite different from the state of the model. Nevertheless, similar states for the model can correspond to similar states in the system under test. The contribution of this paper is that it shows how to generate candidate traces to be tested based on the trace that is known to reveal an issue. We introduce effective ways to generate test suites by eliminating individual transitions from the trace and by eliminating all transitions corresponding to a cycle in the model.

2 Conformance

A trace σ is a sequence of inputs and associated outputs from a given state. The empty trace, ϵ, connects a state to itself: $s \stackrel{\epsilon}{\Rightarrow} s$. We write $s \stackrel{\sigma}{\Rightarrow}$ to indicate $\exists u \,.\, s \stackrel{\sigma}{\Rightarrow} u$ and $s \xrightarrow{i/o} \equiv \exists u \,.\, s \xrightarrow{i/o} u$. Often we are especially interested in the inputs to be supplied to the state machine and only list the inputs of the trace rather than the complete trace. For deterministic systems the rest of the trace can always be deduced.

The inputs for which a transition is possible in state s are given by the set $\mathsf{init}(s) \equiv \{\, i \mid \exists\, o \,.\, s \xrightarrow{i/o} \}$. The states after applying trace σ in state s are given by $s\,\mathsf{after}\,\sigma \equiv \{\, t \mid s \stackrel{\sigma}{\Rightarrow} t \}$. The traces starting in state s are given by $\mathsf{traces}(s) = \{\, \sigma \mid s \stackrel{\sigma}{\Rightarrow} \}$. Operations like init and after are overloaded for sets of states. Note that there are infinitely many traces and an infinitely long trace if the state machine contains a loop, that is $\exists\ s, \sigma \,.\, s \stackrel{\sigma}{\Rightarrow} s$, even if the machine is a finite state machine.

For *conformance* between a model and a system under test sut the observed output of the sut should be allowed by the model for each input i in the init after every trace σ. Formally, conformance of the sut to the specification model is defined as:

$$\mathsf{sut}\ \mathit{conf}\ \mathsf{model} \equiv \forall \sigma \in \mathsf{traces}_{\mathsf{model}}(s_0).\forall i \in \mathsf{init}(s_0\,\mathsf{after}_{\mathsf{model}}\,\sigma).\forall o \in O.$$
$$(t_0\,\mathsf{after}_{\mathsf{sut}}\,\sigma) \xrightarrow{i/o} \Rightarrow\ (s_0\,\mathsf{after}_{\mathsf{model}}\,\sigma) \xrightarrow{i/o}$$

Here s_0 is the initial state of the specification and t_0 is the initial state of the system under test. Note that this conformance relation compares inputs and outputs. The actual states of the system under test are never used, hence the system under test is treated as a black box. Comparing inputs and outputs of the model and the system under test implies that they have to use identical types of inputs and outputs. The states however, can be completely different.

The conformance relation states that the system under test should accept at least the traces allowed by the specification. In particular the system under test should accept an input if the specification states that it is allowed after the observed sequence of transitions. Many systems under test will be *input*

enabled; the system accepts any input in every state. This is obvious for most inputs corresponding to physical buttons or messages sent to a system. In a graphical user interface an input can only occur when the corresponding button or text field is visible. Such a system is not input enabled, but the system under test should accept any input that is allowed by the specification.

2.1 Testing Conformance

Testing state-based machines is built on the conformance relation introduced above. Model-based testing checks the conformance relation for a finite number of finite traces. Since there can be infinitely many or infinitely long traces, testing conformance is in general an approximation of the relation *conf*.

Instead of generating traces of the specification and verifying whether they are accepted by the system under test, the test algorithm of our model-based test tool G∀st, testConf, maintains the after states of the current trace [9, 11]. It determines for n transitions on a single trace *on-the-fly* whether the observed behaviour of the system under test conforms to the model [13, 14]. If there are no after states testing has determined nonconformance. If the remaining number of steps to do is zero, testing this trace is terminated without any conformance problems. Testing a trace is also finished if there are no transitions possible from the current state; the init for the current state is empty. Otherwise, we choose an arbitrary input that is accepted by the specification in one of the current states, apply it to the system under test and observe the corresponding output. The new set of states for the specification is the set of states that is obtained after the transition. The algorithm continues testing with one step less and the new set of states. Expressed in the lazy functional programming language Clean [12] this becomes:

```
:: Verdict      // possible test results
   = Pass       // non problem in the current trace
   | Fail       // nonconformance found
   | Truncate   // testing the given trace is interrupted

testConfRandom :: Int [s] t → Verdict
testConfRandom n [] t = Fail     // no current state: issue found
testConfRandom 0 ss t = Pass     // counter for transitions to do is 0
testConfRandom n ss t
   | isEmpty inputs = Pass       // init is empty: no transition from this state
   | otherwise      = testConfRandom (n–1) (after ss i o) t2
where inputs = init ss
      i      = elemFrom inputs   // input selection
      (o,t2) = sut t i           // output and new state of sut
```

In this definition s is the type of the state of the specification and t is the state of the system under test.

In order to test conformance we evaluate testConfRandom N [s_0] t_0 for M traces. When there is a *test goal* (some specific behaviour) we use this to select

the input i from the current init, otherwise the test system selects a pseudorandom element. The real implementation of this algorithm in G∀st is more involved since it is parameterized by the system under test and model, it collects the trace, guides input selection, allows pseudorandom input selection, etc. When a trace is found that proves nonconformance, the trace is shown together with the intermediate model states on the trace. The system under test is treated as a black box, hence its states cannot be shown.

There is another mode of conformance testing that is relevant for this paper. Here we have a given list of inputs (instead of generating it on-the-fly) and check whether there is conformance of system under test and the model for this sequence of inputs. If the input at some point is not part of the init of the model at some point testing this sequence is truncated; there is no issue found, but we can neither continue the conformance check.

```
testConfGiven :: [I] [S] T → Verdict
testConfGiven n [] t = Fail
testConfGiven [] ss t = Pass
testConfGiven [i:r] ss t
    | isMember i inputs = testConfGiven r (after ss i o) t2
    | otherwise         = Truncate
where inputs = init ss
      (o,t2) = sut t i
```

In order to use model-based testing for abstract data types with a state machine as specification, the system under test must behave as a state machine. For the system under test we construct a very simple machine that stores the actual abstract data type as its state. For the model we use a state machine with a state that contains enough information to check the transitions. Such a state is typically a naïve implementation of the interface offered by the abstract data type, or an abstraction of it.

2.2 Models for G∀st

For our model-based test tool G∀st a specification is just a function taking the current state and input as arguments and yields a list of allowed transitions. In a deterministic system the list of allowed transitions for each reachable state and input combination is a singleton. When this list is empty for some reachable state and input combinations we have reached the end of a trace. When there are multiple transitions possible for one reachable state and input, the model is nondeterministic; all listed transitions are allowed.

A transition is either a pair transition, Pt, containing a list of outputs and a new state, or a function transition, Ft, that contains a function that determines the target states based on the given list of outputs from the system under test.

```
:: Spec state input output :== state input → [Trans output state]
:: Trans output state = Pt [output] state | Ft ([output]→[state])
```

The Ft construct is particularly useful when many outputs are allowed. For instance, a specification that allows any output containing a single element as reaction on the input Go is specified by the function model:

```
model :: State Input → [Trans Output State]
model s Go = [Ft singleton]
where
    singleton [o] = [{state & output = o}]
    singleton out = []
model s i = ...
```

Section 6.1 contains a complete specification of a simple vending machine.

3 The Desire for Small Traces

Theoretically all traces showing nonconformance are equally good to falsify conformance. In practice however, small traces are highly preferable since directly after spotting nonconformance one starts investigating the source of this issue.

Since G∀st immediately stops testing when nonconformance is observed, it is always the last transition of the trace showing the issue. In rare situations it is obvious that the observed combination of input and output is incorrect and we only have to consider the last transition of the sequence showing the issue. In most situations the output is correct in the current state of the system under test, but the system is in another state than it is supposed to be. This implies that in some previous transitions the model produced an approved output, but has gone to the wrong state. Since the system under test is a black box, the test system cannot observe the wrong state. In order to analyse these issues we need to investigate how the system ended up in the state where this issue was observed. For a short trace this is obviously easier than a long one.

Well-known algorithms to find shortest paths in a tree are breadth-first search and iterative deepening depth-first search. These and similar algorithms are used to find minimal counterexamples in model checking [2]. Most model checkers use clever optimisations of these brute force approaches. In model-based testing we are not able to check as thoroughly as in model checking, either because the state space is too big (e.g. infinite due to parameterized states), time constraints do not allow us to check the entire state space, or the system under test is nondeterministic. Nondeterminism in the system under test is either a true random choice of the system, or due to partial knowledge of the state of the system. For instance, a vending machine that delivers an item if it is in stock, or otherwise an 'out of stock' message is completely deterministic. However, when we do not know the amount of goods in stock, the vending machine seems to behave nondeterministically. The vending machine example used in this paper has 10 different inputs in each state. This implies that there are 10^l different paths of length l. Since the state is parameterized by an integer and the product chosen, an exhaustive search of the state space is unfeasible.

Since short traces showing an issue are very desirable and exhaustive search is not feasible we have to rely on heuristics to obtain short traces. Below we discuss some heuristics and measure their effect.

3.1 Assumptions

In the comparison of heuristics for shrinking and measurement of their results we use the following assumptions:

- The system under test is assumed to behave deterministically during the search for smaller traces. This guarantees that each trace that produces an issue will always produce that issue in a new test, and vice versa.
- Testing an actual transition is more time-consuming than some simple calculations to select the next transition to be tested. Testing a transition typically requires the transformation of the input to a suitable form for the interface, calling the system under test as an external program, waiting for the response, and transforming the output to the proper type for the test system. This implies that it is worthwhile to compute smart test traces, instead of just brute force exploration of the search space.
- In this paper the system under test is assumed to be *input enabled*; every input can be given in any state. As a consequence any trace is a valid trace that can be tested. The system under test is usually input enabled since it is hard to prevent that a particular input is given to the system, e.g., by pushing a button or sending a message. The model does not have to be input enabled. One can decide to omit the transition for some state and input pairs deliberately from the model. Such a partial model of the behaviour can be very useful, but is not input enabled. When such an undefined transition occurs during testing the trace is truncated at that spot. For such partial models it is worthwhile to check whether the sequence of inputs generated is a valid trace of the model before the more expensive tests with the system under test are executed. For dynamically generated (on-the-fly) inputs we select an input that happens to be accepted in the current state of the model.

4 Binary Search for Minimal Traces

Until we implemented shrinking we used a form of binary search to find small traces. When testing with a high upper-bound of the trace length yields an issue it is obvious that there is nonconformance. Hence, it is worthwhile to look for a short trace revealing the issue.

When the system finds an issue with a trace of length n, we try to find an issue with length $n/2$ as upper-bound from scratch. If this succeeds we continue with $n/4$ as upper-bound, otherwise we continue with $3n/4$ as upper-bound. By repeating this we can find relatively small traces revealing an issue.

Although this algorithm finds smaller traces showing an issue, it does not scale well in finding minimal traces. When there are c choices for the input in a state and the minimal trace has length l, the chance of finding a minimal trace by pseudorandom input selection is c^{-l}. A complicating factor is that the minimum length l is not known, hence we can only approximate it. The advantage of this approach is that it is very simple to implement, we just have to repeat the tests

with another seed for the pseudorandom generation and the desired upper-bound on the trace length.

In order to measure the effect of this algorithm we have executed tests with 10 erroneous vending machine implementations described in Section 6.2 with 10 different seeds. The average ratio between the longest and shortest trace showing nonconformance is 17. On average the shortest trace showing an issue is a factor of 6 shorter than the average trace for that system under test. This indicates that this approach works quite well. It is definitely worthwhile to apply this algorithm when nothing else is available.

Despite the success of using several random seeds, there are also limitations that makes it worthwhile to develop better algorithms to find short traces showing nonconformance. First of all this algorithm finds the shortest trace possible to reveal the issue only when that trace is extremely short. Typically the limit is only two transitions. The best path found by using different seeds is on average a factor of 6 longer than the shortest path showing nonconformance. These numbers rapidly increase when the systems become more complex and require a longer path to show nonconformance. Moreover, trying various seeds for the pseudo random generation in order to find a short trace showing nonconformance requires many transitions of the system under test. The testing labour can be slightly reduced by using the length of the best trace found until now as upper-bound for the trace length during the tests of new traces. Bigger reductions of the testing job can be achieved by employing more information from the trace found that shows nonconformance.

5 Shrinking

The technique to find smaller traces showing nonconformance based on a previously found counterexample in model-based testing is called shrinking. It is well known from QuickCheck [1,4,6]. Shrinking systematically generates candidate test cases based on a test value that is known to falsify the property at hand. There is a **class** `shrink x :: x → [x]` that generates a list of smaller test values based on the given value of type `x`. Typically there are instances of `shrink` for all data types used in the tests. The default shrinking algorithm for lists is:

```
instance shrink [i] | shrink i where
    shrink :: [i] → [[i]] | shrink i
    shrink []     = []
    shrink [a:x] = [x: [[a:y] \\ y ← shrink x]]     // omit single element
                   ++ [[b:x] \\ b ← shrink a]      // shrink single element
```

This function tries to eliminate the list elements one by one as well as to shrink the list elements themselves. When a smaller counterexample is found, we can try to shrink it again by the same algorithm until it is sufficiently small. There are many successful applications of this algorithm. For example Hritcu et. al. used this successfully to shrink counterexamples for an abstract machine [5].

Our test system G∀st has no shrinking. For the branch of G∀st based on logical properties as model, shrinking is not needed since test cases are generated from

small to large [10]. This implies that the counterexamples found are already minimal. This paper focusses on the other branch of G∀st that uses extended state machines as model.

Since all we need for the trace is a list of inputs, we can use this shrinking algorithm also for traces. The drawback of this algorithm is that it does not use any information from the model. Only when this algorithm is applied multiple times we can use information about the model in the selection of inputs to shrink. The set of traces generated by this implementation of `shrink` removes at most one input element. We would need very many of these shrinking steps to reduce the length of the trace revealing an issue significantly.

In this paper we try to shrink traces by eliminating elements. The individual inputs in the list are not changed since this usually results in other behaviour of the system. This implies that we do not need the last line in the shrinking algorithm for lists. Initial experiments have shown little or no effect of shrinking elements for the examples used in this paper. A thorough exploration of input element shrinking is a subject of future research.

5.1 Implementing Shrinking

In order to have a general implementation of shrinking that allows easy experimentation with various algorithms a binary tree for shrinking is defined:

```
:: Shrinks i = Shrinks [i] (Shrinks i) (Shrinks i) | NoShrinks
```

G∀st tests the sequence of inputs, `[i]`, in the node `Shrinks`. If this sequence of inputs shows an issue, shrinking continues with the first subtree, otherwise it continues with the second subtree. In this way we achieve a separation between the generating of candidate traces and the execution of the associated tests. Hence, one can implement shrinking strategies without detailed knowledge of test execution. Lazy evaluation prevents excessive use of memory for those trees.

5.2 Eliminating Single Transitions

When the states in the model before and after a transition are identical, the transition is most likely a query on the state. There is no guarantee whatsoever that the states in the system under test for such a transition are identical, but the trace where such a transition is removed is a good candidate for testing.

Based on this idea we can systematically try to eliminate inputs from a trace. Without looking at the states of the model we get an algorithm that resembles a repeated version of the basic shrinking algorithm very much. In order to obtain a somewhat efficient algorithm we use a greedy elimination algorithm; when the issue remains after removing a specific input element, this element is removed permanently. Otherwise, it will be part of the trace during the entire shrinking process. The algorithm use to generate the associated shrink tree is:

```
elemElimination :: [i] → Shrinks i
elemElimination inputs
```

```
 = elim 0 (length inputs) inputs
where
    elim n len inputs
    | n < len
        = Shrinks inputs2 (elim n (len−1) inputs2) (elim (n+1) len inputs)
        = NoShrinks
    where inputs2 = removeAt n inputs
```

The index n scans all inputs in the given list of inputs. When inputs2, with element n removed, also yields nonconformance we remove it forever. Otherwise, we keep it in the list and continue with the next element. The head to tail order in this algorithm is an arbitrary choice, we have no arguments why this would in general be better than any other order.

This shrinking algorithm appears to be very effective in removing individual transitions. On average the traces in our example in Section 6.3 are reduced by a factor of 33. The length of the obtained trace is largely independent of the initial trace. After shrinking the maximum difference in the length of the trace revealing an issue is 2. There is no correlation between the length of the initial trace revealing the issue and the trace after shrinking. Apparently this algorithm is able to remove many more transitions than just query transitions that do not change the state.

For a trace of length N there are $O(N)$ elements to consider as candidates to be removed. Testing whether each of these traces still produces an issue requires $O(N)$ transitions. Hence, shrinking with this algorithm requires $O(N^2)$ transitions of the system under test. The measurements show that the actual number of transitions needed grows indeed rapidly with the length of the initial trace. For initial traces longer than 450 the test system needs typically more that 100,000 transitions.

5.3 Eliminating Larger Chunks of Inputs

Based on the success of the previous algorithm to shorten traces revealing nonconformance, it is worthwhile to look for an algorithm that achieves this effect more quickly. The algorithm binElemElimination below tries to remove chunks of the trace from large to small. When removing some chunk succeeds we remove N elements in one step; otherwise we try to remove the first and second half of that chunk. The algorithm maintains a list of chunks as candidates to be eliminated. Each chunk is indicated by the index of the first element and its length.

When testing with this chunk yields an issue, the entire chunk is eliminated. The rest of the chunks is adjusted by changing all starting indices. The next chunk is split in two parts since it cannot also be removed completely. If it could be removed also, the original chunk would not be split in two parts.

When removing this chunk also removes the issue, the current chunk is split in two parts. These parts are new candidate chunks and are added to the list of chunks. The function binElemElimination constructs the associated shrinking tree. This function ensures that the last input element is never removed since that is the input revealing the issue.

```
binElemElimination :: [i] → Shrinks i
binElemElimination inputs
    | len > 1
        = elim [(0, len − 1)] inputs
        = NoShrinks
where
    len = length inputs

    elim [(f, d): rest] inputs
        = Shrinks inputs2 (elim (adjust d rest) inputs2)
            (elim (if (d > 1) [(f, d / 2), (f + d / 2, d − d / 2): rest]
                              rest) inputs)
        where inputs2 = slice f d inputs
    elim [] inputs = NoShrinks

    adjust d [(f, e): rest]
        | e > 1
            = [(f − d, e / 2), (f − d + e / 2, e − e / 2): rest1]
            = rest1
        where rest1 = [(x − d, y) \\ (x, y) ← rest]
    adjust d [] = []

slice :: Int Int [a] → [a]
slice 0 d list   = drop d list
slice f d [a: x] = [a: slice (f − 1) d x]
```

The effect of this algorithm is very significant. On average the number of transitions needed to show nonconformance is reduced by a factor of 39 in the running example. The effect is bigger for large initial traces. The produced final traces are almost always identical with the previous algorithm. Due to the different elimination order of inputs the resulting minimal traces can sometimes be a few transitions longer or shorter.

The binary search strategy is often used in computer science. For instance to find minimal counterexamples in the context of constraint solving problems, CSP [3, 7]. Here we use it to find minimal paths through a state machine that shows nonconformance in model-based testing. In contrast to CSP the order of elements is relevant here. Especially the last transition is essential in model-based testing since it is known to show nonconformance.

Although this algorithm works much better than the previous one in our experiments, its worst case complexity is not better. For instance, when the initial trace contains only needed inputs, or exactly every second input can be removed, we must continue by dividing chunks until their size is one. Hence, in worst case trying to remove the bigger chunks is no improvement. The experiments show that the actual reduction in transitions to be done is quite significant.

5.4 Eliminating Cycles

The previous algorithms are both based on the list of inputs and do not take any knowledge of the system into account. In particular it is worthwhile to try

to remove cycles in the model and system under test. A cycle is a subtrace starting and ending in the same state of the model. These cycles are excellent candidates for elimination since there is a fair change that cycle in the model is mirrors a cycle in the system under test, or has at least very similar states at its begin and end. By definition a cycle in a state machine can be removed without changing the remainder of the behaviour. In contrast to the previous algorithm, cycle removal uses educated guesses of the input chunks to be removed.

Based on the list of inputs it is impossible to identify cycles properly. Only when a subsequence occurs twice (or more), it likely that we have spotted a cycle. There are huge limitations on cycle detection based on the inputs. First, it is only possible to identify cycles that are used two or more times, a single tour over a cycle cannot be detected. Second, any intermediate input that only queries the system but does not alter the state spoils this cycle detection. Third, it is very well possible that inputs have to be repeated for the proper behaviour of the system. For instance, a vending machine can require two or more coins. Any repeated input in the trace and looks like a cycle, but it is not actually a cycle for the machine states.

Since the system under test is a black box we cannot observe its state. Hence, we cannot detect cycles there. For the model however, the test system knows everything. Apart from the inputs used in the trace, we can also record the states visited during the trace. As soon as we discover the same model state twice in the trace we have found an cycle. Next we can test if this cycle in the model can be removed. There is no guarantee whatsoever that a cycle in the model is also a cycle in the system under test.

In general there can be many cycles in a trace. In particular there can be small cycles in a bigger cycle, and cycles that are traversed more than once. For that reason the cycle removal algorithm sorts the cycles from large to small and tries to remove them in that order.

The function `cycleElimination` takes the list of inputs and corresponding model states as argument and produces a shrink tree. The first argument of this function is another shrinking algorithm to be used as continuation after cycle elimination. We explain its use below.

```
cycleElimination :: ([i]→Shrinks i) [i] [s]→Shrinks i | gEq{|*|},gLess{|*|} s
cycleElimination cont [] _ = NoShrinks
cycleElimination cont [i] _ = NoShrinks
cycleElimination cont inputs states = elim (findCycles states) inputs
where
    elim [c=:(f,t):cycles] inputs
        = Shrinks inputs2                                    // test case
                  (elim (updateCycles f t cycles) inputs2)   // Success
                  (elim cycles inputs)                       // Failure
          where inputs2 = cut f t inputs          // remove inputs from f to t
    elim [] inputs = cont inputs
```

The function `findCycles` gets the list of states corresponding to the current trace as argument and produces a sorted list of indices that identify the cycles found.

```
findCycles :: [s] → [(Int,Int)] | gEq{|*|}, gLess{|*|} s
findCycles []     = []
findCycles [s]    = []
findCycles states = sortBy (λ(a,b) (c,d) . b − a > d − c) (mkCycles groups)
where
    pairs  = sortBy (λ(i, s) (j, t) . s < t)
                    [(i,s) \\ i ← [0..] & s ← init states]
    groups = groupby (snd (hd pairs)) [] pairs

mkCycles :: [[a]] → [(a,a)]
mkCycles []              = []
mkCycles [[]:r]          = mkCycles r
mkCycles [[i:r]:next]    = reverse [(j, i) \\ j ← r] ++ mkCycles [r:next]
```

When a cycle is removed from the input, the indices of all other cycles have to be adapted by `updateCycles`. When one of the other cycles is inside the cycle removed, the smaller cycle is removed completely. Otherwise, only the indices are adapted to the removal of the cycle from the trace.

```
updateCycles :: Int Int [(Int,Int)] → [(Int,Int)]
updateCycles f t [(x, y): r]
    | y < f // new cycle before current cycle
        = [(x, y): updateCycles f t r]
    | x > t // new cycle after current cycle
        = [(x − t + f, y − t + f): updateCycles f t r]
    | otherwise // cycles overlap: remove new cycle
        = updateCycles f t r
updateCycles f t [] = []
```

The experiments show that about half of the cycles found in this way can be removed. In some ways cycle removal is more powerful than the previous element removal algorithms. In a number of situations it is possible to remove an entire cycle in one go, while removing the cycle element by element or in arbitrary chunks fails. However, some individual transitions that can be removed are not signalled by the cycle removal algorithm. For this reason the cycle removal algorithm is equipped with a continuation. After all cycles that can be removed are gone, we can try to remove individual transitions by one of the previous algorithms. In the next section we show the effect of using different continuations.

It is of course also possible to switch the order of these algorithms: first remove as many individual elements as possible and next try to remove any remaining cycles. Since cycle removal can remove large chunks at reasonable costs we apply the algorithms in the given order. For the effect on the final trace it does not matter, but the number of transitions needed to find the short trace is usually lower.

6 Measuring the Effect of Shrinking

In order to determine the effect of the various shrinking techniques we measured their effect for a number of examples. In this paper we discuss the experiments

with a vending machine in detail since it is sufficiently small to treat thoroughly and illustrates the effects well.

6.1 Vending Machine Specification

The vending machine is an extended state machine. The possible inputs are coins with value 1 or 2, the choice for some product, a reset button, an information button, and a go button to start the preparation of the selected product:

```
:: Input   = Coin1 | Coin2 | Choice Product | Reset | Info | Go
:: Product = Coffee | Espresso | Double | French | Wiener
```

The possible outputs are either a cup of one of the products, an amount of change, or some text giving information to the user.

```
:: Output  = Cup Product | Change Int | Text String
```

The input and output are identical for the model and the system under test. The states of those machines can be completely different.

The state of the model contains the select product, if any, and an integer for the balance.

```
:: State   = {product :: Maybe Product, balance :: Int}
```

The function spec is the specification of the vending machine used by G∀st. For the inputs Coin1 and Coin2 the balance is incremented accordingly. When a product p is chosen, this choice is stored in the state. On the input Reset state of the model becomes state0, when there was a positive balance in the system the appropriate amount of change is produced. On the input Info there is a function transition. The function used accepts any list of outputs that contain exactly one arbitrary text. When the Go input is given, the model checks whether a product is chosen and the balance is sufficient for that product. If these conditions hold, the output is a cup of the previously chosen product and the state is updated accordingly. Otherwise, there is no output and the state is not changed.

```
spec :: State Input → [Trans Output State]
spec s Coin1      = [Pt [] {s & balance = s.balance + 1}]
spec s Coin2      = [Pt [] {s & balance = s.balance + 2}]
spec s (Choice p) = [Pt [] {s & product = Just p}]
spec s Reset      = [Pt (if (s.balance > 0) [Change s.balance] []) state0]
spec s Info       = [Ft accept] where accept [Text _] = [s]; accept _ = []
spec s Go
   = case s.product of
       Just p | s.balance ≥ value p
              = [Pt [Cup p] {state0 & balance = s.balance − value p}]
       _      = [Pt [] s]

state0 = {product = Nothing, balance = 0}
```

6.2 Vending Machine Implementations

In order to measure the effect of shrinking we made a correct implementation, m0, of this specification in Clean and 10 mutants containing some erroneous transitions. The correct implementation mirrors the specification as much as possible. Instead of a list of allowed transitions, a system under test yields a pair of a list of outputs and the new state. The system under test can be some external program that, for instance in Java, as long as there is communication via some protocol with that implementation. Here we keep it as simple as possible and use the same state as the model. The only real difference between model and implementation corresponds to the input Info. The implementation has to decide what the output is. In this case it it shows the machine state (line 6).

```
m0 :: State Input → ([Output], State)                                    1
m0 s Coin1       = ([], {s & balance = s.balance+1})                     2
m0 s Coin2       = ([], {s & balance = s.balance+2})                     3
m0 s (Choice p) = ([], {s & product = Just p})                           4
m0 s Reset       = (if (s.balance > 0) [Change s.balance] [], state0)    5
m0 s Info        = ([Text (show1 s)], s)                                 6
m0 s Go                                                                  7
   = case s.product of                                                   8
       Just p                                                            9
         | s.balance ≥ value p                                           10
             = ([Cup p], {state0 & balance = s.balance − value p})       11
       _     = ([],s)                                                    12
```

Mutants of the Vending Machine Implementation For the mutants we only list the differences with the correct implementation m0.

m1 remembers the product chosen after preparing a cup of product p. Line 11 form m0 is replaced by:

```
= ([Cup p], {s & balance = s.balance − value p})
```

m2 loses any remaining balance after producing a product. Line 11 becomes:

```
= ([Cup p], state0)
```

m3 incorrectly increments the balance after inserting a Coin2. This typical copy-paste error is reflected in the new line 3:

```
m3 s Coin2       = ([], {s & balance = s.balance+1})
```

m4 has an incomplete reset functionality. The balance (if any) is returned, but this machine remembers the chosen product. Line 5 is replaced by:

```
m4 s Reset = (if (s.balance > 0) [Change s.balance] [], {s & balance=0})
```

m5 uses a uniform price of 3 for all products. This is correct for all products apart from coffee which has a value 2. This is reflected in line 11:

```
= ([Cup p], {state0 & balance = s.balance − 3}
```

m6 yields coffee independent of the requested product. Line 11 reads here:

```
= ([Cup Coffee], {state0 & balance = s.balance − 2})
```

m7 inserting a Coin2 in this machine erases the choice to nothing. Line 3 of this machine is:

```
m7 s Coin2      = ([], {s & product = Nothing, balance = s.balance+2})
```

m8 inserting a Coin2 sets the product choice to coffee. In this machine line 3 is replaced by:

```
m8 s Coin2 = ([], {s & product = Just Coffee, balance = s.balance+2})
```

m9 omits the balance check of line 10. This machine will produce the requested product even if no coins are inserted.

m10 remembers the product if there is money left in the machine after producing a product. The last function alternative is:

```
m10 s Go
    = case s.product of
        Just p
            | s.balance > value p
                = ([Cup p], {s & balance = s.balance − value p})
            | s.balance == value p
                = ([Cup p], state0)
        _   = ([],s)
```

6.3 Measurements

Table 1 contains the results of our measurements of the effect of shrinking with the different algorithms for the systems under test. The first two rows contain the number of transitions for using 10 different random seeds and the length of the shortest trace found. The next two rows contain the average number of transitions and average length of traces found by shrinking the results from random testing by element elimination by elemElimination. The next two rows contain the average number of transitions and average trace length for binary elimination by binElemElimination. The following sets of two rows contain those results for cycle elimination, cycleElimination, cycle elimination followed by element elimination, and cycle elimination followed by binary elimination.

The shrinking result with the shortest average length is printed bold and underlined. When several algorithms find the same minimal trace, the one which does this in the lowest number of transitions is underlined, the other shortest lengths are only bold. For **m2** the optimal result is plain binary elimination, for all other cases this is cycle elimination followed by binary elimination.

6.4 AVL Storage Testing

We repeated those measurements for a state-based system to test AVL-tree implementations. The input actions are to insert an element in the storage, remove

Table 1. Number of transitions and length of traces for the various shrinking algorithms and vending machines from Section 6.2

algorithm		m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
random	trans	3716	567	183	3307	1265	452	1409	1006	274	3690
seed	best	70	11	4	55	11	8	40	15	2	70
element	trans	88591	2067	222	91954	12688	1303	12792	7469	692	97162
elimination	avg	7	6	2	5	6	4	4	2	2	7
binary	trans	1073	275	47	1219	377	128	300	203	55	1140
elimination	avg	7	**6**	3	5	6	4	4	2	2	7
only cycle	trans	625	188	58	909	279	99	192	126	36	724
elimanation	avg	13	14	9	10	14	12	10	7	4	13
cycle +	trans	739	297	95	967	397	167	231	146	43	829
element	avg	7	**6**	2	5	5	4	3	2	2	7
cycle +	trans	731	289	71	948	364	140	216	133	40	821
binary	avg	**7**	**6**	**2**	**5**	**5**	**4**	**3**	**2**	**2**	**7**

an element, and check the presence of an element. The model stores the elements in a list. This makes those operations $O(n)$ in the number of arguments stored, but very easy to implement correctly. The 30 different systems under test store the elements in the AVL-tree are implementations made by our students as homework assignments [8].

The results are very similar to the results in Table 1. The only significant difference is that the effect of removing cycles is much stronger for those storages than for the vending machine. Space limitations prevents the listing of those results in this paper.

6.5 Observations

By comparing the shrinking results we make the following observations:

1. All heuristics used here have significant effects in finding shorter traces.
2. Trying different random seeds is the simplest, but least successful heuristic. The number of transitions required is high and the effect on the minimal path length is limited. It scales badly, $O(n^2)$; for longer minimal paths it is less effective and relatively more expensive.
   Nevertheless the average ratio between the length of the longest and shortest path with 10 different seeds for the pseudorandom generation is 17. Given the minimal implementation effort this is a good return of investment.
3. In all but one case even the best path found by random walk is considerably longer than the shortest paths found by other algorithms. The average path found by random walk is a factor of 33 longer than that path after shrinking. On average the length of the shortest path found by trying different random seeds can be reduced by a factor of 6 with respect to the other shrinking algorithms. Only when the shortest trace is extremely short, length 2, there is a fair chance to find the shortest path by a random walk.

4. The measurements show that it is not worthwhile to select the minimal trace by trying different seeds for the pseudorandom generation to have a better starting point for the real shrinking algorithms. Typically the shrinking algorithms are able to find a minimal trace independently of the initial trace. Searching a better starting point just increases the total number of transitions required.
5. Single element elimination, binary elimination, and cycle elimination followed by one of those heuristics are often equally effective. The difference in the length of the obtained trace in Table 1 is at most two transitions. For the AVL-trees the gain of cycle removal can be bigger. When there is a difference in effect, cycle elimination followed by single element elimination, or binary elimination is always more effective.
6. There are huge differences in the number of transitions needed to find minimal traces. Single element elimination is the most expensive. Binary elimination achieves almost the same effect (in worst case 2 additional transitions) faster, on average a factor of 39. For longer paths the reduction is bigger.
7. Cycle elimination alone is less effective than element elimination methods, but it is usually the cheapest heuristic to execute. Cycle elimination is a model-based shrinking technique that appears to scale very well.
8. Cycle elimination followed by binary element elimination combines best of both worlds. It yields typically the optimum shrinking result with a low number of transitions. Single and binary element elimination after cycle elimination are much cheaper due to the smaller initial trace for those algorithms.

7 Conclusion

To simplify the identification of errors based on issues found by model-based testing it is worthwhile to shrink the traces found. All heuristics tried in this paper have a significant shrinking effect. Typically the length of the trace revealing an issue is reduced by more than an order of magnitude. On average the reduction effect increases with the length of the trace. The best result-cost ratio is achieved by removing cycles based on repeated states in the trace of the model, followed by binary element elimination. This algorithm uses information from the model instead of only the list of inputs to guide the shrinking.

The examples used in this paper are representative for real world systems, but relatively small. In future research we will investigate whether the best shrinking results of this paper scales to real world applications. Since all the shrinking algorithms work with a shrink tree that monotonically decreases the size of the trace, it is possible to interrupt the algorithm if that would be necessary and still have a shrinking effect on the trace.

Acknowledgement. Special thanks to Thomas Arts of QuviQ for an interesting discussion about shrinking in model-based testing of state-based systems. We would like to express our gratitude to the anonymous referees and to the participants of TFP 2013 for their valuable feedback and suggestions.

References

1. Arts, T., Castro, L.M., Hughes, J.: Testing Erlang data types with Quviq Quickcheck. In: Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG, ERLANG 2008, pp. 1–8. ACM, New York (2008)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
3. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Form. Asp. Comput. 20(4-5), 379–405 (2008)
4. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the 5th International Conference on Functional Programming, ICFP 2000, Montreal, Canada, pp. 268–279. ACM Press (2000)
5. Vytiniotis, D., de Azevedo Amorim, A., Lampropoulos, L.: Testing noninterference, quickly. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, pp. 455–468. ACM, New York (2013)
6. Hughes, J.: Software testing with quickcheck. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) CEFP 2009. LNCS, vol. 6299, pp. 183–223. Springer, Heidelberg (2010)
7. Junker, U.: Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In: IJCAI 2001 Workshop on Modelling and Solving Problems with Constraints (2001)
8. Koopman, P., Achten, P., Plasmeijer, R.: Model based testing with logical properties versus state machines. In: Gill, A., Hage, J. (eds.) IFL 2011. LNCS, vol. 7257, pp. 116–133. Springer, Heidelberg (2012)
9. Koopman, P., Plasmeijer, R.: Testing reactive systems with Gast. In: Gilmore, S. (ed.) Proceedings of the 4th Symposium on Trends in Functional Programming, TFP 2003, pp. 111–129. Intellect Books (2004) ISBN 1-84150-122-0
10. Koopman, P., Plasmeijer, R.: Generic generation of elements of types. In: Proceedings of the 6th Symposium on Trends in Functional Programming, TFP 2005, Tallin, Estonia, September 23-24, pp. 163–178. Intellect Books (2005) ISBN 978-1-84150-176-5
11. Koopman, P., Plasmeijer, R.: Fully automatic testing with functions as specifications. In: Horváth, Z. (ed.) CEFP 2005. LNCS, vol. 4164, pp. 35–61. Springer, Heidelberg (2006)
12. Plasmeijer, R., van Eekelen, M.: Clean language report, version 2.1 (2002), `http://clean.cs.ru.nl`
13. de Vries, R., Tretmans, J.: On-the-fly conformance testing using SPIN. Software Tools for Technology Transfer, STTT 2(4), 382–393 (2000)
14. van Weelden, A., Oostdijk, M., Frantzen, L., Koopman, P., Tretmans, J.: On-the-fly formal testing of a smart card applet. In: Sasaki, R., Qing, S., Okamoto, E., Yoshiura, H. (eds.) SEC 2005. IFIP AICT, vol. 181, pp. 564–576. Springer, Boston (2005); Also available as Technical Report NIII-R0428

Control-Flow Analysis with SAT Solvers

Steven Lyde and Matthew Might

University of Utah, Salt Lake City, Utah, USA

Abstract. Control-flow analyses statically determine the control-flow of programs. This is a nontrivial problem for higher-order programming languages. This work attempts to leverage the power of SAT solvers to answer questions regarding control-flow. A brief overview of a traditional control-flow analysis is presented. Then an encoding is given which has the property that any satisfying assignment will give a conservative approximation of the true control-flow, along with additional ideas to improve the precision and efficiency of the encoding. The results of the encodings are then compared to those of a traditional implementation on several example programs. This approach is competitive in some instances with hand-optimized implementations. Finally, the paper concludes with a discussion of the implications of these results and work that can build upon them.

1 Introduction

A control-flow analysis determines the control-flow of a program. This is a difficult problem in higher order languages, because data-flow affects control-flow and control-flow affects data-flow. To address this issue, much work has been done. The first major effort was k-CFA as created by Shivers [8]. It is a family of algorithms where the chosen value of k determines the precision of the analysis. A higher value of k gives greater precision but at the cost of a greater runtime. When $k = 0$, the algorithm, more commonly known as 0CFA, has been shown to be cubic [9]. For $k \geq 1$, it has been shown that the algorithm is complete for EXPTIME [10].

We present an alternative approach to the problem by encoding a control-flow analysis into SAT. The results are more similar to 0CFA than k-CFA as SAT is a NP-hard problem, while k-CFA is EXPTIME-hard. Similar work that took the idea of encoding k-CFA into another problem for performance reasons was done by Prabhu et al. [7]. They run the analysis on a GPU by encoding the problem into matrix operations. Another work that will feel similar to the work presented in this paper is constraint based 0CFA analysis as summarized by Nielson [6]. They formulate 0CFA using constraints on sets and then provide an algorithm for solving these constraints. This work differs in that the constraints are not not encoded using matrices or sets, but propositional logic.

1.1 Motivation

Many problems are readily encoded into SAT and even though satisfiability is NP-complete, fast implementations are available. Every year there is considerable

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 125–133, 2014.

work being done to create efficient SAT solvers A CFA implementation based on satisfiability could benefit directly from that work.

1.2 Accomplishments

This work attempts to leverage the power of SAT solvers to answer questions regarding control-flow. It presents an encoding and compares its results to two traditional 0CFA implementations.

2 Preliminaries

In order to understand this work, you will need a passing understanding of continuation-passing style (CPS) lambda calculus and k-CFA. Brief descriptions of both will be given. The original formulation of k-CFA operates on CPS lambda calculus and this work operates on the same language.

CPS is similar to the untyped lambda calculus but with additional constraints: functions never return, all calls are tail calls; where a function would normally return, the current continuation is invoked on the return value; and when calling a function, the caller must supply a continuation procedure. There are three types of terms: applications, anonymous functions, and variables. The grammar for CPS lambda calculus follows.

$$
\begin{aligned}
\mathit{call} \in \mathsf{Call} &::= (f\ e\ \ldots) \\
f, e \in \mathsf{Exp} &= \mathsf{Var} + \mathsf{Lam} \\
v \in \mathsf{Var} &\text{ is a set of identifiers} \\
\mathit{lam} \in \mathsf{Lam} &::= (\lambda\ (v\ \ldots)\ \mathit{call})
\end{aligned}
$$

The abstract state space and the abstract semantics of k-CFA reformulated as an operational semantics are easily accessible [3]. The idea is to take an abstract machine and abstract it by making the number of addresses finite. Successor states are then generated, starting at the initial state of the program, until all the states have been visited. Because the number of addresses is finite the abstract state space is finite and the exploration will terminate.

3 Encodings

This section describes the devised encoding scheme. Here is a simple program we will work with in describing the encodings. In the following explanation, each lambda term will be identified by its label.

```
((lambda¹ (x)
   ((lambda² (y)
      (y (lambda³ (z) (x z)))) x))
 (lambda⁴ (a) (a a)))
```

For the encoding, we introduce a variable for every variable lambda pair in the program. The variable will be true if the lambda flows to the variable, and false if it doesn't. We will assume that the program has been alphatised, meaning that each variable is only bound by a single lambda. In the example we have four variables and four lambda terms, resulting in sixteen variables. Lambdas use their label as their subscript.

	λ_1	λ_2	λ_3	λ_4
a	a_1	a_2	a_3	a_4
x	x_1	x_2	x_3	x_4
y	y_1	y_2	y_3	y_4
z	z_1	z_2	z_3	z_4

To generate the clauses of our encoding we look at each point where binding occurs in lambda calculus, at application sites. From the grammar of CPS lambda calculus we can see that there are four cases which need to be considered. The function and the arguments at an application can either be a lambda term or a variable.

Case 1: Lambda Lambda The first case to consider is the simplest, when there is a lambda term in both function and argument position. The top level application of the sample program is an example of this.

```
((lambda[1] (x) call) (lambda[4] (a) (a a)))
```

We know that the lambda in argument position flows to the parameter of the lambda in function position. For this call site, we would add the clause x_4.

Case 2: Lambda Variable The second case to consider is when there is still a lambda in function position but a variable in argument position. Observe the following call site from the example.

```
((lambda[2] (y) call) x)
```

If we know a lambda flows to x, then we know that it must flow to y. We must assume that any lambda can flow to x, so we must create a clause for each lambda. This results in the following clauses: $x_1 \rightarrow y_1$, $x_2 \rightarrow y_2$, $x_3 \rightarrow y_3$, $x_4 \rightarrow y_4$.

Case 3: Variable Lambda The third case to consider is having a variable in function position and a lambda term in argument position. Observe the following call site from the example.

```
(y (lambda[3] (z) call))
```

We must assume that any variable can flow to y. Thus we need to create a clause for each lambda in the program. We infer that if a lambda term flows to y, then λ_3 will flow to the parameter of that lambda. This results in the following clauses: $y_1 \rightarrow x_3$, $y_2 \rightarrow y_3$, $y_3 \rightarrow z_3$, $y_4 \rightarrow a_3$.

Case 4: Variable Variable The most complicated case is when we have a variable in both function and argument position. Observe the following call site from the example program.

```
(x z)
```

We must assume that any lambda can flow to x and any lambda can flow to z. If we know that two flows are true for x and z, we can infer a third flow. For example, if we know λ_2 flows to x and λ_4 flows to z, we can infer that λ_4 flows to y, the parameter of λ_2. Thus we create the clause $x_2 \wedge z_4 \rightarrow y_4$. Since there are four lambda terms, there are 16 total such clauses that need to be generated.

4 Additional Encoding Details

The generated clauses described above are necessary but not sufficient. The problem is that every variable can be set to true and the formula is still satisfied. What we really want is the lowest possible number of flows set to true that still satisfy all the generated clauses. However, the SAT solver is free to give any satisfying solution. In the end, we have constraints that will never give us false negatives, but we need constraints that will ideally never give us false positives, or at least limit them. Note that in an anlysis, having false positives is still sound; only in having false negatives does the analysis become unsound.

4.1 Additional Encodings

For each case we will show additional clauses that can be added which will limit the number of false positives.

Case 1: Lambda Lambda Since the program is alphatised we not only know that the given flow must be true, but we know that all other flows to that variable must be false. For the above example we add the clauses: $\neg x_1$, $\neg x_2$, $\neg x_3$.

Case 2: Lambda Variable In the description found above, we said you could infer an additional flow if a given lambda flows to the variable in argument position. But more can be inferred since the program is alphatised. The clauses are not just implications, because the call site is the only place where the binding of the variable can occur. Thus we can change the clauses to equivalences: $x_1 \leftrightarrow y_1$, $x_2 \leftrightarrow y_2$, $x_3 \leftrightarrow y_3$, $x_4 \leftrightarrow y_4$.

Case 3: Variable Lambda

Unlike the previous case, we cannot turn the inference described in the previous section for case 3 into an equivalence. The issue is that because the lambda which flows to the variable in function position can flow to other application sites where there is a variable in function position, this is not the only place where a binding can occur. However, we can infer the disjoin of all the call sites where the binding could occur. An example will be be given below.

Case 4: Variable Variable

Much like the previous case, we cannot infer equivalences because bindings can happen at any call site where there is variable in function position. However, like the above case, additional clauses can still be created; we can infer the disjoin of all the call sites where the binding could occur. For example, if λ_3 flows to z it would mean that either λ_3 flows to y, λ_3 flows to a, or that λ_3 flows to x and λ_3 flows to z. Thus we would add the following clause: $z_3 \rightarrow y_3 \vee a_3 \vee (x_3 \wedge z_3)$.

4.2 Enhancements

The encodings presented above give way to some enhancements that can be used to make the encoding more efficient, by generating less clauses.

- Not all lambdas can flow. Lambdas that appear in function position cannot be bound to variables, thus we do not need to create a variable for pairs involving lambdas in function position.
- Not all lambdas are compatible. Although the example shows lambda terms with only one parameter, the lambda terms can have any number of parameters. When there is a variable in function position, only lambdas with the same number of parameters as there are arguments at the application site need to be considered.
- Some clauses will be trivially true. While iterating through every lambda, when faced with a variable at an application site, some of the implications will involve the same pairs on both sides, thus they are trivially true and can be omitted.

In the implementation, the first two enhancements were used, but the third was omitted.

4.3 Complexity

In the described encoding, many clauses can be generated. However, it is bounded by a polynomial of the size of the program. The worst case to consider is when you have a variable in both function and argument position. You must consider each lambda flowing to each variable. If there are n terms in the program, there are at most n call sites and n lambda terms. Thus the number of generated clauses will be bound by n^3. This seems logical as one of the simplest formulations of 0CFA is "nearly" cubic: $O(n^3/\log n)$ [2].

5 Implementation and Evaluation

We implemented the encoding in Scala using the back end of the analyzer written by Might et al. for parsing and preprocess transformations [5]. We compared its runtimes to those of that same analyzer, which closely follows the formal semantics, as well as a fast Racket implementation, which employs abstract

Church encodings and binary CPS lambda calculus [7]. MiniSat was used for solving the constructed encodings. All experiments were run on a 2.7 GHz Intel Core i7 on Ubuntu.

The first experiments were run on synthetic programs, which in a *constructive* complexity proof are shown to be the worst case for k-CFA when $k \geq 1$ and difficult for 0CFA [9,10]. The results can be found in the Table 1. The first column is the number of terms in the program. The second column is the runtime of the optimized Racket implementation. The Scala column is the runtime of the traditional Scala implementation. The SAT column is the time taken to encode and solve the problem using SAT. This column is broken down into its two components in the last two columns. The Encode column is the time taken to create the encoding. The Solve column is the time taken by MiniSat to solve the encoding.

Table 1. Runtime comparison of a control-flow analysis using a fast Racket implementation, a Scala implementation and using MiniSAT

Terms	Racket	Scala	SAT	Encode	Solve
37	0.008s	1.059s	0.730s	0.725s	0.005s
63	0.016s	1.056s	0.796s	0.792s	0.004s
115	0.046s	1.454s	1.025s	1.017s	0.008s
219	0.222s	2.338s	1.418s	1.387s	0.031s
427	1.374s	5.337s	2.759s	2.642s	0.117s
843	8.396s	44.873s	11.337s	10.481s	0.856s
1675	49.029s	12m34.301s	1m15.984s	1m9.222s	6.762s
3339	4m46.726s	>6h	8m50.671s	8m43.103s	7.568s

We also looked into the sensitivity of the encoding to different SAT solvers, using SAT solvers that were some of the best performers from the 2011 international SAT competition. See Table 2. We report the time taken to solve the encoding, the number of flows that agree with the Scala implementation and the number of flows that disagree. When there is a disagreement, the encoding says that the flow does occur but the traditional 0CFA reports that it does not.

From the results in Table 1, we see encoding the problem and solving it with MiniSat takes about the same amount of time as the fast Racket implementation. However, this is not always the case. Experiments were also run on more traditional benchmarks. To run these, the language on which the encoding operates had to be enriched. Additional constructs were added (*e.g.*, `if` and `set!`) as well as support for Scheme primitives. The fast Racket implementation could not be run on these examples without using Church encodings, as it only supports pure binary CPS lambda calculus. See Table 3.

The first two benchmarks test common functional patterns; sat is a simple SAT solver; rsa is a RSA implementation; prime is a Solovay-Strassen primality tester; scm2java is a Scheme to Java compiler; interp is a Scheme interpreter.

These benchmarks provide a stark contrast to the previous examples in performance. Further investigation is needed to find the source of this large difference

Table 2. Runtime and precision results from some of the best performers from the 2011 international SAT competitions

Solver	Results	$n = 37$	$n = 63$	$n = 155$	$n = 219$	$n = 237$	$n = 843$	$n = 1675$
minisat	Time	0.005s	0.007s	0.012s	0.039s	0.133s	0.848s	6.714s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
3S	Time	2.548s	2.570s	2.554s	2.777s	5.952s	1m15.335s	>2m
	Agree	96	280	936	3400	12936	50440	-
	Disagree	0	0	0	0	0	0	-
cirminisat	Time	0.004s	0.005s	0.009s	0.031s	0.152s	1.312s	11.299s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
clasp	Time	0.004s	0.005s	0.010s	0.029s	0.152s	1.055s	7.959s
	Agree	54	150	486	1734	6534	25350	165378
	Disagree	42	130	450	1666	6402	25090	33798
cryptominisat //	Time	0.007s	0.011s	0.026s	0.086s	0.413s	3.598s	>2m
	Agree	96	280	936	3400	12936	50440	-
	Disagree	0	0	0	0	0	0	-
csls //	Time	0.006s	0.006s	0.034s	0.579s	32.656s	>2m	>2m
	Agree	60	216	711	2754	12936	-	-
	Disagree	36	64	225	646	0	-	-
eagleup	Time	0.004s	0.006s	0.017s	0.063s	0.541s	18.500s	>2m
	Agree	70	192	674	2566	9479	36639	-
	Disagree	26	88	262	834	3457	13801	-
glucose	Time	0.011s	0.012s	0.020s	0.050s	0.198s	1.415s	4.805s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
glueminisat	Time	0.004s	0.005s	0.011s	0.036s	0.164s	1.363s	11.640s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
lingeling	Time	0.006s	0.010s	0.024s	0.078s	0.454s	1.980s	10.365s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
march_rw	Time	0.007s	0.010s	0.033s	0.466s	18.936s	>2m	>2m
	Agree	54	150	486	1734	6534	-	-
	Disagree	42	130	450	1666	6402	-	-
plingeling	Time	0.009s	0.012s	0.028s	0.096s	0.477s	2.851s	19.695s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
plingeling //	Time	0.010s	0.017s	0.037s	0.083s	0.465s	3.551s	30.653s
	Agree	96	280	574	1992	7813	30463	120112
	Disagree	0	0	362	1408	5123	19977	79064
ppfolio	Time	0.006s	0.009s	0.010s	0.051s	0.341s	2.453s	14.588s
	Agree	78	280	936	3400	12936	50440	165378
	Disagree	18	0	0	0	0	0	33798
ppfolio //	Time	0.007s	0.007s	0.012s	0.028s	0.178s	0.989s	7.409s
	Agree	92	280	936	3400	12936	50440	165378
	Disagree	4	0	0	0	0	0	33798
qutersat	Time	0.035s	0.042s	0.061s	0.134s	0.638s	4.786s	41.886s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
sattime2011	Time	0.005s	0.008s	0.021s	0.093s	0.796s	10.857s	0.020s
	Agree	83	230	805	2919	11275	44001	-
	Disagree	13	50	131	481	1661	6439	-
sparrow2011	Time	0.013s	0.007s	0.029s	0.100s	3.034s	>2m	>2m
	Agree	72	266	936	3400	10846	-	-
	Disagree	24	14	0	0	2090	-	-

Table 3. Runtime comparison between a traditional abstract interpreter and determining the control-flow usining MiniSAT

Program	Terms	Scala	SAT	Encode	Solve
eta	79	0.879s	0.805s	0.801s	0.004s
map	182	0.879s	0.805s	0.801s	0.004s
sat	250	1.311s	1.216s	1.198s	0.018s
rsa	609	1.805s	1.427s	1.396s	0.031s
prime	891	2.258s	4.584s	4.269s	0.315s
scm2java	2505	3.845s	1m6.550s	1m0.090s	6.460s
interp	4484	6.314s	5m6.519s	4m26.078s	40.441s

in performance. One possible explanation is that the Scheme primitives are not well modelled. Also, the traditional small step abstract interpreter is able to use widening to converge to the minimum fixed point faster. In addition, since its analysis is directed by the syntax of the program more closely, it can explore less spurious flows.

For the first set of benchmarks, the results returned by the encoding are exactly the same as those provided by the traditional implementations. However, running #SAT on the encodings, revealed that there are multiple valid interpretations. Thus the encoding does not exactly encode traditional 0CFA, which has a unique minimum fixed point.

5.1 Alternative Approach Using BDDs

Another approach attempted was to use a binary decision diagram (BDD) instead of a SAT solver to solve the constraints. The constraints are encoded in the same way, but the approach has the benefit that the minimum prime implicant is readily available from the structure of the BDD. The minimum prime implicant provides an equivalent solution as 0CFA. However, in practice, using a BDD requires large amounts of memory and time for even simple examples.

5.2 Alternative Approach Using MaxSAT

Another approach that could be promising is to use a MaxSAT solver instead of a traditional SAT solver. The additional clauses from Section 4 could be elided and only the clauses from Section 3 would be needed. The partial maximum satisfiability problem has two types of clauses, hard and soft. The hard clauses must be satisfied, while the soft clauses can be relaxed. The solver finds the assignment with maximum number of soft clauses satisfied. All the clauses from Section 3 would be hard clauses and then for each variable, its negation would be added as a soft clause. A satisfying assignment from this formulation would be equivalent to 0CFA.

6 Conclusion

This work has presented an encoding for control-flow analysis of CPS lambda calculus. It has shown that in some cases, the approach can be as fast as a highly optimized solution. While the soundness of the encoding was not proven, empirical results showed it to be accurate.

This work also provides a solid basis for additional work. Many avenues exist which can build upon it. Better encoding schemes can be developed, which possibly could be even more precise than 0CFA, given the extra power provided by SAT solvers being able to solve NP-complete problems. Van Horn and Mairson give a reduction from SAT to k-CFA, effectively showing how to do SAT solving with $k > 1$ CFA, which merits further investigation [9]. Also, while this work operates on CPS lambda calculus, the encoding could easily be adapted to work on a more direct style language, such as ANF lambda calculus [1], as analyzed by Might and Prabhu [4].

This work was supported by the DARPA programs APAC and CRASH.

References

1. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 237–247. ACM, New York (1993)
2. Midtgaard, J., Van Horn, D.: Subcubic control flow analysis algorithms. Tech. rep., Roskilde Unversitet (2009)
3. Might, M.: Environment Analysis of Higher-Order Languages. PhD thesis, Georgia Institute of Technology (June 2007)
4. Might, M., Prabhu, T.: Interprocedural dependence analysis of higher-order programs via stack reachability. In: Proceedings of the 2009 Workshop on Scheme and Functional Programming, Boston, Massachussetts, USA (2009)
5. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In: PLDI 2010: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, pp. 305–315. ACM Press (2010)
6. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, corrected ed. Springer (December 2004)
7. Prabhu, T., Ramalingam, S., Might, M., Hall, M.: EigenCFA: Accelerating flow analysis with GPUs. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, vol. 38, pp. 511–522. ACM Press, New York (2011)
8. Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
9. Van Horn, D., Mairson, H.G.: Relating complexity and precision in control flow analysis. In: ICFP 2007: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, pp. 85–96. ACM, New York (2007)
10. Van Horn, D., Mairson, H.G.: Deciding k-CFA is complete for EXPTIME. In: ICFP 2008: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 275–282. ACM Press (2008)

A Survey of Polyvariance in Abstract Interpretations

Thomas Gilray and Matthew Might

University of Utah
tgilray@cs.utah.edu, might@cs.utah.edu

Abstract. Abstract interpretation is an efficient means for approximating program behaviors before run-time. It can be used as the basis for a number of different useful techniques in static analysis more broadly, and can thus in-turn be used to prove properties needed for security or optimization. Polyvariance represents a way of obtaining higher precision in an abstract interpretation by producing multiple abstract states for each function or lexical point of interest in the program. This paper explores the role of polyvariance in these analyses and how it is manifested, unifying the disparate presentations in the literature.

1 Abstract Interpretation

An abstract interpretation is a non-deterministic interpretation of a program that determines abstract flow-sets, each representing all possible values a given expression could refer to during any particular concrete execution. The result is a finite abstract state-space which conservatively approximates a usually infinite number of different concrete state-spaces.

All valid paths in the program are guaranteed to be represented in a sound analysis. Above and beyond these genuine executions, imprecision is manifested as spurious traces which are indicated by the analysis but which do not exist in any concrete execution.

1.1 CPS λ-Calculus

For our survey of polyvariance, we will be using a simple language with familiar abstract semantics at each step to stay consistent. Call-sites are marked with a unique label which refers to its containing lambda. Consider the CPS λ-calculus:

$$\begin{aligned} call \in \mathsf{Call} &::= (ae\ ae\ \ldots)^l \mid (\text{halt}) \\ ae \in \mathsf{AE} &::= x \mid lam \\ lam \in \mathsf{Lam} &::= (\lambda\ (x\ \ldots)\ call) \\ x \in \mathsf{Var} &::= \textit{set of program variables} \\ l \in \mathsf{Label} &::= \textit{set of unique labels} \end{aligned}$$

The grammar structurally distinguishes between atomic expressions and call-sites to permit only calls in tail position. This constrains the language to a

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 134–148, 2014.

continuation-passing-style (CPS) form. Abstract interpretation can be implemented for any language so long as we have a concrete (in our case, operational) semantics to abstract. CPS is used here (as it was in its original formulation) purely for the purposes of simplifying our discussion. We can compactly represent its semantics using a CES-style machine:

$$\begin{aligned}
\varsigma \in State &= \mathsf{Call} \times Env \times Store \times Time \\
\rho \in Env &= \mathsf{Var} \rightharpoonup Addr \\
\sigma \in Store &= Addr \rightharpoonup Value \\
t \in Time &= Label^* \\
a \in Addr &= \mathsf{Var} \times Time \\
v \in Value &= \mathsf{Lam} \times Env
\end{aligned}$$

and a single small-step transition:

$$\frac{\langle (\lambda\ (x_1\ \ldots\ x_j)\ call),\ \rho' \rangle\ =\ \mathcal{A}(ae_f,\ \rho,\ \sigma)}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \rho,\ \sigma,\ t)\ \Rightarrow\ (call,\ \rho'',\ \sigma',\ t')}$$

$$\begin{aligned}
\text{where}\quad & \rho'' = \rho'[x_i \mapsto a_i] \\
& \sigma' = \sigma[a_i \mapsto \mathcal{A}(ae_i,\ \rho,\ \sigma)] \\
& a_i = (x_i,\ t') \\
& t' = l : t
\end{aligned}$$

where $\mathcal{A}$ is a concrete atomic-expression evaluator:

$$\begin{aligned}
\mathcal{A}(x,\ \rho,\ \sigma)\ &= \sigma(\rho(x)) \\
\mathcal{A}(lam,\ \rho,\ \sigma)\ &= \langle lam,\ \rho \rangle
\end{aligned}$$

Each state (machine configuration) contains a call-site, a binding environment, a value-store, and a timestamp. Each state transitions to a new state when a function can be invoked at the current call-site, or fails to transition and terminates when a (halt) is reached. The atomic-expression in call-position ae_f is evaluated to a closure and evaluation transitions to its body, another call-site. The closure's binding environment is augmented with addresses for each function-argument, and the store maps each of these to the value being bound. Each address is guaranteed to be unique because it is being paired with the new timestamp t'. t' is constructed by prefixing the current timestamp with a label for the current call-site. Because this call-history increases in length with each transition, no two values will share a binding.

1.2 0-CFA

0-CFA is the monovariant form of the k-CFA algorithm as presented in Shivers' seminal paper [25] [16]. We use an abstract version of our concrete semantics to

compute a conservative approximation of program behavior. In order to make this state-space finite, we need only to bound the size of our timestamp or call-history. k-CFA uses a k-length approximation of call-history, and 0-CFA merges all histories together.

As a repercussion of bounding $\widehat{Time}$, multiple values will now share a single address. Our abstract store maps addresses to flow-sets: sets of abstract values. All possible values for a particular variable now share the same address:

$$\begin{aligned}
\hat{\varsigma} \in \widehat{State} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Value}) \\
\hat{t} \in \widehat{Time} &= \mathsf{Label}^0 \\
\hat{a} \in \widehat{Addr} &= \mathsf{Var} \times \widehat{Time} \\
\hat{v} \in \widehat{Value} &= \mathsf{Lam} \times \widehat{Env}
\end{aligned}$$

The abstract transition function is non-deterministic, as multiple closures can be referenced by a single variable:

$$\frac{\langle (\lambda\ (x_1\ \ldots\ x_j)\ call),\ \hat{\rho}' \rangle\ \in\ \hat{\mathcal{A}}(ae_f,\ \hat{\rho},\ \hat{\sigma})}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ ())\ \approx\!\!>\ (call,\ \hat{\rho}'',\ \hat{\sigma}',\ ())}$$

$$\begin{aligned}
\text{where}\quad \hat{\rho}'' &= \hat{\rho}'[x_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\rho},\ \hat{\sigma})] \\
\hat{a}_i &= (x_i,\ ())
\end{aligned}$$

The abstract atomic-exppression evaluator returns flow-sets:

$$\begin{aligned}
\hat{\mathcal{A}}(x,\ \hat{\rho},\ \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(x)) \\
\hat{\mathcal{A}}(lam,\ \hat{\rho},\ \hat{\sigma}) &= \{\langle lam,\ \hat{\rho} \rangle\}
\end{aligned}$$

When discarding typographical differences, the two semantics are almost identical. There are essentially only two fundamental changes we've made to achieve a finite approximation: we use a finite set of abstract addresses to bound the size of our store, and introduce merging between values at each address. If we were including other basic types, we could also replace them with a finite abstraction. An unbounded set of numbers might become just $\{num\}$ to differentiate from other basic types, or perhaps elaborated slightly to $\{+, 0, -\}$ in order to perform a sign analysis.

In our case, the only types involved are closures, which thanks to our abstraction for addresses, are now drawn from a finite set. These however, are now being merged together at bindings in our abstract store. Where before we indicated a strong-update of our concrete store, we now use function-join to indicate merging sets of values together via set-union. In this way, all values which have have

ever been bound to an address are kept. In 0-CFA there is a single address for each program-variable. If some argument z is bound to 3 different closures in our analysis, all 3 need to be represented by the same address z upon completion [29].

1.3 Soundness

Soundness of an abstract interpretation entails showing that all possible concrete executions are represented by the final analysis in general, for all inputs. Its embarrassing imprecision notwithstanding, $\lambda x.\widehat{Value}$ is an example of a trivially sound store because it does indeed represent all possible flows in any concrete execution of any program.

Showing that a more precise analysis is sound in general involves introducing a bit more machinery we won't bother with fully, and so we'll not attempt to do more than give a very rough sketch of the proof here. A proof of soundness relies on defining the relationship between the concrete and abstract domains. This relationship is a pair of functions for abstraction and concretization known as a Galois Connection. Previous work has shown the use of this model in both proving an existing analysis sound, and in producing analyses which are correct by construction. Methods have been developed for automatically constructing abstract approximations of concrete machines through the composition of these Galois Connections [29] [15] [12].

To specify the correspondence between our abstract semantics and our concrete semantics, we would need to provide at least an abstraction function α which maps concrete states to their most precise abstract representative:

$$\alpha\colon State \to \widehat{State}$$

With this specification we can prove a statement *for each concrete transition* $\varsigma \Rightarrow \varsigma'$*, there exists an abstract transition* $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ *such that* $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$ *and* $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$ which shows that simulation is preserved across transition. The soundness proof for k-CFA has been published for both a denotational [25] and an operational [16] style of semantics.

1.4 Complexity

Termination is guaranteed because the search is being performed over a finite state-space.

0-CFA is known specifically to be of worst-case cubic complexity. To determine whether or not an abstract closure flows to a variable, requires examining at most each call site in the program $O(n)$. There are then at most $O(n) * O(n)$ of these possible flows because the number of variables is bounded by the size of the program, as is the number of lambdas [16]. The number of abstract closures in the monovariant analysis is the same as the number of lambdas since each abstract binding environment is fixed by the free variables in its function which can be determined lexically.

VanHorn and Mairson reduce the circuit value problem to an instance of the 0-CFA control flow problem, proving it to be PTIME-hard [27].

2 Polyvariance

In 0-CFA, each syntactic callsite is represented by a single abstract state. Polyvariance, in general terms, is the degree to which an analysis breaks up these syntactic points in the program and represents them with multiple differentiated abstract states.

2.1 k-CFA

k-CFA is the broader heirarchy of algorithms to which 0-CFA belongs. All forms of this algorithm where $k \geq 1$ represent increasingly polyvariant analyses. k-CFA differentiates states with the addition of an abstract history, or calling-context, referred to in its original presentation as an "abstract contour" [25].

The semantics below introduce a k-length calling-context $\hat{t}$ at each state which serves to differentiate like variables with unlike calling histories. Each calling-context is a tuple of call-site labels which represents the abstract history of calls that lead to a given state. The state's successors then get a calling-context which has lost its oldest callsite, and has been appended with the label for the most recent callsite. This new history is then included in the abstract addresses for these new states, differentiating their flow-sets and giving our binding environment a purpose for the first time.

$$\begin{aligned}
\hat{\varsigma} \in \widehat{State} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Value}) \\
\hat{a} \in \widehat{Addr} &= \mathsf{Var} \times \widehat{Time} \\
\hat{v} \in \widehat{Value} &= \mathsf{Lam} \times \widehat{Env} \\
\hat{t} \in \widehat{Time} &= \mathsf{Label}^k
\end{aligned}$$

$$\frac{\langle (\lambda\ (x_1\ \ldots\ x_j)\ call), \hat{\rho}' \rangle\ \in\ \hat{\mathcal{A}}(ae_f,\ \hat{\rho},\ \hat{\sigma}) \qquad \hat{t} = (l_1\ \ldots\ l_k)}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})\ \approx\!\!>\ (call,\ \hat{\rho}'',\ \hat{\sigma}',\ \hat{t}')}$$

$$\begin{aligned}
\text{where}\quad \hat{\rho}'' &= \hat{\rho}'[x_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\rho},\ \hat{\sigma})] \\
\hat{a}_i &= (x_i,\ \hat{t}') \\
\hat{t}' &= (l\ l_1\ \ldots\ l_{k-1})
\end{aligned}$$

A state $(call^{l_9},\ \hat{\rho},\ \hat{\sigma},\ (l_2\ l_5\ l_6))$ would mean that l_9 could be reached by a call from l_2, when reached after a call from l_5 and so-forth. A calling-context like

this if found in an address $(x,\ (l_2\ l_5\ l_6))$ would indicate that the values stored at this address were bound to x following the above history. Values in k-CFA are only merged once the fixed amount of call-history has been exceeded.

Consider an example where there are two calls of indirection in front of a function:

$$(\lambda x.\ (\lambda y.\ (\lambda z.\ \ldots)\ y)\ x)$$

Here, if x is bound to two different values in a 2-CFA analysis, by the time they reach z, the original context for the call to λx will have been lost and the values will be merged. If multiple values reach a recursive function, no matter how long a context is used, the values will eventually merge assuming the analysis cannot determine a bound on the calling depth before the context runs out. Using sufficiently precise abstract values to make this possible in the general case would tend to make the analysis impractical to compute.

2.2 Exponential Complexity for $k \geq 1$

The use of these call-string histories pays dividends where unlike call-sites provide a lambda with unlike abstract values. Where the history used is sufficient to capture these differences, they will be kept apart in the store, avoiding the usual merging and loss of precision. The major downside of k-CFA for $k \geq 1$ is that its precision against run-time trade-off comes at too great a price: polyvariant k-CFA is intractible for real world inputs.

Though long suspected, the proof that k-CFA is EXPTIME-complete came only recently in another work by Van Horn and Mairson [27].

3 Object Sensitivity

Object Sensitivity is an alternative notion of context for object-oriented languages which can be used in place of call-string histories or in conjunction with them [21]. There are various presentations of this strategy with subtle differences. The flavor which best fulfills the original intentions of the technique, and which has appeared most effective in practice is k-full-object-sensitivity by Smaragdakis, Bravenboer, and Lhotak [26]. This method differentiates argument-bindings by the allocation-history of a member-function's receiving object. This requires syntactic allocation-points to be stored inside the abstract representation of an object upon creation, so they can be retrieved later when one of its methods is invoked. When $k = 0$, object-sensitivity is equivalent to 0-CFA.

Consider a 1-full-object-sensitive analysis of Java. The abstract value for an object will store its allocation-point internally and when a method is invoked on the object, its bindings are made specific to this saved context. Take for example:

$$\mathsf{Object\ obj\ =\ new\ Object}();^{l_3}$$

The abstract value which flows into obj contains an allocation-history (l_3). When a method obj.m(...) is invoked, its bindings are unique to this program-point.

To extend this strategy to deeper levels of context sensitivity, we include allocation-history from the object which performs the allocation. If we instead wish to perform a 3-full-object-sensitive analysis on the same program, our addtional context is drawn from the variable *this* at the allocation-site. For example, if *this* contains an allocation-history $(l_8\ l_4\ l_9)$, the variable obj is represented by an object with a timestamp $(l_3\ l_8\ l_4)$.

3.1 Closure Sensitivity

We cannot easily modify our previous analysis to faithfully represent true object-sensitivity because the CPS λ-calculus does not include objects or classes. Instead we present a purely functional analog of this technique we call closure-sensitivity. Just as object instances are the building blocks of object-oriented programs, closures are the building blocks of functional programs. Objects can be implemented as a closure which accepts an additional parameter for selecting the method to invoke. Likewise, flat-closures can be implemented as an object with a single method. The allocation-point of a function is thus the syntactic position of the lambda, its point of closure-creation.

With this in view, we can produce a context-sensitve analysis of our language where abstract closures directly contain their allocation history.

$$\hat{v} \in \widehat{Value} = \ \mathsf{Lam} \times \widehat{Env} \times \widehat{Time}$$

Instead of appending a label for the current call-site to our timestamp, an abstract transition simply pulls the allocation-context out of our closure and uses this for new bindings.

$$\frac{\langle (\lambda\ (x_1\ \ldots\ x_j)\ call),\ \hat{\rho}',\ \hat{t}' \rangle\ \in\ \hat{\mathcal{A}}(ae_f,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})\ \approx\!\!>\ (call,\ \hat{\rho}'',\ \hat{\sigma}',\ \hat{t}')}$$

$$\begin{aligned} \text{where}\quad & \hat{\rho}'' = \hat{\rho}'[x_i \mapsto \hat{a}_i] \\ & \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})] \\ & \hat{a}_i = (x_i,\ \hat{t}') \end{aligned}$$

When a lambda is atomically evaluated, this allocation-point is combined with the current context and stored inside the abstract closure. This requires a slight modification so the current context $\hat{t}$ is available to the atomic-expression evaluator:

$$\begin{aligned} & \hat{\mathcal{A}}(x,\ \hat{\rho},\ \hat{\sigma},\ \hat{t}) = \hat{\sigma}(\hat{\rho}(x)) \\ & \hat{\mathcal{A}}(lam^l,\ \hat{\rho},\ \hat{\sigma},\ (l_1\ \ldots\ l_k)) = \{\langle lam,\ \hat{\rho},\ (l\ l_1\ \ldots\ l_{k-1}) \rangle\} \end{aligned}$$

This analysis has the same fundamental complexity as k-CFA, but where call-sensitivity causes merging, closure-sensitivity might not and vice versa. As the basic technique has proved more effective than k-CFA in practice for languages like Java [26], our analog may have something to offer in the functional realm.

4 The Cartesian Product Algorithm

The Cartesian Product Algorithm (CPA) was originally introduced as an enhancement to a type inference algorithm which itself can be viewed as a specialization of the abstract interpretation concept: one where dynamic program types are used as constituents of the abstract value domain. We will present the source of imprecision that the original formulation attempts to address, generalize the solution as a form of polyvariance in abstract interpretations (as was suggested in publications which followed), and discuss CPA's complexity and precision relative to k-CFA.

4.1 The Problem / Original Formulation

In an abstract interpretation using types for values, where polymorphism is non-existent each flow-set could contain a maximum of one value each, and the algorithm reduces to a straightforward type-inference. Therefore, the authors of CPA introduce it as an enhancement to a basic flow-set based type-inference algorithm where polymorphic functions introduce merging and thus spurious concrete variants. They turn a single polymorphic call in the analysis into multiple monomorphic calls, preserving the precise values across function calls, and their inter-argument relationships.

The basic algorithm that CPA enhances works similarly to an abstract interpretation over types. It also assigns a flow-set of dynamic types for each variable in the program, but it then establishes constraints based on the program text, and propagates values until all these constraints have been met. The primary method for overcoming this merging, is introduced as the p-level expansion algorithm of Palsberg and Schwartzbach – a kind of type-inference analog to call-string histories in k-CFA, where the use of p parallels that of k. This is shown to be insufficient however, as the authors of CPA give a case of merging which cannot be overcome by any sized p. Their motivating example is the polymorphic *max* function:

$$max(a, b) = if\ a > b\ then\ a\ else\ b$$

Here, the only constraint for an input to *max* is that it support comparison, so a call $max(\text{“}abc\text{”}, \text{“}xyz\text{”})$ makes as much sense as a call $max(3, 5)$. However, if both these calls are made with a sufficient amount of obfuscating call-history behind them, merging will cause the flow-sets for both a and b to each include both *string* and *int*. This is imprecise as it implies that a call $max(int, string)$ is possible when it is not.

The solution that CPA proposes is to replace flow-sets of per-argument types, with flow-sets of per-function tuples of types. In such an analysis, the function *max* itself would be typed $\{(int, int), (string, string)\}$ preserving inter-argument patterns and eliminating spurious concrete calls like $(int, string)$ [1].

4.2 Abstract Contour Formulation

In essence, this change makes flow-sets for each argument specific to the entire tuple of types received in a call. This suggests an abstract contour representation which pairs variables with tuples of abstract values in the store, instead of pairing them with call histories as in k-CFA [17].

$$\widehat{Store} = \widehat{Addr} \rightharpoonup \widehat{Value}$$
$$\widehat{Time} = \widehat{Value}^*$$

This would seem to maintain perfect precision; exact values would be known for any given address. The problem with this approach is that it introduces recursion into our state-space making it again unbounded. Closures contain environments containing contours made of closures. Our analysis again becomes a concrete interpreter using arbitrarily precise values to differentiate one another in the store.

To faithfully extend this algorithm to a higher-order language, in the spirit of its original presentation, we reduce abstract values to their types. An abstract value like *string* could potentially remain as it is, but closures must be limited to a finite set of types. We've chosen to reduce them to only their syntactic lambda, merely dropping environments, on the assumption that this point in the program is associated with a single type signature – whether it is known pre-analysis or not.

$$\widehat{Store} = \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Value})$$
$$\widehat{Time} = \mathcal{P}(\widehat{Type})^*$$
$$\widehat{Type} = \mathsf{Lam}$$

A helper function can be defined which performs this reduction:

$$\hat{\mathcal{T}} : \mathcal{P}(\widehat{Value}) \to \mathcal{P}(\widehat{Type})$$

At each call, a new contour is formed by reducing each of the flow-sets of the atomically-evaluated function arguments:

$$\frac{\langle(\lambda\ (x_1\ \ldots\ x_j)\ call), \hat{\rho}'\rangle \in \hat{\mathcal{A}}(ae_f,\ \hat{\rho},\ \hat{\sigma})}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})\ \approx\!\!>\ (call,\ \hat{\rho}'',\ \hat{\sigma}',\ \hat{t}')}$$

$$\begin{aligned}
\text{where}\quad & \hat{\rho}'' = \hat{\rho}'[x_i \mapsto \hat{a}_i] \\
& \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\rho},\ \hat{\sigma})] \\
& \hat{a}_i = (x_i,\ \hat{t}') \\
& \hat{t}' = (\hat{\mathcal{T}}(\hat{\mathcal{A}}(ae_1,\ \hat{\rho},\ \hat{\sigma}))\ \ldots\ \hat{\mathcal{T}}(\hat{\mathcal{A}}(ae_j,\ \hat{\rho},\ \hat{\sigma})))
\end{aligned}$$

The semantics for a CPA-like abstract interpretation are fundamentally that of k-CFA with the exception that our abstract contours in $\widehat{Time}$ are now tuples of

abstract values. This preserves the exact flows from function to function and the only merging now possible is exists in the predetermined merging inherent to our abstract values. Essentially, each abstract address is already specific to the exact abstract value it points to in the store, thus flows are no longer sets, and non-determinism can no longer occur at a call-site. Non-determinism would in practice be reintroduced by the addition of practical language constructs such as primitive operations on basic values, conditionals/if-statements, etc. Each of these would need abstract transition rules which produce multiple monomorphic abstract calls as opposed to making a single call which sends a set of abstract values.

4.3 Precision and Complexity

It is straightforward to see intuitively that CPA is more precise than k-CFA, as is discussed in the original publication. For any pre-determined value of k, a program can be constructed which nests calls passed this call depth and causes merging. Any such merging, even when completely precise at the level of a particular argument, can produce spurious inter-argument patterns. CPA on the other hand differentiates functions directly based on the full tuple of arguments they receive and obtains perfect precision for a given finite set of abstract values.

That no length call-history can match the precision of CPA has also been formally demonstrated on an object-oriented language [2]. It is important to note that k-CFA contains context information which CPA does not and which might be useful for its own sake. k-CFA may also be more general and amenable to infinitely wide value domains, while CPA may rely more directly on the finiteness of the abstract values used to ensure computability.

CPA is of-course, like k-CFA, of exponential complexity, and exceedingly impractical for use on sufficiently complex input programs. Somewhat ironically, where CPA improves precision, it is also fastest, and where CPA is unnecessary and delivers no improvement over k-CFA, it is enormously inefficient. For a function like *max*, one where the types of the arguments should match, CPA might require as few as one flow per-type; just as with k-CFA, except carries a vast improvement in precision. For a function where all combinations of arguments are possible, CPA requires each to be explicitly made, while k-CFA implies them for equal precision at far greater efficiency.

5 Practical Polyvariance

In contrast to CPA's attempt to improve on the precision of abstract call-string histories, attempts have been made to bring a degree of call-string history polyvariance to an analysis without incurring the full cost of 1-CFA.

5.1 Polymorphic Splitting

Polymorphic Splitting is a compromise between 0-CFA and k-CFA where the length of the contour used varies on a per-function basis. Lambdas which have

been let-bound are analyzed with a contour length 1-greater than that of their parent expression. In this way, let-bindings can be used as a heuristic for guiding the length of the contour used within a function during analysis. Because the number of let forms within-which an expression can be nested is bound by the program's size, the maximum length of k is likewise fixed.

In order to give a semantics for polymorphic splitting which will work on our simple language, make it quickly understandable to the reader, and comparable to the other analyses discussed here, we imagine our program has been CPS-converted from a direct-style language with a let-form, and we add a k annotation to call-sites which indicate their function's let-depth:

$$call \in CALL \ ::= \ (ae \ \ldots)^{l}_{k}$$

This annotation can then be used to direct the amount of polyvariance used in our abstract transition:

$$\frac{\langle(\lambda \ (x_1 \ \ldots \ x_j) \ call_{k'}), \hat{\rho}'\rangle \ \in \ \hat{\mathcal{A}}(ae_f, \ \hat{\rho}, \ \hat{\sigma}) \qquad \hat{t} = (l_1 \ \ldots \ l_k)}{((ae_f \ ae_1 \ \ldots \ ae_j)^{l}_{k''}, \ \hat{\rho}, \ \hat{\sigma}, \ \hat{t}) \ \approx\!\!> \ (call, \ \hat{\rho}'', \ \hat{\sigma}', \ \hat{t}')}$$

$$\begin{aligned}
\text{where} \quad & \hat{\rho}'' = \hat{\rho}'[x_i \mapsto \hat{a}_i] \\
& \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i, \ \hat{\rho}, \ \hat{\sigma})] \\
& \hat{a}_i = (x_i, \ \hat{t}') \\
& \hat{t}' = (\text{take} \ (l \ l_1 \ \ldots \ l_k) \ k') \\
& \hat{\mathcal{A}}(x, \ \hat{\rho}, \ \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(x)) \\
& \hat{\mathcal{A}}(lam, \ \hat{\rho}, \ \hat{\sigma}) = \{\langle lam, \ \hat{\rho}\rangle\}
\end{aligned}$$

We have added a subscript k' to the call-site found in a closure for ae_f which determines the length of the contour we'll use for our new argument bindings. The function $(take \ lst \ n)$ removes all but the first n entries from lst.

This presentation of polymorphic splitting may perhaps introduce a confusion as to how we know that k' is not greater than $k+1$; that there is enough history for us to use at each transition. This concern is unlikely to arise from looking at the original semantics. We know a call into a let-bound function is unreachable above that let form's body, and since that let form's body shares the contour length of its parent expression, it can be at most one less than the contour length of the let-bound function.

The complexity of polymorphic spitting remains exponential, as it easily devolves into doing all the work of a k-CFA analysis in the worst-case, however it has been empirically shown to be practical for sizable benchmarks. The authors found its precision comparable to that of a 1-CFA, while its running times were closer to that of 0-CFA. That it even beat the running time for 0-CFA in some test-cases can be attributed to its higher precision culling spurious paths which would have otherwise been explored by the monovariant analysis [30].

5.2 Polynomial-Time 1-CFA

Polynomial-time 1-CFA differentiates each state with a single call history, as 1-CFA does, but only allows free variables in a closure's environment to remember this history for a single closure creation deep. Each time a function is called, its abstract contour is updated and all the flows for its free variables are propagated to the new history for that call. They then share a history with the latest arguments to be sent in all new closures created. An environment in this analysis boils down to the single abstract contour it maps all variables onto. We simplify this and pair lambdas directly with a single contour to form a closure:

$$\hat{\varsigma} \in \widehat{State} = \mathsf{CALL} \times \widehat{Store} \times \widehat{Time}$$
$$\hat{v} \in \widehat{Value} = \mathsf{LAM} \times \widehat{Time}$$
$$\hat{t} \in \widehat{Time} = \mathsf{Label}$$

$$\frac{\langle(\lambda\ (x_1\ \ldots\ x_j)\ call), \hat{t}'_b\rangle\ \in\ \hat{\mathcal{A}}(ae_f,\ \hat{t},\ \hat{\sigma})}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\sigma},\ \hat{t})\ \twoheadrightarrow\ (call,\ \hat{\sigma}',\ \hat{t}')}$$

where
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(x_i, \hat{t}') \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{t},\ \hat{\sigma})] \sqcup \bigsqcup\{[(y, \hat{t}') \mapsto \hat{\mathcal{A}}(y,\ \hat{t}_b,\ \hat{\sigma})] \mid y \in free(call)\}$$
$$\hat{t}' = l$$
$$\hat{\mathcal{A}}(x,\ \hat{t},\ \hat{\sigma}) = \hat{\sigma}((x, \hat{t}))$$
$$\hat{\mathcal{A}}(lam,\ \hat{t},\ \hat{\sigma}) = \{\langle lam,\ \hat{t}\rangle\}$$

Because the closure is updated at each call, the binding environment previously in the second position of our abstract state is redundant with the single call-history in the final position, so we omit it. Likewise, the creation of a new binding environment (previously called $\hat{\rho}''$) is no longer needed as it was in k-CFA since it would simply be set to $\lambda\_.\hat{t}'$ and so is subsumed here by $\hat{t}'$ itself. Our updated store is one joined with the bindings formed by the function call, along with bindings which propagate values for the free variables in the function to their new contour.

Polynomial-time 1-CFA improves on 0-CFA in many of the usual places. Function parameters given different values at different callsites are analyzed polyvariantly. Where it compromises as compared with full 1-CFA is in the addresses used for free variables. When a function is closed over its free variables, they are differentiated by the call-history of the containing lambda. Upon invocation however, these values are propagated to addresses using the most recent callsite. This means if we call a function $\lambda x.\lambda y.x$ more than once, we may obtain multiple different abstract closures, but if we invoke each of them at the same callsite l, all these variants of x will be merged together into an address $(x,\ l)$.

Polynomial-time 1-CFA has not yet been empirically investigated, but its complexity has an upper bound of $O(n^6)$ [8].

6 The Future

The potential for new explorations into this area looks bright. The recent paper *A posteriori soundness* by Might and Manolios [20] has provided an exceptionally general guarantee of soundness for abstract allocation functions which allows for nearly any form of merging or differentiation in the store which could be conceived. Even methods which tune a live analysis directly for precision are allowed for, so no fully pre-defined strategy would even be necessary.

6.1 A Posteriori Soundness

The usual process for demonstrating the soundness of an abstract interpretation is *a priori* in the sense that the concrete and abstract transition relations along with the abstraction map relating the two state-spaces have been defined in advance, and are then justified as sound before any analysis is produced. *A posteriori* soundness differs from this in that a portion of the justifying abstraction map cannot be known until after the analysis is run.

The *a posteriori* soundness proof relies on factoring apart the concrete semantics, abstract semantics, and their correspondence. A portion of the abstraction map α is isolated which represents the correspondence between concrete addresses and abstract addresses: α_L. A portion of the transition relation is also factored out which represents the process of producing bindings. The abstract transition relation can then be parameterized by an allocation-policy $\hat{\pi}$ which determines this process for a given abstract state. The crux of the argument is then that given a non-deterministic selection of $\hat{\pi}$, a justifying α_L can always be produced after the fact, which proves the prior selection sound – whatever it might have been. This means that so long as the remaining analysis follows a single liberal soundness condition: the choice of allocation policy $\hat{\pi}$ is entirely arbitrary as far as the correctness of the analysis is concerned [20].

6.2 Precision-Adaptive Analyses

The implication of this is that the allocation policy $\hat{\pi}$ of an abstract interpretation can be selected entirely with precision and complexity in view. A policy can even adapt to the source text itself to make these choices without soundness needing to be proven for each specific program. If soundness needed to be proved *a priori*, this would not be possible since the mechanics of the proof would rely upon aspects of specific programs which could not be known in advance. The work thus not only simplifies deciding that a new form of polyvariance would be sound, but makes it possible to produce polyvariant analyses which use different amounts of history for different functions, different kinds of history for different functions, and which make these decisions while the analysis is still live.

7 Conclusion

The concept of polyvariance in abstract interpretations covers a wide array of techniques which allows for an analysis to be tuned up or down along the

precision/complexity trade-off. Merging and differentiation of flow-sets in the store, beyond one address per variable, requires a value on which to base the differentiation: in the case of k-CFA this is Shivers' abstract contour. It has since been proved that any basis for differentiation which obeys a single liberal constraint will remain sound, and a number of specific variants on the traditional contour have already been discussed in the literature each offering a unique trade-off in precision.

References

1. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Echtle, K., Powell, D.R., Hammer, D. (eds.) EDCC 1994. LNCS, vol. 852, pp. 2–26. Springer, Heidelberg (1994)
2. Besson, F.C.: beats ∞-CFA. Formal Techniques for Java-like Programs, p. 7 (July 2009)
3. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F, pp. 421–506 (1999)
4. Cousot, P.: Types as Abstract Interpretations. In: Symposium on Principals of Programming Languages, pp. 316–331 (1997)
5. Cousot, P., Cousot, R.: Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principals of Programming Languages, pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Symposium on Principals of Programming Languages, pp. 269–282 (1979)
7. Felleisen, M., Findler, R., Flatt, M.: Semantics Engineering with PLT Redex (August 2009)
8. Jagganathan, S., Weeks, S.: A Unified Treatment of Flow Analysis in Higher-Order Languages. In: ACM Symposium on Principles of Programming Languages, pp. 393–407. ACM Press (January 1995)
9. Jones, N.D.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Symposium on Principles of Programming Languages, pp. 66–74 (1982)
10. Jones, N.D., Muchnick, S.: Flow analysis of lambda expressions (preliminary version). In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 114–128. Springer, Heidelberg (1981)
11. Midtgaard, J.: Control-Flow Analysis of Functional Programs. ACM Computing Surveys 44 (June 2012)
12. Midtgaard, J., Jensen, T.: A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 347–362. Springer, Heidelberg (2008)
13. Midtgaard, J., Van Horn, D.: Subcubic Control Flow analysis Algorithms. Higher-Order and Symbolic Computation (May 2009)
14. Midtgaard, J., Jensen, T.: Control-ow analysis of function calls and returns by abstract interpretation. In: International Conference on Functional Programming (2009)
15. Might, M.: Abstract interpreters for free. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 407–421. Springer, Heidelberg (2010)

16. Might, M.: Environment Analysis of Higher-Order Languages. Ph.D. Dissertation. Georgia Institute of Technology (2007)
17. Might, M.: Logic-Flow Analysis of Higher-Order Programs. In: Principals of Programming Langauges, pp. 185–198 (January 2007)
18. Might, M., Shivers, O.: Environment analysis via ΔCFA. In: Symposium on the Principals of Programming Languages, pp. 127–140 (January 2006)
19. Might, M., Shivers, O.: Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In: International Conference on Functional Programming, pp. 13–25 (September 2006)
20. Might, M., Manolios, P.: *A posteriori* soundness for non-deterministic abstract interpretations. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 260–274. Springer, Heidelberg (2009)
21. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transaction on Software Engineering and Methodology, 1–41 (2005)
22. Nielson, F., Nielson, H.R., Hankin, C.: Principals of Program Analysis. Springer (1999)
23. Palsberg, J., Pavlopoulou, C.: From Polyvariant Flow Information to Intersection and Union Types. In: Principals of Programming Languages, pp. 197–208 (1998)
24. Shivers, O.: Control-flow analysis in Scheme. In: Programming Language Design and Implementation,pp. 164–174 (June 1988)
25. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. PhD dissertation. School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Technical Report CMUCS-91-145 (May 1991)
26. Smaragdakis, Y., Bravenboer, M., Lhotak, O.: Pick your contexts well: understanding object-sensitivity. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 17–30. ACM, New York (2011)
27. Van Horn, D., Mairson, G.H.: Deciding k-CFA is complete for EXPTIME. In: International Conference on Functional Programming, pp. 275–282 (September 2008)
28. Van Horn, D., Mairson, H.G.: Flow analysis, linearity, and PTIME. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 255–269. Springer, Heidelberg (2008)
29. Van Horn, D., Might, M.: Abstracting Abstract Machines. In: International Conference on Functional Programming 2010, Baltimore, Maryland, pp. 51–62 (September 2010)
30. Wright, A.K., Jagannathan, S.: Polymorphic splitting: An effective polyvariant flow analysis. ACM Transactions on Programming Languages and Systems, 166–207 (January 1998)

Functional Video Games in CS1 III
Distributed Programming for Beginners

Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu

Abstract. This article advocates that developing distributed multiplayer video games using functional programming should be a new trend in the CS1 classroom. This is premised on two facts: most students are excited by video game development and distributed programming is now common and not beyond the abilities of beginning students. A design recipe for the development of distributed applications is presented which has successfully been used at Seton Hall University over the past few semesters. The primary goal is to expose students to distributed programming and to have students think about some of the problems programmers face when writing distributed applications. To the CS1 instructor, this article presents a model for developing their own distributed programming module.

1 Introduction

The explosion in development of internet applications (such as social media sites and associated games) and the arrival of multicore processors to the mass market make it clear that the use of distributed programming is a trend that is likely to become as common as the use of the light bulb. Therefore, it is desirable for a CS1 course to introduce students to distributed programming. The key in CS1 is to expose students without expecting them to become experts–expertise is developed in a more advanced course. To be successful, however, distributed programming must be made appealing to students and must be presented in a manner that is accessible to them.

This article argues that developing distributed multiplayer video games using functional programming ought to be a new trend in the CS1 classroom. This is premised on two facts: most students are excited by video game development and distributed programming is now common and not beyond the abilities of beginning students. The approach implemented at Seton Hall University (SHU) using the *Program by Design* methodology presented in *How to Design Programs* (HtDP) is described. A novel design recipe for the development of distributed applications is presented. This new design recipe is used to illustrate how first-year students can be led to develop a multiplayer Space-Invaders-like game called Aliens Attack. The development of the game builds on letting students develop code that contains subtle distributed programming bugs, like process synchronization and communication overhead, which motivate refinements.

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 149–167, 2014.

2 Background

2.1 Student's Design and Programming Experience

At SHU, the introductory Computer Science courses focus on problem solving using a computer [9,10]. The languages of instruction are the successively richer subsets of Racket known as the student languages which are tightly-coupled with HtDP [5]. Before being introduced to distributed programming, students have studied topics such as: primitive data, primitive functions, programmer defined functions and variables, programmer defined data, processing finite compound data, processing arbitrarily large compound data and structural recursion, and abstraction with higher-order functions. These topics are covered following much of the structure of HtDP [5]. There are two 75-minute lectures every week and the typical classroom has between 20 to 30 students. In addition to the lectures, the instructor is available to students during office hours (3 hours/week) and via e-mail.

The curriculum, however, also varies in significant ways from HtDP by including a module for distributed programming. Distributed programming is introduced after structural recursion for two reasons: our experience suggests that students that have developed some programming expertise do not find distributed programming intimidating and from a student's perspective much more interesting video games can be developed after knowing how to design programs that process data of arbitrary size. The curriculum also places a great deal of emphasis on iterative refinement with a video game going through versions that grow in complexity as the course advances culminating in a multiplayer distributed version.

2.2 The Universe Teachpack

The course uses the universe teachpack [6] for video game development which provides the functionality to develop distributed games. The clients/players/worlds in a universe exchange messages with a server. The universe teachpack provides two functions to create messages: make-package and make-bundle. The first is used by a client to create a pair that contains a (possibly new) game state and a message to the server. The second is used by the universe server to create a structure that contains a (possibly new) server state, a list of mails to any of the clients, and a list of worlds to be disconnected from the universe. The constructor for a mail, make-mail, requires the recipient client and the message. Any message transmitted must be an *S-expression*. This means that students must design and implement functions to marshal and unmarshal their data–a topic first-year students can understand and will encounter later in an operating systems course [12]. This set-up also forces students to program using a specific API which is a useful skill to have them develop.

The syntax required to create a player's world specifies handlers that update the game or render the game to the screen. Version 1 of the game requires:

```
(big-bang
   INIT-WORLD                    ;; initial world
   (on-draw draw-world)          ;; handler for drawing the world
   (on-key process-key)          ;; handler for key events
   (on-tick update-world)        ;; handler for clock ticks
   (stop-when game-over?)        ;; handler to test for game end
   (register LOCALHOST)          ;; registers with the server
   (on-receive process-message) ;; handler for incoming messages
   (name MY-ID))                 ;; name of this world
```

During development students use LOCALHOST as the address of the server, but at play time they may also use an internet address to specify where the server is running.

The syntax for the universe server is similar and specifies the event handlers. For version 1 of distributed Aliens Attack the following syntax is required:

```
(universe
   initU                      ;; the initial universe
   (on-new add-new-world)     ;; handler for new worlds joining
   (on-msg receive-message)   ;; handler for incoming messages
   (on-disconnect rm-world)) ;; handler for worlds disconnecting
```

This syntactical set-up provides a framework to get students started. Specifically, students identify the handlers which are needed and must write each handler along with any auxiliary functions that may be needed. Readers interested in further details about the universe teachpack are referred to the help pages in DrRacket [8] and the modest guide on how to design worlds [4].

3 A Design Recipe for Distributed Computing

After developing a single player Aliens Attack [9], students ask if it is possible to have multiple players. Figure 1 displays a snapshot of a multiplayer version of Aliens Attack that they have in mind. Students are generally excited about the possibility of playing together which sets the stage to discuss distributed programming.

Students are led through an informal discussion of what is needed to write a multiplayer Aliens Attack. The idea of the game being distributed naturally comes to our students given their experience with internet games. They realize the need to send and to receive messages as well as the need for a server that provides support to coordinate all the players/clients. The following design recipe for distributed programming is presented:

1. Divide the problem into components.
2. Draft data definitions for the different components and the server.
3. Design a communication protocol.
4. Design and implement marshalling-unmarshalling functions and create data definitions for messages.

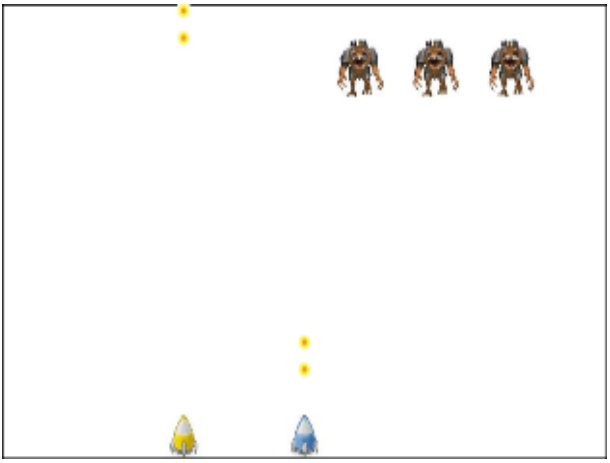


Fig. 1. A Snapshot Illustrating Multiplayer Aliens Attack

5. Design and implement the components (starting with handlers)
6. Design and implement a server (starting with the handlers).
7. Test your program.

One of the main goals of the above design recipe is to gently introduce students to distributed programming. Students are explained that, as any other design recipe seen earlier in the course, each step has a specific outcome. This new design recipe, however, is more akin to the design recipe for generative recursion in HtDP (which provides less guidance on how to complete the steps) than to the design recipes for structural recursion. Like generative recursion, distributed programming requires the development of insight into a problem in order to identify components and to understand how to integrate components. The first four steps are intended to help students develop such insight which guides the actual development of code for individual components and the server using the design recipes in HtDP.

In Step 1, they must define what each component/client as well as the server does. In Step 2, they must draft data definitions for the data that each component/server is to manipulate. In Step 3, they must define a communication protocol specifying when a client sends a message to the server and when the server sends mail to a client. An efficient way to achieve this is by using protocol diagrams that illustrate when communication occurs. In step 4, students must develop marshalling and unmarshalling functions. This step provides an excellent opportunity to help students make a connection with a topic they have studied in their Mathematics courses given that a marshalling and the corresponding unmarshalling function are inverses of each other. Another important result of this step is data definitions for different kinds of messages. In step 5, students must design and implement the handlers as well as any necessary auxiliary functions for each client.

In step 6, students design and implement the server. In step 7, students must test their programs and redesign/reimplement if necessary. In this step, students must consider the subtle problems that arise in distributed programming such as process synchronization, communication overhead, and speed.

4 Multiplayer Aliens Attack Version 1

Students are asked to think about how to make a multiplayer game from their single player Aliens Attack [9]. By an overwhelming margin, the most common answer is to add to each single player the other players. That is, each player runs their game and others can join. The details of how to do this are, of course, fuzzy at best and they are invited to use the new design recipe.

4.1 Problem Components

Students identify each player as a component that is responsible for rendering the state of the game, moving a single rocket, moving the aliens, changing the direction the aliens are moving in, and moving the shots. In addition, each component must provide support for a list of allies and must receive messages to reproduce the actions taken by other players. This component decomposition is very attractive to students, because it means that they can re-use code they have written for a single player Aliens Attack by making a small number of changes and additions (e.g., the development of a message processing handler). This turns out to be important to keep frustration low with what some students view as a Herculean task at the beginning.

Student-guided class discussion leads to the server being responsible for receiving messages from the players indicating their rocket moving and shooting actions and for broadcasting said messages to all the other players. That is, a thin-server is the intuitive choice for (most) students. In addition, the server sends the initial army of invading aliens to the first player that joins the game.

4.2 New Data Definitions

For a player, the new data definitions are displayed in Figure 2. For the server, the only new data definition is for a universe (of players/worlds/clients):

```
;; A universe is a (listof iw), where iw is an iworld.
```

An iworld is the internal representation used by the universe teachpack for the clients that join the server. All these data definitions are in familiar territory for the students and require the development of examples and function templates.

4.3 Communication Protocol Design

A communication protocol is described to beginning students as a collection of communication chains. A communication chain is defined as a series of messages

```
;; A rocket is a non-negative number.
;; An ally rocket, (make-ar x n), is a structure where x is a number
;; and n is a string for the name of the player that controls it.
(define-struct ar (x name))
;; A list of ally rockets (loar) is a (listof ar).
;; An alien is a posn.
;; A list of aliens (loa) is a (listof alien).
;; An alien army (aa) is  either 'uninitialized or a loa.
;; A world is a structure, (make-world r l a d s), where r is a rocket,
;;   l is an loar, a is an aa, d is a string, and s is a los.
(define-struct world (rocket allies aliens dir shots))
```

Fig. 2. Player Data Definitions for Multiplayer Aliens Attack Version 1

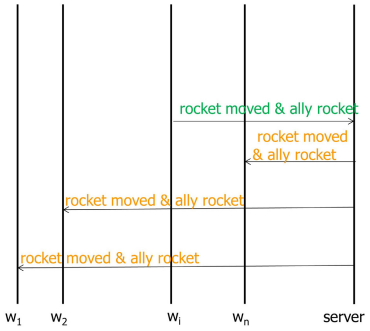


Fig. 3. Communication Protocol for a Rocket Move

that are exchanged between the server and the clients. These chains are sparked by either an action taken by a client or an action taken by the server. A communication chain is visualized using a *protocol diagram*–a diagram illustrating the messages in a chain. This abstraction is understood by students and allows for a well-focused discussion during classroom development.

In Aliens Attack, a player sparks a communication chain when a key event occurs. That is, when a rocket move or shot is made by, p_i, the i^{th} player. For example, when p_i moves the rocket, a rocket-moved message is sent to the server that includes the new ally rocket[1]. The server forwards the message to all the other players. Figure 3 displays the protocol diagram for a rocket move. A similar protocol diagram is developed for shot creation.

[1] To all other players a move made by p_i is a move made by an ally rocket.

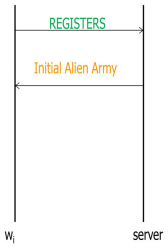


Fig. 4. Joining an empty universe

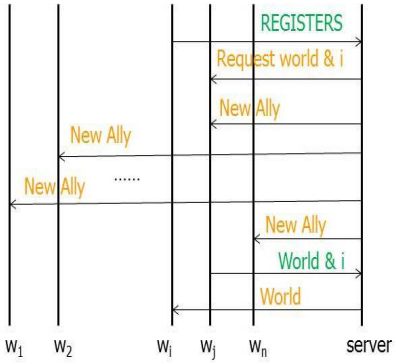


Fig. 5. Joining a non-empty universe

The server sparks a communication chain when its state changes. Classroom analysis reveals that this occurs two times: when a new player joins the game and when a player disconnects from the game. Two cases are distinguished when a player, p_i, joins the universe. In the first case, the new player is the first in the universe and the server only needs to send the initial alien army. This communication chain is captured in the protocol diagram in Figure 4. In the second case, p_i, joins a non-empty universe. In this case, the server requests the state of the game from an existing player, p_j such that $i \neq j$, with a mail that includes i. The server also sends a new-ally message to all the worlds already in the universe. After the server receives a message from p_j that includes the state of the game and the destination for said state (i.e., i), the server forwards the game state to p_i. This communication chain is captured in the protocol diagram in Figure 5. A similar analysis leads to the communication chain required when a player leaves the game.

4.4 Design Marshalling and Unmarshalling Functions and Data Definitions for Messages

Students are now ready to design and implement marshalling and unmarshalling functions as well as to develop data definitions for messages. Marshalling is done by converting data into an S-expression and appropriately tagging the message. Unmarshalling is done by removing the tag and reconstructing the original data.

There are two types of messages: To-Server messages and To-Client messages. To-Server messages are identified by incoming arrows to the server in the protocol diagrams. Likewise, To-Client messages are identified by incoming arrows to the clients. Each set of clients that can receive different kinds of messages must have their own To-Client message data definition. In Aliens Attack this task is

```
A To-Server Message is either:
  1. (list 'rocket-move rocket string)
  2. (list 'new-shot number number)
  3. (list 'world
           string
           (list-of (list-of number string))
           (list-of (list-of number number))
           string
           (list-of (list-of number number)))
```

Fig. 6. To-Server Message Data Definition

simplified since all clients are the same. Thus, only one To-Client data definition is required which is ideal for pedagogy in CS1.

Consider the communication chain in Figure 3. The protocol requires that a rocket-move message be sent to the server that includes the new ally rocket created by the move. This means that an ally rocket must be marshalled and unmarshalled. Since an ally rocket is a structure with a number, n, and a string, s, a To-Server rocket-move message is defined as a list containing the symbol rocket-move, n, and s. The corresponding marshalling and unmarshalling functions are:

```
; ally-rocket --> message
(define (marsh-rckt-mv an-ar)
  (list 'rocket-move (ar-x an-ar) (ar-name an-ar)))

; message --> rocket
(define (unmarsh-rckt-mv m)
  (make-ar (first (rest m)) (first (rest (rest m)))))
```

Repeating this process for every incoming arrow to the server labeled differently in the protocol diagrams leads students to a complete data definition for a To-Server message as displayed in Figure 6, to the development of marshalling and unmarshalling functions, and to a function template for functions that process To-Server messages.

To develop the data definition for messages to clients, observe in the protocol diagrams that any To-Server messages is echoed to the clients. Therefore, a To-Client message can be a To-Server message. The protocol diagrams also inform us that that the server can send a player a message to request the world, to send the initial alien army, and to inform a player of a new ally or of an ally lost. The To-Client message data definition is displayed in Figure 7 from which a corresponding function template is developed.

4.5 Component Implementation

Each different component is independently implemented. For Aliens Attack all clients are the same except for their identifying name (a string). This simplifies

```
A To-Client Message is either:
  1. To-Server Message
  2. (cons 'init-army (listof (listof number number)))
  3. (list 'rm-ally string)
  4. (list 'req-world string)
  5. (list 'new-ally number string)
```

Fig. 7. To-Client Message Data Definition

the task for students given that only one component needs to be developed. Furthermore, students can see that their task now is to refine their single player code into multiplayer code. This requires updating functions that process data whose definition has been refined, adding communication code to functions that make changes to the state of the game, and the creation of a message processing function. For the students, the updates are not hard nor intellectually obscure. Previously in the course, students have had to refine their code when a refinement has been made to a data definition. This step is not surprising to them, but some do find it tedious and time-consuming.

The addition of communication code is, however, a new element for them. For any arrow in the protocol diagrams that goes from a player to the server, communication code must be added. For example, the protocol diagram in Figure 3 tells us that a rocket-move message must be sent to the server when the rocket is moved. This means that their original key event handler requires small updates: updating the function signature to return a package and updating the function body to create a package by marshalling the rocket move. The updated code is displayed in Figure 8. As the reader can observe, adding communication code to the client is not complex for students after a communication protocol has been designed. Performing the same work for all out-going arrows from a player to the server yields the refined functions for player-sparked communication chains.

The final step implements a handler to process To-Client messages. This function is written by specializing the function template for To-Client messages which contains a conditional statement to distinguish among the variety of messages. This handler takes as input a world and a message and it returns a (new) world. For example, when a rocket moved message arrives the list of allies is updated and a new world is produced. This snippet illustrates the idea:

```
[(symbol=? 'rocket-move (first mess))
 (make-world (world-rocket w)
             (update-allies (unmarsh-rckt-mv mess)
                            (world-allies w))
             (world-aliens w)
             (world-dir w)
             (world-shots w))]
```

```
; process-key: world key --> package
; Purpose: Handler to process key events.
(define (process-key a-world key)
  (cond
    [(key=? "up" key) (process-up-key a-world)]
    [else (local [(define new-world
                    (make-world
                       (move-rocket (world-rocket a-world) key)
                       (world-allies a-world)
                       (world-aliens a-world)
                       (world-dir a-world)
                       (world-shots a-world)))]
            (make-package new-world
                          (marsh-rckt-mv
                            (make-ar (world-rocket new-world)
                                     MY-ID)))])))
```

Fig. 8. Refined Key-Processing Handler for Multiplayer Aliens Attack

4.6 Server Implementation

The server is implemented in a top-down manner starting with the handlers and consulting the protocol diagrams. For example, the handler used when a player joins the game is based on the protocol diagrams of Figures 4 and 5. This function takes as input a universe and a joining iworld and produces a bundle. To create the new universe, the joining world is added to the list of current worlds. The mails that must be generated depend on the state of the universe. According to Figure 4, if the universe is empty the server sends the joining world the initial alien army. According to Figure 5, if the universe is not empty the server requests the game state from an existing world and sends a new ally message to all the current players in the universe. There are no worlds that need to be disconnected from the universe. The resulting handler is displayed in Figure 9. The handler for a world disconnecting from the game is developed in the same fashion.

The server's message processing handler is developed using the template for functions on a To-Server message. For example, for the communication chain in Figure 5 the following snippet of code is written:

```
[(symbol=? (first msg) 'world)
 (make-bundle
   u
   (list (make-mail (get-world (first (rest msg)) u) msg))
   empty)]
```

This snippet keeps the universe unchanged, forwards the world message to the player indicated in the message, and removes no players from the universe.

```
; add-new-world: universe iworld --> bundle
(define (add-new-world u w)
  (make-bundle
   (cons w u)
   (cond [(not (empty? u))
          (cons (make-mail (first u) (marsh-req-world (iworld-name w)))
                (map (lambda (iw)
                       (make-mail iw  (marsh-new-ally (iworld-name w))))
                     u))]
         [else (list (make-mail w (marsh-loa INIT-ALIEN-ARMY)))])
   empty))
```

Fig. 9. The Server's New Player/World Handler

4.7 Testing

Students are advised that testing is two-fold: the testing they are familiar with checking that functions produce the correct output (using Racket's check-expect library) and testing for bugs that only arise in distributed programming such as synchronization, communication overhead, and deadlock.

The distributed programming bugs are tested for by running the game. Students see on their screens a working game with allies, but unlike the experienced reader they do not realize there is a synchronization bug. The instructor ought to let the students discover the bug by joining the game and projecting the instructor's screen to the class. It does not take long for students to realize that not all players have the game in the same state. This approach makes process synchronization a real concern for students and with some class discussion they realize that messages take time to travel from the source to the destinations. While the messages travel, the source player continues changing the state of their game. Thus, different players have different states.

5 Multiplayer Aliens Attack Version 2

Students quickly realize that a possible solution is for the game state to reside in one location and this strongly motivates the next refinement. It is important to note that this refinement is not prescribed by the instructor based on knowledge that students do not have. Instead, this refinement has its genesis in the students based on their results from version 1 of the multiplayer game.

5.1 Problem Components

Students identify each player as a component that is responsible for rendering the state of the game to the screen and for processing key events. Players do not update the state of the game and, therefore, do not need a handler to update the world every time the clock ticks. When a key event occurs, a message is sent to

the server requiring a new key event handler. The syntax required for a player is:

```
(big-bang INIT-WORLD
          (on-draw draw-world)
          (on-key process-key)
          (on-receive process-message)
          (register LOCALHOST)
          (name MY-ID)
          (stop-when game-over?))
```

As in version 1, the server needs handlers to add new players, to remove players, and to process messages. The server is now also responsible for maintaining the state of the game, thus, requiring a handler for clock ticks. When the state of the game changes, the players are sent the new state. In essence, the students are defining a thick-server that is solely responsible for all the necessary computing. The required syntax for the server is:

```
(universe initU
          (on-new add-new-world)
          (on-msg receive-message)
          (on-disconnect rm-this-world)
          (on-tick update-univ))
```

5.2 Draft Data Definitions

Students are led to see that in this refinement there is no need to distinguish between a rocket and the allies. For the server, all the players are allies each of which is still controlled by a single player. In addition, students include a boolean in the game state to indicate if the game has ended. The following is the refined data definition for the game state:

```
;; A world is a structure, (make-world l a d s o), where
;; l is a loar, a is a aa, d is a string, s is a los, and
;;    o is a boolean.
(define-struct world (allies aliens dir shots over))
```

Given the added work done by the server, the representation of the state of the server must also be refined to include both the state of the game and, as before, the players represented as iworlds. The refined data definition for the state of the server is:

```
;; A univ is a structure, (make-univ l w), where l is a
;;    (listof iworld) and w is a world
(define-struct univ (worlds state))
```

5.3 Communication Protocol Design

Students are asked when does a player initiate a communication chain and are asked to develop protocol diagrams. Figure 10 displays the protocol diagram

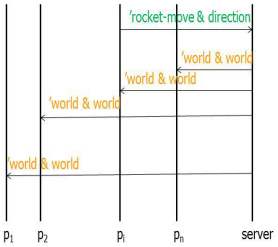


Fig. 10. Version 2 Protocol Diagram for a Rocket Move

students develop for a rocket move. A player sends the server a rocket move message containing the direction of the move. The server processes the move and sends all players an updated world. A similar diagram is developed for new shots.

Students realize that the server starts a communication chain when a player joins the game, a player disconnects from the game, and when the game state is updated after a clock tick. The protocol diagrams are easy to visualize with the server always sending the game state to all the players.

5.4 Design Marshalling and Unmarshalling Functions and Data Definitions for Messages

The protocol diagrams reveal to students that there is only one variety for a To-Client message and only two varieties for To-Server messages in this refinement:

```
A To-Client message is:
    (list 'world
          (listof (listof number string))
          (listof (listof number number))
          string
          (listof (listof number number))
          boolean)

A To-Server message is either:
    1. (list 'rckt-move string)
    2. (list 'new-shot number number)
```

This means only three pairs of marshalling-unmarshalling functions. For example, for a rocket move we have:

```
; process-key: world key --> package or world
; Purpose: This function is the handler to process key events.
(define (process-key a-world key)
  (cond [(key=? "up" key)
         (make-package
          a-world
          (marsh-shot (make-posn (get-my-x (world-allies a-world))
                                 ROCKET-Y)))]
        [(or (key=? "left" key) (key=? "right" key))
         (make-package a-world (marsh-rckt-move key))]
        [else a-world]))
```

Fig. 11. Player key-event handler for version 2

```
; string --> message
(define (marsh-rckt-move direction) (list 'rckt-move direction))

; message --> string
(define (unmarsh-rckt-move m) (first (rest m)))
```

The most complex pair is the one for a world which provides the opportunity to reinforce lessons using lambda expressions and higher-order functions like map.

5.5 Component Implementation

Implementing the components means updating the handlers for processing key events and for rendering the game state using the refined data definition for world. In addition and according to the protocol diagrams, communication code must be added for key event handling and message handling. As before, one goal is to reuse as much code as possible.

Figure 11 displays the handler for key events developed by the students during class discussion using the protocol diagrams. If the "up" key is pressed, the state of the game is not changed by the player and a message with a new marshalled shot is sent to the server. Similarly, if the "left" or "right" key are pressed the state of the game is not changed and a marshalled rocket move is sent to the server. If any other key is pressed, the state of the game is unchanged and no message is sent to the server (which means a package is not constructed).

Given that there is only one type of To-Client message the message handler is very straightforward:

```
; process-message: world message --> world
(define (process-message w mess)
  (cond [(symbol=? 'world (first mess)) (unmarsh-world mess)]
        [else (error "World received an unknown message" mess)]))
```

Similarly the handler to check if the game has ended is also straightforward for students at this point in the course:

```
; univ --> univ
(define (update-univ u)
  (cond [(game-over? (univ-state u))
         (make-bundle
         u
         (map (lambda (iw)
                (make-mail iw (marsh-world (mk-end-wrld (univ-state u)))))
                (univ-worlds u))
          empty)]
        [else
          (local [(define new-w (update-world (univ-state u)))]
            (make-bundle
               (make-univ (univ-worlds u) new-world)
            (map (lambda (iw) (make-mail iw (marsh-world new-w)))
                 (univ-worlds u))
            empty))]))
```

Fig. 12. Clock tick handler for version 2

```
; world --> boolean
(define (game-over? w) (world-over w))
```

5.6 Server Implementation

The four handlers for the server are implemented during class in the same manner as version 1. The handler to add a new world dispatches on whether the state of the universe has an empty list of iworlds or not. The message handler dispatches on the two varieties of To-Server messages. The handler used when a player disconnects, creates a bundle with a new list of iworlds that does not contain the disconnected player and a new game state in which the disconnected player is not one of the allies.

The clock tick handler is the most complex. It dispatches on whether or not the game has come to a end. If the game is over, then a world in which the over flag is set is mailed to all the players. Otherwise, the state of the server is updated by updating the state of the game. This updated game state is mailed to all the worlds. A sample implementation developed by students is displayed in Figure 12.

5.7 Testing

Testing reveals that the synchronization problem appears resolved. We say *appears*, because we do not prove that it is resolved[2]. An instructor can, indeed, leave it at that and move on. Students have done enough to get them started thinking about synchronization. There is, of course, an additional issue that can

[2] Program correctness is not yet woven into CS1 at SHU.

be pointed out to students. In the case of Aliens Attack, the order in which shots are added to the game state does not matter. In a different distributed application, however, order may very well matter and students are made aware that in such cases mutual exclusion must be guaranteed. This topic is not thoroughly discussed, but students are told that solutions will be studied, for example, in an operating systems course.

More importantly for our purpose, testing also reveals a most annoying characteristic for students: the game is much slower. The issues of bottleneck and communication overhead are brought forward during class discussion. This motivates the development of a third version of the game.

6 Multiplayer Aliens Attack Version 3

The development of version 2 marks the end of lecturing in the distributed-programming module in CS1 at SHU. Students now have some experience with a complex communication protocol (version 1), with a simple communication protocol (version 2), and with some important bugs that arise in distributed programming. It is time for them to test their skills and their understanding on their own.

The next refinement of the game is assigned as a group project. Students are divided into groups of 2 or 4 students. Each group is further divided into two subgroups. One subgroup is responsible for developing the components (i.e., the players) and the other is responsible for developing the server. The subgroups must work together to agree on the data definitions, the communication protocol, and the marshalling functions. Then each subgroup develops their own code. When both subgroups are ready, they get together to test their program and, hopefully, enjoy the game and/or fix bugs.

The programs developed by students have been extremely encouraging. Students submit working games that employ a communication protocol that can be described as middle of the road between version 1 and version 2. That is, they keep the components of version 2, but do not transmit the whole state of the game every time a server makes an update. Instead, they only transmit the part of the state that is changed. This type of communication protocol has been implemented in practice by, for example, Quake 3 [11]. Having CS1 students writing distributed applications on their own is nothing short of amazing.

7 Student Assessment

After each semester of CS1 at SHU, students are asked to fill out a short survey to evaluate the distributed-programming module. On a scale from 1 (low) to five (high), students are asked if distributed programming is intellectually stimulating. The average of the distribution to date is 3.35 with 76% of the students answering 3-5. The middle 50% of the students are in the range 3-4. Surprisingly (to the author), a follow-up question reveals that students felt that in terms of intellectual stimulus distributed programming was much like what they have

been doing all semester. From the student's perspective, the module contained new interesting material, but the transition to distributed programming required mostly tasks they had done before. This can only be interpreted as a success for the described methodology. The introduction to distributed programming is gentle enough that students feel it is a natural progression that builds on what they have learned.

Students are also asked to rank how much more difficult distributed programming is to non-distributed programming on a scale from 1 (not more difficult at all) to 5 (a lot more difficult). The average of the distribution to date is 3.9 with the middle 50% in the range 3-5. A follow-up question revealed that the top reason distributed programming is harder is error messages that are not very informative. This type of problem occurred mostly when there were bugs in the marshalling and unmarshalling functions that led to "unknown message" errors or errors trying access parts of a message that did not exist. The difficulty lies in that a message that, for example, causes the server to crash is not always fixed in the server's code. Instead, it may have to be fixed in the client's code. Students, however, tend to only search for the bug in the code that signals the error (i.e., the server's code in this example). Another reason cited as to why distributed programming is harder by some students is that they felt that keeping track of a communication protocol was a lot of work. That is, they had to add communication code to "a lot" of functions and had to write message processing functions.

Finally, students were asked about their level of excitement to develop a multiplayer video game on a scale from 1 (not at all excited) to 5 (extremely excited). The average of the distribution to date is 3.5 with the middle 50% in the range 3-4 and with 76% of the students in the range 3-5. The overwhelming majority of students in the top half of the range clearly indicates that the use of multiplayer video games can serve as great motivation for students to explore distributed programming.

8 Related Work

Teaching distributed programming in CS1 was virtually unheard of a few years ago. Now, there is a growing group of academics attempting it. The developers of DrRacket and HtDP have taught distributed programming in CS1 and have briefly described their approach using a step-locked game[3] to control a UFO [6]. In contrast, the work presented in this article aims to expose students to both distributed programming and to some of its pitfalls like synchronization and communication overhead. Exposing students to such pitfalls is difficult to do with step-lock games like the UFO game [6] and Chat Noir [7]. In addition, the work described in this article can be used by educators "in the trenches" focusing on the actual deployment of a distributed functional video game module in the classroom that is tightly-coupled with other work developed by students during the semester.

[3] A game in which players take discrete turns.

A modest introduction to distributed programming for novices, with some previous exposure to programming, is found in *Realm of Racket* (ROAR) [2]. This book is intended as a general introduction to programming using video games and uses Racket (not the student languages) as the programming medium. ROAR presents the development of a distributed video game, *Hungry Henry*, in which players run around the screen eating cupcakes. Like the work presented in this article, ROAR advocates that distributed programming is a natural part of an introduction to programming. In contrast to the work presented in this article, ROAR exposes readers only to the thick-server model (used in version 2 of Aliens Attack) and does not discuss the pitfalls of distributed programming. The development outlined in ROAR is in the spirit of the design recipe presented in this article, but does not explicitly put forth a design recipe for distributed programming nor does it make explicit how to develop a distributed program through a series of verifiable steps.

The use of functional video games in CS1 is a little more extensive, but still just beginning to flourish. Soccer-Fun, developed using Clean, aims to motivate students by having them write programs to play soccer games [1]. There have been no reported efforts to make the platform distributed in order to allow players to compete against each other nor has this platform been used in CS1. Yampa is a language embedded in Haskell used to program reactive systems such as video games [3]. The use of Yampa in the classroom appears to have been mostly discontinued, but work using functional video games in CS1 [9,10] has sparked an interest to reignite the use of Yampa in education. In previous work, the author presents how to use video games to teach programming using primitive data, structures, and structural recursion [9] and using generative and accumulative recursion [10].

9 Concluding Remarks

Distributed programming ought and can be an integral part of CS1. The need for distributed programming in CS1 is based on the undeniable fact that the use of distributed computing is becoming ubiquitous. The argument for success with distributed computing in CS1 is based on the illustrative development of a non-trivial functional multiplayer video game in SHU's CS1. Not a single function needed for the presented multiplayer game is beyond the ability of students that have studied structural recursion and the associated design recipes in HtDP. One of the major advantages of including distributed functional video game development in CS1 is that students become very excited about programming. There is no doubt that students feel empowered when they can develop a distributed application in a realm that is of interest to them. Another advantage is that students think about programming issues early in their undergraduate years, thus, providing a solid foundation for advanced courses.

Acknowledgements. The author thanks the plt-scheme and the plt-edu mailing list community for the many frank and eye-opening discussions about teaching programming, about HtDP, and about interesting programming projects for

students. The author also thanks Matthias Felleisen for our discussions about distributing programming in CS1, for his feedback on the approach described in this article, and for the "code walk" done with students at Seton Hall University that implemented distributed Aliens Attack.

References

1. Achten, P.: Teaching Functional Programming with Soccer-Fun. In: FDPE 2008, pp. 61–72. ACM, New York (2008)
2. Bice, F., De Maio, R., Florence, S., Lin, F.-Y.M., Lindeman, S., Nussbaum, N., Peterson, E., Plessner, R., Van Horn, D., Felleisen, M., Barski, C.: Realm of Racket. No Starch Press (2013)
3. Courtney, A., Nilsson, H., Peterson, J.: The Yampa Arcade. In: Haskell 2003, pp. 7–18. ACM, New York (2003)
4. Felleisen, M., Findler, R., Fisler, K., Flatt, M., Krishnamurthi, S.: How to Design Worlds (2008), `http://world.cs.brown.edu/1/`
5. Felleisen, M., Findler, R., Flatt, M., Krishnamurthi, S.: How to Design Programs: An Introduction to Programming and Computing. MIT Press, Cambridge (2001)
6. Felleisen, M., Findler, R., Flatt, M., Krishnamurthi, S.: A Functional I/O System or, Fun for Freshman Kids. In: ICFP 2009, pp. 47–58 (2009)
7. Findler, R.: CS 15100 Fall 2008 Project 3: ChatNoir. Dept. of Electr. Engr. and Comp. Sci., Northwestern University (2008), `http://www.eecs.northwestern.edu/robby/uc-courses/15100-2008-fall/proj3. pdf`
8. Findler, R., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: A Programming Environment for Scheme. J. of Functional Programming 12(2), 159–182 (2002)
9. Morazán, M.T.: Functional Video Games in the CS1 Classroom. In: Page, R., Horváth, Z., Zsók, V. (eds.) TFP 2010. LNCS, vol. 6546, pp. 166–183. Springer, Heidelberg (2011)
10. Morazán, M.T.: Functional Video Games in CS1 II. In: Peña, R., Page, R. (eds.) TFP 2011. LNCS, vol. 7193, pp. 146–162. Springer, Heidelberg (2012)
11. Fabien Sanglard. Quake 3 Source Code Review: Network Model (June 2012), `http://fabiensanglard.net/quake3/network.php`
12. Silberschatz, A., Galvin, P.: Operating System Concepts. Addison-Wesley, Reading (1994)

Author Index