Enrico Pontelli
Tran Cao Son (Eds.)

LNCS 9131

Practical Aspects of Declarative Languages

17th International Symposium, PADL 2015
Portland, OR, USA, June 18–19, 2015
Proceedings

Springer

Lecture Notes in Computer Science **9131**

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

More information about this series at http://www.springer.com/series/7408

Enrico Pontelli · Tran Cao Son (Eds.)

Practical Aspects of Declarative Languages

17th International Symposium, PADL 2015
Portland, OR, USA, June 18–19, 2015
Proceedings

Springer

Editors
Enrico Pontelli
New Mexico State University
Las Cruces
New Mexico
USA

Tran Cao Son
Department of Computer Science
New Mexico State University
Las Cruces
New Mexico
USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-19685-5 ISBN 978-3-319-19686-2 (eBook)
DOI 10.1007/978-3-319-19686-2

Library of Congress Control Number: 2015939925

LNCS Sublibrary: SL2– Programming and Software Engineering

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Preface

Declarative languages build on sound theoretical bases to provide attractive frameworks for application development. These languages have been successfully applied to many different real-world situations, ranging from data base management to active networks to software engineering to decision support systems.

New developments in theory and implementation have opened up new application areas. At the same time, applications of declarative languages to novel problems raise numerous interesting research issues. Well-known questions include designing for scalability, language extensions for application deployment, and programming environments. Thus, applications drive the progress in the theory and implementation of declarative systems, and benefit from this progress as well.

PADL is a forum for researchers and practitioners to present original work emphasizing novel applications and implementation techniques for all forms of declarative concepts, including, functional, logic, constraints, etc.

This volume contains the papers presented at PADL 2015: 17th International Symposium on Practical Aspects of Declarative Languages held during June 18–19, 2015, in Portland, Oregon (USA).

Originally established as a workshop (PADL 1999 in San Antonio, Texas), the PADL series developed into a regular annual symposium; the preceding editions took place in San Antonio, Texas (1999), Boston, Massachusetts (2000), Las Vegas, Nevada (2001), Portland, Oregon (2002), New Orleans, Louisiana (2003), Dallas, Texas (2004), Long Beach, California (2005), Charleston, South Carolina (2006), Nice, France (2007), San Francisco, California (2008), Savannah, Georgia (2009), Madrid, Spain (2010), Austin, Texas (2012), Rome, Italy (2013), and San Diego, California (2014).

PADL 2015 was organized by the Association for Logic Programming (ALP), in collaboration with the Organizing Committees of the co-located events at the 2015 ACM Federated Computing Research Conferences, the Department of Computer Science at New Mexico State University, and the Department of Computer Science at the University of Texas at Dallas.

The event received generous sponsorships from the Association for Logic Programming, New Mexico State University, and the National Science Foundation. The organizers are also thankful for the in-cooperation support from ACM SIGPLAN.

Many people contributed to the success of the conference, to whom we would like to extend our sincere gratitude. The members of the Program Committee provided invaluable help in the process of selecting papers and developing the conference program. The numerous referees invested countless hours in reading submissions and providing professional reviews.

Last but not least, we wish to extend our heartfelt thanks to all the authors who submitted their excellent research contributions to the conference.

April 2015

Tran Cao Son
Enrico Pontelli

Organization

Program Committee

Marcello Balduccini	Drexel University, USA
Agostino Dovier	University of Udine, Italy
Matthew Flatt	University of Utah, USA
Marco Gavanelli	University of Ferrara, Italy
Jeff Gray	University of Alabama, USA
Serdar Kadioglu	Oracle America Inc.
Sam Lindley	The University of Edinburgh, UK
Laurent Michel	University of Connecticut, USA
Enrico Pontelli	New Mexico State University, USA
C. R. Ramakrishnan	University at Stony Brook, USA
Norman Ramsey	Tufts University, USA
Ricardo Rocha	University of Porto, Portugal
Claudio Russo	Microsoft Research
Tim Sheard	Portland State University, USA
Tran Son	New Mexico State University, USA
Terrance Swift	CENTRIA, Universidade Nova de Lisboa, Portugal
Neng-Fa Zhou	CUNY Brooklyn College and Graduate Center, USA

Additional Reviewers

Fowler, Simon
Kushner, Sarah
Najd, Shayan

Contents

Ontology-Driven Data Semantics Discovery for Cyber-Security

Marcello Balduccini(✉), Sarah Kushner, and Jacquelin Speck

College of Computing and Informatics, Drexel University, Philadelphia, USA
{mbalduccini,sak388,jspeck}@drexel.edu

Abstract. We present an architecture for data semantics discovery capable of extracting semantically-rich content from human-readable files without prior specification of the file format. The architecture, based on work at the intersection of knowledge representation and machine learning, includes machine learning modules for automatic file format identification, tokenization, and entity identification. The process is driven by an ontology of domain-specific concepts. The ontology also provides an abstraction layer for querying the extracted data. We provide a general description of the architecture as well as details of the current implementation. Although the architecture can be applied in a variety of domains, we focus on cyber-forensics applications, aiming to allow one to parse data sources, such as log files, for which there are no readily-available parsing and analysis tools, and to aggregate and query data from multiple, diverse systems across large networks. The key contributions of our work are: the development of an architecture that constitutes a substantial step toward solving a highly-practical open problem; the creation of one of the first comprehensive ontologies of cyber assets; the development and demonstration of an innovative, non-trivial combination of declarative knowledge specification and machine learning.

Keywords: Data semantics discovery · Ontologies · Machine learning · Cyber-security

1 Introduction

An *ad hoc* data source is a data source for which parsing and analysis tools are not readily available [6]. Even well-documented, established file formats can evolve over time or change with various configuration settings, effectively becoming ad hoc to users who have not followed the changes. Ad hoc data sources present unique challenges for information technologists, cyber-security analysts, and other professionals who must parse and interpret such data for diagnostic or forensics purposes.

We attempt to address the challenges associated with ad hoc file formats through development of an data semantics discovery architecture for extracting semantically-rich content from human-readable files without prior specification

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 1–16, 2015.
DOI:10.1007/978-3-319-19686-2_1

of the file format. The proposed system includes modules for automatic file format identification, tokenization, entity identification, and storage of extracted records and entities. Using a process driven by an ontology of domain-specific concepts, these components interact to parse data from an input file by identifying file format, records and entities within them, and by associating the extracted content with concepts from the ontology. Once data is extracted and stored, the ontology also provides an essential abstraction layer for querying the extracted data, with queries that can span across multiple file systems, file formats and levels of abstraction. In the prototype implementation presented in this paper, the ontology is tailored to cyber-security applications.

Searching for signs of a cyber attack in log files is one practical use for the proposed architecture. Time constraints and lack of documentation can make it difficult to find or create parsers for every log file format encountered, and the magnitude of those challenges increases when dealing with large networks of independent file systems. Security analysts must not only be aware of every type of log available on every network node, but be able to correlate information from multiple sources and reveal important underlying relationships between them. As a motivating example, consider a scenario in which a cyber-security analyst is notified of a new kind of cyber attack following this pattern:

1. A malicious e-mail with an attachment is received somewhere on the network. The sender's e-mail address varies, but it always ends in a ".net" suffix.
2. The recipient of the e-mail opens the attachment, unaware that it is a virus.
3. The virus establishes a DNS (Domain Name Server) tunnel[1] towards a server with the domain name "cyberattacks.com"

In this scenario, an analyst wishes to investigate whether this attack occurred somewhere on his or her network. However, the network includes many nodes, each with their own unique configuration, services, and corresponding log files (see Figure 1). The information is stored using different formats depending on the

```
client 192.168.157.1#5544: query: maliciousserver.com IN AXFR +T
  (192.168.157.129)
```

```
11/21/2013 2:15:59 PM 0A30 PACKET 00000085FD4B0610 UDP Snd
  199.7.91.13 cf1b Q [0000 NOERROR] A (3)www(6)maliciousserver(3)com(0)
11/21/2013 2:16:01 PM 00D4 PACKET 00000085FD4B2320 UDP Snd
  193.0.14.129 aba7 Q [0000 NOERROR] AAAA (3)www(6)maliciousserver(3)com(0)
```

Fig. 1. Sample DNS query records: bind format (top) vs MS DNS format (bottom)

node's specific configuration and softwares used, and understanding the meaning of a log entry requires knowledge that is not explicitly stated in the entry itself (e.g., string "+T" in Figure 1). To make things worse, in realistic circumstances, the analyst often has incomplete knowledge about the attack. In our case, for

[1] http://beta.ivc.no/wiki/index.php/DNS_Tunneling.

instance, the full address of the malicious DNS server and the e-mail address from which the virus originates are both unknown. Although fictitious, this scenario captures many challenges analysts are faced with in actual situations. In particular, the large amounts of data and the disparate, hardly predictable ways in which it may have been encoded make manual browsing of the files unfeasible. Additionally, traditional text-based search, which is relied upon by most state-of-the-art cyber-forensics tools, is also not advisable, as it typically leads to many irrelevant results and forces analysts to a time-consuming and error-prone manual post-processing phase. For example, searching for strings or email addresses (e.g., using regular expressions) with a ".net" suffix across all of the files will likely return matches that have nothing to do with emails received by a mail server, such as records from authentication logs. Furthermore, use of string matching does not allow an analyst to specify additional constraints, such as checking whether other logs indicate that the recipient's computer may have initiated a DNS tunnel to a certain family of servers.

Using our proposed architecture, the analyst can import log files from across the network into a unified knowledge base. The architecture includes modules capable of parsing all log files, regardless of configuration-dependent format variations. Finally, the analyst can ask queries that specify the *types* of information they wish to find, while the system automatically identifies the correct sources and content. This enables searching for signs of the cyber attack using high-level queries that capture the entire attack, rather than having to piece by hand the possible evidence of the individual stages.

The key contributions of our work are: the development of an architecture that constitutes a substantial step toward solving an open problem of high practical importance; the creation of one of the first comprehensive ontologies of concepts related to cyber assets; the development and demonstration of an innovative, non-trivial combination of declarative knowledge specification and machine learning techniques.

The remaining sections of this paper are organized as follows. Background on existing solutions for the problems addressed by the architecture are described in Section 2. Section 3 provides details of each component of the architecture. An experimental evaluation of performance of the machine learning techniques user by the architecture is presented in Section 4. Section 5 concludes the paper and discusses possible directions of future work.

2 Related Work

To the best of our knowledge, our data semantics discovery architecture is the first of its kind. It is worth pointing out that the problem being solved here is substantially different from Natural Language Processing (NLP) and from traditional Information Extraction (IE). The data sources considered here typically lack the grammatical structure considered by NLP and IE. Furthermore, differently from NLP and IE, the meaning of a record frequently depends on the file that contains it – e.g., line "03/08/2015 10.0.0.1" describe very different events

depending on whether it is found in a web server log file or in a DNS server log file. For the most part, earlier and ongoing research has studied sub-problems addressed by our architecture.

The problem of describing knowledge related to cyber-security scenarios is the object of various proposed specifications, such as STIX, CybOX, MAEC.[2] However, none of them provides a comprehensive and hierarchical description of the software and hardware components of a system, covering operating system objects and events.

Aggregating data from multiple file systems can help network administrators detect network problems or diagnose potential causes of earlier problems. Varying file formats and data schemas can complicate these tasks. Doan, et al. present Learning Source Descriptions (LSD), a system for reconciling schemas from disparate data sources using machine learning [5]. LSD learns semantic mappings between multiple XML data sources, employing and extending established machine learning techniques. LSD incorporates user feedback to improve the accuracy of the mappings.

Splunk is a tool for aggregating massive heterogeneous datasets of log file text into a semistructured time series database [3]. It claims to accept logs in "any" format, and allows full text searches across various data sources via its own query language. The decision to aggregate data into a time series database was motivated in part by the fact that time stamps are one of the only common fields among many different types of log data, and contain essential information for many types of analysis (including cyber-forensics). Splunk exploits this time series organization during searches, operating on only the time slices that intersect the query target time. The Splunk query language supports a wide range of complex functionality, including data mining techniques such as clustering, anomaly detection, and prediction.

The presence of log file formats unfamiliar to network analysts often complicates their diagnosis of system failures and vulnerabilities. SherLog is a diagnostic tool capable of reverse-engineering log file formats. However, SherLog is limited to single file systems and only applies to log files produced by specific, known executable programs [15].

Tupni, another tool for reverse-engineering both protocol and file formats, expands beyond simple data types, extracting record types, record sequences, and input constraints [4]. However, Tupni requires both a sample file and an application capable of parsing the file as input. The tool therefore can not support ad hoc data sources, which have no readily available parsing tools. Splunk, an aggregation tool discussed above, supports ad hoc formats in the sense that users may configure arbitrary input types. However, it does not automatically learn how to parse these input types. While Splunk does not require users to specify a schema for the data to be indexed, users must specify fields and values to extract. It includes tools that guide the user through creating regular expressions to extract fields and values for each incoming time-delineated event.

[2] https://stix.mitre.org/.

Tokenization and entity extraction form ad hoc data sources is one of the core problems related to data semantic discovery. In-depth studies of the log analysis process have found that non-technical users increasingly need data from log files, but code development knowledge is a beneficial or even necessary prerequisite to log file understanding [1,12]. A lack of documentation can also create difficulties for technical users, who often have to sort through program code in order to discover what information is logged. Technical and non-technical users would benefit from tools that can automatically extract, categorize, and assign semantic meaning to tokens from log files.

DEC0DE is a tool for recovering information from mobile phones with unknown storage formats, to aid in criminal investigations [13]. The tool compares small blocks of unparsed data to a library of known hashes in order to reveal information of interest, then parses the remaining data with adapted NLP techniques. Fisher, et al. introduced LearnPADS, an end-to-end system for generating data processing tools directly from ad hoc data [6–8]. It employs a multi-phase algorithm for inferring the structure of ad hoc data sources and generating templates in the PADS data description language. The data itself is then used to generate a semistructured query engine, format converters, statistical analyzers, and visualization routines, without human intervention. The system has similar goals to our work, but does not include a method for storing and retrieving previously-recognized formats, which would prevent repeating the structure-inference process every time a particular ad hoc structure is encountered. Furthermore, unlike our work, the LearnPADS system is not capable of inferring higher-level relationships from the available data in order to establish links between information from multiple files, possibly across multiple file systems (e.g., to allow a user to ask "show me all incoming traffic from source IP 10.0.0.10"). Another drawback is the PADS language itself, which requires users to provide a priori knowledge of the data formats present in the data set to be analyzed. This means that LearnPADS can support ad hoc file formats, but not ad hoc entity strings as our proposed architecture can.

After learning to parse and extract tokens from an ad hoc file format, it is necessary to assign meaning to the extracted entities. Splunk relies on the user to specify the semantic meaning of extracted entities that it does not already recognize. Seaview uses fine-grained type inference to generate log file visualizations based on the semantic meanings (e.g., "Student ID" as opposed to "Integer" or "String") of extracted tokens [9]. Seaview infers semantic relationships between fields in log files, but does not represent record types or file types as our architecture does.

FlashExtract is a newer framework for extracting data from ad hoc documents using examples [10]. However, extraction is performed on a per-file basis, requiring users to highlight examples in every individual input file instead of generating parsing templates for previously-seen formats. FlashExtract also does not utilize an ontology to define semantic relationships between data entities.

3 The Data Semantics Discovery Architecture

Figure 2 shows the proposed data semantics discovery and the relationships between its components.

The system includes a software modules for File Format Identification, Template Generation, Parsing, Data Storage, and Querying. The user provides an ontology of domain-specific concepts and their relationships, which captures the concepts and the levels of abstractions that the user expects to later use for querying. All of the classes that one would expect for such a domain-specific ontology are specified, including concepts file types, record types, entity types, but also processes, files, system queues, etc. In addition to these typical components, concepts descriptions in the ontology also include, whenever available, a specification of training data in the form of collections of labeled samples.

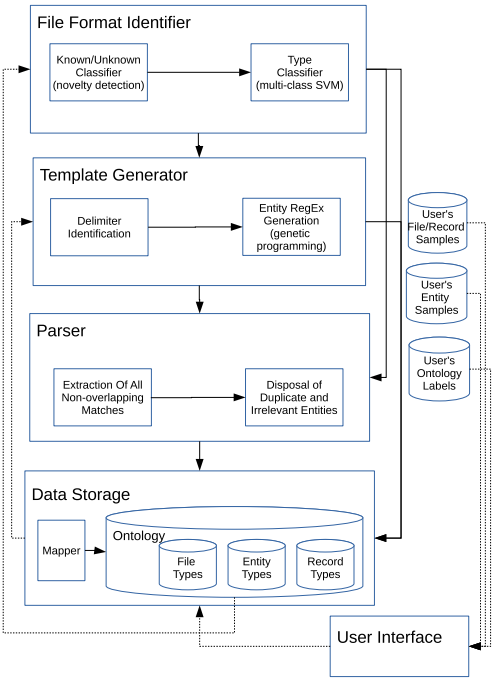


Fig. 2. Proposed architecture

An overview of each component is provided below. However, any architectural component can be modified or replaced to better suit other applications.

In order to simplify the process, our work relies on a few assumptions. The Template Generator and Parsing components assume that alphanumeric characters are never used as delimiters between tokens, and only non-alphanumeric characters may serve as delimiters. However, the Template Generator lifts this assumption, as non-alphanumeric characters are often part of data entities (e.g., "." as part of an IP address token). While the Template Generator does not assume structure to be homogeneous throughout the entire input file, it assumes that the file contains at least one group of lines that can be parsed using each set of delimiters. The process for identifying delimiters is described in greater detail in later. Tokens from the same column in any given line group are assumed to represent the same entity type (e.g., if the token before the first delimiter on a line represents a timestamp, all lines that are formatted the same way contain a timestamp in that position). While all entities appearing on a given line of text are considered to be part of the same record, their order is not considered for interpretation of the record.

The architecture first identifies the format of the input using supervised machine learning. The result dictates whether the system attempts to retrieve the corresponding parsing template, or create one if no such template exists. The Parser extracts tokens to match per-entity regular expressions in the template, then disposes of duplicate or irrelevant entries. Finally, the extracted tokens are classified by entity type (e.g., "Date-Time," "IPV4," etc.), and mapped to the ontology as complete records. If it encounters information that does not correspond to existing ontology classes, the system is capable of adding new classes to the ontology. This feature is, however, beyond the scope of the present paper and will be discussed separately.

3.1 Ontology

In the proposed architecture, an ontology provides domain knowledge about cyber assets, associates type labels and samples, and stores the data extracted from the files. The domain-specific knowledge contained in the ontology is of the type that is found in textbooks or manuals. This information is specified at development time, and we expect that it is sufficiently general to be sufficient for most scenarios and applications, but obviously it can also be easily extended at run-time. The two top-level classes of the ontology are events and objects, described in more details next.

Events: this class is used to describe host-level events. The data semantics discovery architecture views log files as collections of records of events, with each log file potentially including multiple types of records. The sub-classes of events include:

- *Hardware Events*: events that occurred at the hardware level. This category contains sub-classes for events such as overheating, physical disk damage, peripherals connected/disconnected, etc.
- *OS Events*: events relevant to the kernel, communications layer, software processes, and user actions. Each is described by a different sub-classes, and further divided as appropriate.

All event records identified by the system will be (direct or indirect) sub-classes of the above. The latter class encompasses the largest set of sub-classes, and it is likely that most ad hoc log formats encountered by users will describe events from that category.

Objects: this class represents basic data entities, such as email addresses and network addresses, as well as physical and software objects. Intuitively, events result from actions performed by objects and/or on objects. The ontology includes all objects related to events the user wishes to monitor through log file records. Sub-classes capturing specific object types, including:

- *Hardware Objects*: physical components of a computer, such as a keyboard or a video card.
- *OS Objects*: software objects for which the OS is directly responsible, such as processes, threads, memory; also, objects that exist within, or are created by, applications.

Sub-classes of OS Objects include DHCP tables, which are maintained by the DHCP service, and email messages, which are handled by the email daemon. Files and directories are also OS Objects, and the various types of log files are further sub-classes of files.

The links among the various objects and between objects and events are expressed using a few general *properties*. For example, properties allow the system to memorize in which log file a record was found. Some properties apply to whole classes of the ontology, while others are specific to instances of classes containing the information extracted from log files.

The most important property in the first category is `trainingSamples`. This property is applicable to any class and specifies a path the file(s) containing samples of that class to be used for classifier training. Samples are used as training data for extracting and identifying information from the data sources (see below). Properties that are applied to specific instances include:

- `in-file`: applicable to any event record (see *Events* class), this property allows to specify in which log file the record was.
- `contains`: this property is applicable to event records as well, and is used to specify which data entities were found in that record. For example, many record types contain a date-time entity.
- `text`: this property is applicable to any data entity. It is used to specify the text that was identified as describing that data entity. For example, the value of the `text` property for an *IPv4Address* data entity could be "10.0.0.1."

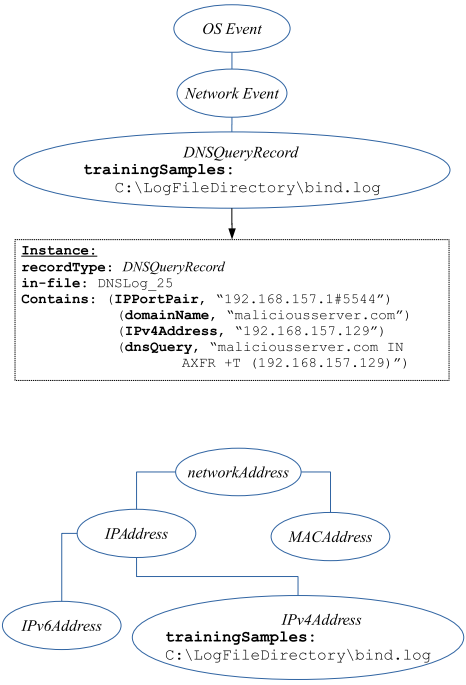


Fig. 3. Storage of a DNS Query Record

To see how the information from data sources is encoded by the architecture, consider the sample DNS query record from Figure 1 (top), a special case of a DNS record that a user might search for in the example from Section 1.

Examples of key ontology elements pertinent to the identification and storage of this record are illustrated in Figure 3. Property `trainingSamples` of classes *DNSQueryRecord* and *IPv4Address* provides the location of the training samples for the extraction algorithms. Parsing of the record, performed using the algorithms described later, creates a new instance of the *DNSQueryRecord* class.

This class is categorized as a *Network Event*, which is a sub-class of *OSEvent*. A reference to the file in which the record was found is memorized by the instance's `in-file` property. Finally, the components of the record identified by the parsing algorithms are stored as instances of appropriate data entity classes and linked to the *DNSQueryRecord* instance via its `contains` property. For illustrative purposes, here we visualize them as pairs of entity types and values:

- (*IPPortPair*, "192.168.157.1#5544")
- (*domainName*, "maliciousserver.com")
- (*IPv4Address*, "192.168.157.129")
- (*dnsQuery*, "maliciousserver.com IN AXFR +T (192.168.157.129)")

3.2 File Type Identification

Machine learning approaches for automatic classification can be divided into two broad categories: supervised and unsupervised. Supervised methods both compare unlabeled input samples to a set of labeled training samples. Unsupervised approaches require no training data, instead labelling unknown input samples by searching for hidden structures in the data set. A supervised approach best suits our goal of categorizing formats according to labels from an ontology, although exploration of semi-supervised approaches that require fewer labeled input samples is a goal for future work (see Section 5). The File Format Identification component of our architecture identifies file types using a two-stage approach:

- Determining if file type is known: Has the system previously seen files of the same type as the given input file? More generally, does this file contain entities that the system is capable of identifying?
- Identifying the file type: If the given input file is of a known type, which known type is it?

In order to train both classifiers, the architecture extracts training data from the samples provided by the ontology, as discussed above.

The first stage of File Format Identification determines whether the input file is of a "known" or "unknown" (i.e., not previously seen) type using a One-Class Support Vector Machine (SVM) classifier for Novelty Detection. The One-Class SVM is an adaptation of the traditional pairwise SVM, which determines whether or not observed data points come from the same distribution by classifying them as "in-distribution" or "outliers" [11]. Given a set of initial observations from the same distribution, each described by p features, the classifier learns a contour enclosing the distribution in p-dimensional space. If new observations lay within the contour, they are considered to come from the same population as the initial observation. If they lay outside the contour, they are considered "outliers" belonging to some other distribution. We train a One-Class SVM to recognize all files for which the system has data samples as "known" (i.e., "in-distribution"). The second stage applies to input files classified as "known" during the first stage, and determines which known file type the input corresponds to. Training

data for this stage includes labels for each known file format, which coincide with the corresponding class names from the ontology. For the second stage of File Format Identification, we combine several "traditional" pairwise SVM classifiers to create a multi-class classifier [14].

From a technical perspective, both classification stages of the file type identification process use, as features for the learning algorithms, n-grams of space-delimited tokens. In early evaluations, a combination of tri-grams and 4-grams produced the best performance with over 99% accuracy for the second stage of File Format Identification. To reduce dependency on the appearance of specific string values, we perform pre-processing to replace characters with generic *character type labels*, i.e. numeric characters are replaced by character "N" and alphabetic characters are replaced by "A." Punctuation characters are left as-is because they are often important features of specific entities (e.g., "." in the IP address "192.168.1.1").[3]

As an example of file type identification in the prototype implementation of this system, consider a file being analyzed for the motivating use case introduced in Section 1. An excerpt from the file is shown in Figure 1 (bottom).

After extracting features, we use novelty detection to determine whether the file is of a recognized type. The file falls within the distribution of recognized samples, and is classified as being of a known type. The second classification stage compares the file to each class of known files, and identifies it as a MS DNS log.

3.3 Template Generation and Parsing

When it encounters an unrecognized file format, the architecture uses structural cues from the file to generate a parsing template. For the prototype implementation presented here, Template Generation is a two-stage approach, consisting of Delimiter identification, which identifies groups of lines that are parsed similarly, and then identifies the delimiters in each line group, and Regular Expression Generation, which forms regular expressions for the entities in each line group after separating tokens in each line group using the delimiters identified. Both steps are detailed next. The output of this process is a template for parsing the input file. The template contains a set of regular expressions for matching each entity contained in the file.

Delimiter Identification. Many log file formats are "homogenous," meaning that all lines contain the same fields and use the same delimiter. Examples of homogenous log file formats include Comma-Separated Values (CSV) files and Tab-Separated Values files. For these log files, identifying the delimiter is relatively straightforward. However, some log files include a variety of different line formats, with varying delimiters, field types, and even numbers of fields on each line. To account for these files, Template Generation begins by grouping

[3] The evaluation of the effects of the replacement by character type labels and a comparison with other possible approaches are in progress and will be discussed in a separate article.

lines in the input file that are formatted the same way. The remaining steps in the Template Generation procedure are applied separately for each line group.

We have explored several methods for identifying line groups, largely based on heuristics:

- Clustering by whitespace: lines with the same number of whitespace characters are clustered together.
- Clustering by all punctuation characters: similar to the first proposed method, but considering the weighted counts of all punctuation characters.
- Clustering by alphabetic and numeric characters: considering weighted counts of all types of characters.

In preliminary experiments with the prototype implementation, the first method produced accurate templates for the files relevant to the scenarios of interest.

Delimiters are identified for each line group. We first identify candidate delimiters by counting the number of appearances of each punctuation character on each line in the given line group, and counting the number of characters between appearances on each line. The delimiter for the line group is the candidate that meets the following criteria:

- Has the minimum standard deviation in its per-line count.
- Has the largest standard deviation in the character distance between its appearances (only applies if multiple characters meet the first criterion).

The first criterion is motivated by the assumption that all lines within the same group contain the same number of fields. If this is the case, a delimiter character should appear the same number of times on each line. However, we allow for some variation in the count in case the character also appears within a nested entity. The second criterion only applies if multiple characters meet the first criterion, and is motivated by the experimental observation that few entity types have fixed character lengths.

Regular Expression Generation. Once data entities in a given line group have been identified, by the process of elimination after identifying delimiters, the Template Generator learns Regular Expressions representing each column of tokens (i.e., tokens before the first delimiter in a line group, tokens between the first and second delimiters, etc.) extracted from the input file. To generate regular expressions that match each column of tokens from a given line group, the prototype implementation uses a Genetic Programming algorithm similar to the one described in [2]. Extracted tokens are first pre-processed to replace alphanumeric characters with "A" or "N." The pre-processed tokens are used as inputs to the Genetic Programming algorithm, from which we obtain a regular expression that best fits all of the tokens from the column.

The templates generated by this process contain lists of regular expressions for each line group. This substantially simplifies the parsing process, allowing to reduce it to the task of extracting strings using regular expressions.

A particularly challenging case for Template Generation and Parsing is that of user-configurable file formats, such as that of Apache web server logs. The

softwares that generate these logs provide users practically complete freedom in the specification of which of the available data entities should appear in the logs, and in which order. The Template Generation and Parsing component accommodates these types of files by adopting two strategies:

- Allowing multiple templates for each file type: When multiple templates exist, the Parser attempts to extract entities using all of them. The template with the largest number of recognized entities extracted is considered to be "correct" for our purposes.
- Generating a new template if too few recognized entities are extracted by the Parser: When very few of the extracted tokens represent recognized entity types, the system generates a new template and stores it with the existing template(s) for the input file format. We apply a threshold for percentage of recognized entity types to determine whether a new template should be created.

3.4 Ontological Mapping

After parsing, the extracted tokens, records, and the files themselves are mapped to the ontology, as follows.

The mapping algorithm begins with Entity Identification, in which tokens extracted during the parsing process are mapped to data entities from the ontology. The process is similar to that of File Type Identification: using the classes and samples from the ontology as training data, first we determine if the input token is a known entity type, then determine which type it is. We use SVMs for this classification problem as well, with features extracted by applying the same replacement by character type labels described earlier and by creating a count vector of the character types (letters, numbers, and punctuation) for each token.

Once the extracted entities have been identified, combinations of entities are stored as records as in the example from Section 3.1. The Mapper module must determine which type of event record object to create (see Section 3.1 for an overview of event types from the built-in ontology).

As before, training data for the record classification task is obtained from the ontology, which specifies training samples and whose class names are used as labels. The output of the File Format Identification module and the results of the Entity Identification process described above are combined to form a feature vector for supervised classification using SVMs.

The first element of the feature vector contains the file format label for the input file, or the special label "UNKNOWN" if the file format was not recognized. The remaining elements contain the count of every known entity type found during Entity Type Identification, including the number of unrecognized entities. For the kind of data considered here, the prototype implementation has shown good results with this feature representation, which ignores the order of entities' appearance in each record.

Finally, a supervised learning algorithm similar to the algorithm described in Section 3.2 is used to classify the record as one of the known types from the ontology.

3.5 Querying

Once the information has been extracted and stored, the analyst can leverage the hierarchical organization of the ontology to ask queries that span across multiple files and are independent of how the information was originally encoded. In the case of the motivating example from Section 1, our architecture enables the analyst to check for instances of the cyber attack of interest by posing the following query, encoded here in a simplified pseudo-language to simplify the presentation:

```
SELECT R1, R2, R3 WHERE
R1 is a mailRecord,
  R1.contains (emailAddress, .net),
  R1.contains (DateTime D1),
R2 is a dnsQueryRecord,
  R2.contains (DateTime D2),
  D2 > D1,
  R2.contains (domainName, *cyberattacks.com),
  R2.contains (networkAddress, victimPC),
R3 is a dnsQueryRecord,
  R3.contains (DateTime D3),
  D3 > D2,
  R3.contains (domainName, *cyberattacks.com),
  R3.contains (networkAddress, victimPC)
```

The first four lines of the query identify receipt of an e-mail from a ".net" e-mail address and the remaining lines identify two subsequent DNS queries to the malicious server, both occurring on the same network node. The query also requires that the email arrival precedes the DNS queries (conditions $D2 > D1$ and $D3 > D2$). The first line of the query specifies that the corresponding records must be returned, although of course it would be easy to also return, for example, the date times and address of the victim, or the complete sender e-mail address.

Such a query can be easily expressed in a state-of-the-art query language such as SPARQL; for example the following shows how the first 4 lines are translated:

```
SELECT ?r1 ?r2 ?r3 WHERE {
    ?r1 rdf:type dsd:mailRecord .
    ?r1 dsd:contains ?e1 .
    ?e1 rdf:type dsd:EmailAddress .
    ?e1 dsd:text ?addr .
    FILTER (REGEX(str(?addr), ".net")) .
    ?r1 dsd:contains ?d1 .
    ?d1 rdf:type dsd:DateTime .
    ...
}
```

Queries can even be built automatically from a higher-level specification, which for example could be part of a library of known cyber attacks.

It is important to stress the practical advantage of the design of the architecture for high-level query answering. In the case of the present example,

analyst can identify which network node opened the DNS tunnel regardless of how the DNS queries were actually logged by the server(s). In fact, depending on a server's configuration, the information in the log files might identify network nodes by their IPv4 addresses, their IPv6 addresses, or even their MAC addresses. However, because class *networkAddress* is a super-class for *IPv4Address*, *IPv6Address*, and *MACAddress* (see Figure 3), the use of the *networkAddress* class in the query encompasses all three network address types. This additional level of abstraction allow analysts to disregard irrelevant low-level details as needed.

4 Supervised Learning Evaluation

Successful identification and storage of known data types depends on the effectiveness of supervised learning as described in Section 3. In this section, we report on an empirical evaluation of the learning components of our architecture to enable comparison with future approaches. We evaluate the performance of the supervised learning modules for file format, entity, and record classification with ten cross-fold validations. The data used for this evaluation consist of 2,022 text files from 29 file classes, which contain a combined 12,622 distinct record samples from 22 record classes. The data for entity classification consist of 291 entity samples from 11 classes. The number of entity samples is small compared to the number of record and file samples because we have accounted for duplicates removed during the feature pre-processing step by replacement by character type labels described earlier. Recall that this pre-processing replaces specific alphanumeric characters with character type representations, reducing the number of unique samples required. For all classification problems, the number of samples was distributed as close to uniformly as possible across all classes.

Common performance metrics for supervised learning include *precision*, the fraction of retrieved instances that are relevant, *recall*, the fraction of relevant instances that are correctly retrieved, and *f-measure*, the harmonic mean of precision and recall. Each metrics ranges from 0 to 1, with 1 being the best possible score. The average metrics over all ten cross-validation folds are shown in Table 1. Although the performance is, overall, satisfactory, a discussion of possible ways to improve it is provided in the next section. SVMs were chosen for all three classification problems because they outperformed several other classifier types in preliminary evaluations, but further evaluations of various classifier and feature combinations will be included in future work (see Section 5).

Table 1. Supervised learning performance

	Precision	**Recall**	**F-Measure**
File Format Identification	0.9791	0.9808	0.9799
Record Type Identification	0.835	0.8438	0.8394
Entity Type Identification	0.8279	0.7819	0.8042

5 Conclusions and Future Work

We have presented a data semantics discovery architecture capable of parsing and interpreting data from multiple ad hoc data sources, and of correlating information from multiple sources regardless of the format and level of abstraction at which the information was originally encoded. The extraction process is effectively driven by an ontology of domain-specific concepts, which provides samples and labels for the underlying algorithms. The same architecture is also used for answering queries about the extracted data.

Our architecture moves beyond parsing techniques requiring prior knowledge of file formats and is a step toward parsing data sources with completely arbitrary formats. Any component of the architecture can be adjusted or replaced to better suit a user's needs or to perform comparative studies of alternative techniques. The evaluation of the architecture in realistic conditions is under way.

From a practical point of view, our architecture improves upon existing search methods common in cyber-security tools by adding a layer of semantic understanding of the extracted data via an ontology, which allows a user to ask higher-level queries and at the same time tends to return more relevant results than the string-based search methods used by most cyber-security tools.

This paper presented the overall architecture and described its use. Next, we plan to study how the performance of the system (e.g., execution time, accuracy) is affected by the adoption of different techniques for the implementation of its various components. For example, different combinations of features or use of ensemble methods may improve classification performance over the metrics presented in Section 4. In turn, improving classification performance may benefit overall performance because the system naturally depends on accuracy of classification during insertion of data into the knowledge base.

The use of the ontology may help to compensate for misclassifications or ambiguities between related low-level data types. If, for example, an IPv4 address is misclassified as an IPv6 address, it will still be identified as network address and will be returned in response to queries for entities of that type. Exploration and quantitative evaluation of this idea are another subject for future work.

Finally, although we have discussed our architecture in a cyber-security context, we believe it to be applicable to a wide range of domains by simply providing an appropriate ontology and training samples. Verification of this claim will be another direction of future work.

Acknowledgments. The authors would like to thank Philip J. Yoon for useful discussions on the topic of ad hoc data sources.

References

1. Alspaugh, S., Chen, B., Lin, J., Ganapathi, A., Hearst, M., Katz, R.: Analyzing log analysis: an empirical study of user log mining. In: Conference on Large Installation System Administration (LISA) (2014)

2. Bartoli, A., Davanzo, G., De Lorenzo, A., Mauri, M., Medvet, E., Sorio, E.: Automatic synthesis of regular expressions from examples with genetic programming. In: Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (2012)
3. Bitincka, L., Ganapathi, A., Sorkin, S., Zhang, S.: Optimizing data analysis with a semistructured time series database. In: Proceedings of the 2010 Workshop on Managing Systems Via Log Analysis and Machine Learning Techniques (SLAML 2010) (2010)
4. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. ACM (2008)
5. Doan, A., Domingos, P., Halevy, A.Y.: Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. ACM SIGMOD Record **30**(2), 509–520 (2001)
6. Fisher, K., Walker, D., Zhu, K.Q., White, P.: From dirt to shovels: fully automatic tool generation from ad hoc data. ACM SIGPLAN Notices **43**(1), 421–434 (2008)
7. Fisher, K., Walker, D., Zhu, K.Q.: LearnPADS: automatic tool generation from ad hoc data. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1299–1302 (2008)
8. Fisher, K., Walker, D.: The PADS project: an overview. In: Proceedings of the 14th International Conference on Database Theory. ACM (2011)
9. Hangal, S.: Seaview: Using Fine-Grained Type Inference to Aid Log File Analysis (2011)
10. Le, V., Gulwani, S.: FlashExtract: a framework for data extraction by examples. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (2014)
11. Scholkopf, B., et al.: Estimating the support of a high-dimensional distribution. Neural Computation **13**(7), 1443–1471 (2001)
12. Shang, W., Nagappan, M., Hassan, A.E., Jiang, Z.M.: Understanding log lines using developmental knowledge. In: 2014 IEEE International Conference on Software Maintenance and Evolution (2014)
13. Walls, R.J., Learned-Miller, E.G., Levine, B.N.: Forensic triage for mobile phones with DEC0DE. In: USENIX Security Symposium (2011)
14. Wu, T.-F., Lin, C.-J., Wen, R.C.: Probability estimates for multi-class classification by pairwise coupling. JMLR **5**, 975–1005 (2004)
15. Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., Pasupathy, S.: SherLog: error diagnosis by connecting clues from run-time logs. ACM SIGARCH Computer Architecture News **38**(1), 143–154 (2010)

State Space Planning Using Transaction Logic

Reza Basseda(✉) and Michael Kifer(✉)

Stony Brook University, Stony Brook, New York, NY 11794, USA
{rbasseda,kifer}@cs.stonybrook.edu

Abstract. State space planning algorithms have been considered as one of the main classical planning techniques to solve classical planning problems since 1960. In this paper, we show that Transaction Logic is an appropriate language and framework to study and compare these planning algorithms, which enables one to have more efficient planners in logic programming frameworks. Specifically, we take *STRIPS* planning and forward state space planning algorithms, and show that the specification of these algorithms in Transaction Logic lets one implement complicated planning algorithms in declarative programming languages (e.g. Prolog). We first provide a formal representation of these planning algorithms in Transaction Logic, which can be used to automatically translate *STRIPS* planning problems in PDDL to Transaction Logic rules. Then, we use the resulting Transaction Logic rules to solve planning problems and compare the performance of those algorithms in our simple interpreter implemented in XSB Prolog. We use several case studies to show how the linear *STRIPS* planning algorithm is faster than forward state space search. Our experiments highlight the fact that a planner implemented by logic programming framework can become faster if an appropriate planning algorithm is applied.

Keywords: Declarative planning algorithms · Planning in logic programming · State space planning

1 Introduction

The classical automated planning has been used in a wide range of applications such as robotics, multi-agent systems, and more. This wide range of applications has made automated planning one of the most important research areas in Artificial Intelligence (AI). The history of using logical deduction to solve classical planning problems in AI dates back to the late 1960s when situation calculus was applied in the planning domain [30]. There are several planners that encode planning problems into satisfiability problems [35][29][17] or constraint satisfaction problems (CSP) [47][18][2] and use logical deduction to solve the planning problems. Beside planning as satisfiability and CSP, a number of deductive planning frameworks have been proposed over the years. Linear connection proof method [6][7][8], equational horn logic [32], and linear logic [34][15]

This work was supported, in part, by the NSF grant 0964196.

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 17–33, 2015.
DOI:10.1007/978-3-319-19686-2_2

are well-known examples of logic-based deduction methods applied for solving classical planning problem. Answer set programming is another, more recent logic based technique to solve planning problems [36][26][24][44].

There are several reasons that make logical deduction suitable to be used by a classical planner [45][16]: (1) Logical deduction used in planning can be cast as a formal framework that eases proving different planning properties such as completeness and termination. (2) Logic-based systems naturally provide a declarative language that simplifies the specification of planning problems. (3) Logical deduction is usually an essential component of intelligent and knowledge representation systems. Therefore, applying logical deduction in classical planning makes the integration of planners with such systems simpler. In-depth discussion of these reasons is beyond the scope of our paper. Despite the benefits of using logical deduction in planning, many of the above mentioned deductive planning techniques are not getting as much attention as algorithms specifically devised for planning problems. There are several reasons for this state of affairs:

- Many of the above approaches invent one-of-a-kind techniques that are suitable only for the particular problem at hand. For instance, the effects or preconditions of actions are sometimes encoded indirectly in answer set programming planners. This makes the encoding of planning problem difficult, and thus reduces the generality of this technique.
- These works generally show how they can represent and encode classical planning actions and rely on a theorem prover of some sort to find plans. Therefore, the planning techniques embedded in such planners are typically some of the simplest state space planning strategies (e.g. forward state space search) and they have extremely large search space. Consequently, they cannot exploit heuristics and techniques invented by different classical and neoclassical planning technique to reduce the search space.

In this paper, we show that a general logical theory, called *Transaction Logic* (or $\mathcal{TR}$) [12–14], addresses the above mentioned issues and also provides multiple advantages for specifying, generalizing, and solving planning problems. Transaction Logic is an extension of classical logic with dedicated support for specifying and reasoning about actions. To illustrate this point, [5] has shown how state space planning techniques, such as *STRIPS* (also known as goal-stack state space planning) and forward state space planning algorithms, can be naturally represented and improved upon using Transaction Logic. Since planning techniques are cast here as purely logical problems in a suitable general logic, a number of otherwise non-trivial further developments became low-hanging fruits and were gotten almost for free. In the present paper, based on the aforesaid representations of planning algorithms in Transaction Logic, we develop a simple translator that maps *STRIPS* planning problems (specified in PDDL [1]) and planning algorithms to Prolog programs. This technique makes many already existing Prolog based planners [4][49][3][48] more efficient. We emphasize that this paper, unlike [5], does not propose new planning algorithms in Transaction Logic. Instead, we use the sequential subset of Transaction Logic (i.e., without concurrent transactions) to represent the linear *STRIPS* planning algorithm [19],

as suggested in [11]. A more computationally complex strategy was proposed in [5]. It deals with non-linear *STRIPS* and *Concurrent* Transaction Logic, and it was shown to be complete. The present paper, in contrast, just shows how planning algorithms written in Transaction Logic can be simply mapped into Prolog rules.

The next section briefly characterizes a planning problem and overviews Transaction Logic. Section 3 explains how we formally encode planning techniques in $\mathcal{TR}$. Section 4 also provides the results of our simple experiments to illustrate the practical applications of this method. Section 5 describes the relation of this work to PDDL and other research on planning with logic. The last section concludes our paper.

2 Characterization of a Planning Problem

In this section, we briefly remind the reader the basic concepts of logic and formally define an extended STRIPS planning problem. Then we briefly overview Transaction Logic ($\mathcal{TR}$) [11].

2.1 *STRIPS* Planning Problem

In a *STRIPS* planning problem, actions update the state of a system (e.g. Knowledge-Base): Facts may be inserted into or removed from the state as a result of execution of an action. We assume denumerable sets of variables $\mathcal{X}$, constants $\mathcal{C}$, and disjoint sets of predicate symbols, extensional ($\mathcal{P}_{ext}$) and intensional ($\mathcal{P}_{int}$) ones. A *term* is a variable or constant. Extensional (resp. intensional) ***Atoms*** have the form $p(t_1, ..., t_n)$, where t_i is a term and $p \in \mathcal{P}_{ext}$ (resp. $p \in \mathcal{P}_{int}$). A ***ground*** atom is a variable free atom. A ***literal*** is either an atom or a negated extensional atom, $\neg p(t_1, ..., t_n)$. Note that negative intensional atoms are not literals. A substitution θ is a set of expressions of the form $X \longleftarrow c$, where $X \in \mathcal{X}$ and $c \in \mathcal{C}$. Given a substitution θ, an atom $a\theta$ is obtained from atom a by replacing its variables with constants according to θ.

Intensional predicate symbols are defined by *rules*. A rule r, shown as $head(r) \leftarrow b_1 \wedge \cdots \wedge b_n$, consists of an intensional atom $head(r)$ in the head and a conditional body, a (possibly empty) conjunction of literals $b_1, \ldots, b_n$, where $b_i \in body(r)$. A ***ground instance*** of a rule, $r\theta$, is any rule obtained from r by a substitution of $head(r)$ and $body(r)$ with ground atoms $head(r)\theta$ and $body(r)\theta$ respectively. Given a set of literals $\mathbf{S}$ and a ground rule $r\theta$, the rule is *true* in $\mathbf{S}$ if either $head(r)\theta \in \mathbf{S}$ or $body(r)\theta \not\subseteq \mathbf{S}$. A (possibly non-ground) rule is *true* in $\mathbf{S}$ if all of its ground instances are true in $\mathbf{S}$. A ***fact*** is a ground extensional atom that can be inserted or deleted by *STRIPS* actions. A set $\mathbf{S}$ of literals is ***consistent*** if there is no atom, a, such that $\{a, \neg a\} \subseteq \mathbf{S}$.

Definition 1 (State). *Given a set of rules* $\mathbb{R}$*, a consistent set* $\mathbf{S}$ *of literals is called a* **state** *if and only if*

1. *For each fact a, either, $a \in \mathbf{S}$, or $\neg a \in \mathbf{S}$.*
2. *Every rule of $\mathbb{R}$ is true in $\mathbf{S}$.*

Definition 2 (*STRIPS* action). *A STRIPS* **action** $\alpha = \langle p_\alpha(X_1, ..., X_n), Pre(\alpha), E(\alpha)\rangle$ *consists of an intensional atom $p_\alpha(X_1, ..., X_n)$ in which $p_\alpha \in \mathcal{P}_{int}$ is a predicate that is reserved to represent the action α and can be used for no other purpose, a set of literals $Pre(\alpha)$, called the* ***precondition*** *of α, and a consistent set of extensional literals $E(\alpha)$, called the* ***effect*** *of α. The variables in $Pre(\alpha)$ and $E(\alpha)$ must occur in $\{X_1, ..., X_n\}$.*

Note that the literals in $Pre(\alpha)$ can be both extensional and intensional, while the literals in $E(\alpha)$ can be extensional only.

Definition 3 (Execution of a *STRIPS* action). *A STRIPS action α is* ***executable*** *in a state $\mathbf{S}$ if there is a substitution θ such that $\theta(Pre(\alpha)) \subseteq \mathbf{S}$. A* ***result of the execution*** *(with respect to θ) is the state $\mathbf{S}'$ such that $\mathbf{S}' = (\mathbf{S} \setminus \neg\theta(E(\alpha))) \cup \theta(E(\alpha))$, where $\neg E = \{\neg\ell | \ell \in E\}$.*

Note that $\mathbf{S}$ is well-defined since $E(\alpha)$ is consistent. Observe also that, if α has variables, the result of an execution, $\mathbf{S}$, may depend on the chosen substitution θ.

Definition 4 (Planning problem). *Given a set of rules $\mathbb{R}$, a set of STRIPS actions $\mathbb{A}$, a set of literals G, called the* ***goal****, and an* ***initial state*** *$\mathbf{S}$, a* **planning solution** *(or simply a* ***plan****) for the planning $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$ is a sequence of ground actions $\sigma = \alpha_1, \ldots, \alpha_n$ such that for each $1 \leq i \leq n$;*

- *there is a substitution θ_i and a STRIPS action $\alpha'_i \in \mathbb{A}$ such that $\alpha'_i\theta = \alpha_i$; and*
- *there is a sequence of states $\mathbf{S}_0, \mathbf{S}_1, \ldots, \mathbf{S}_n$ such that*
  - *$\mathbf{S} = \mathbf{S}_0$ and $G \subseteq \mathbf{S}_n$ (i.e., G is satisfied in the final state);*
  - *α_i is executable in state $\mathbf{S}_{i-1}$ and the result of that execution is the state $\mathbf{S}_i$.*

2.2 Overview of Transaction Logic

To make this paper self-contained, we provide a brief introduction to the *subset* of Transaction Logic ($\mathcal{TR}$) [9,11–14] that are needed for the understanding of this paper.

As an extension of first-order predicate calculus, $\mathcal{TR}$ is sharing most of its syntax with the first-order predicate calculus' syntax. One of the new connectives that $\mathcal{TR}$ adds to the first-order predicate calculus is the ***serial conjunction***, denoted $\otimes$. It is a binary associative, non-commutative connective. The formula $\phi \otimes \psi$, showing a composite action, denotes an *execution* of ϕ followed by an execution of ψ. When ϕ and ψ are regular first-order formulas, $\phi \otimes \psi$ reduces to the usual first-order conjunction, $\phi \wedge \psi$. The logic also introduces other connectives to support hypothetical reasoning, concurrent execution, etc., but these are beyond the scope of this paper.

To take the *frame problem* out of many considerations in $\mathcal{TR}$, it has an extensible mechanism of ***elementary updates*** (see [10,11,13,14,42]). Due to the definition of *STRIPS* actions, we just need the following two types of elementary updates (actions): $+p(t_1,\ldots,t_n)$ and $-p(t_1,\ldots,t_n)$, where $p(t_1,\ldots,t_n)$ denotes an *extensional* atom. Given a state $\mathbf{S}$ and a *ground* elementary action $+p(a_1,\ldots,a_n)$, an execution of $+p(a_1,\ldots,a_n)$ at state $\mathbf{S}$ deletes the literal $\neg p(a_1,\ldots,a_n)$ and adds the literal $p(a_1,\ldots,a_n)$. Similarly, executing $-p(a_1,\ldots,a_n)$ results in a state that is exactly like $\mathbf{S}$, but $p(a_1,\ldots,a_n)$ is deleted and $\neg p(a_1,\ldots,a_n)$ added. Apparently, if $p(a_1,\ldots,a_n) \in \mathbf{S}$, the action $+p(a_1,\ldots,a_n)$ has no effect, and similarly for $-p(a_1,\ldots,a_n)$.

We can define a ***complex action*** using ***serial rule*** that is a statement of the form

$$h \leftarrow b_1 \otimes b_2 \otimes \ldots \otimes b_n. \tag{1}$$

where h is an atomic formula denoting the complex action and b_1, ..., b_n are literals or elementary actions. That means that h is a complex action and one way to execute h is to execute b_1 then b_2, etc., and finally to execute b_n. Note that we have regular first-order as well as serial-Horn rules. For simplicity, we assume that the sets of intensional predicates that can appear in the heads of regular rules and those in the heads of serial rules are disjoint. *Extensional atoms* and *Intensional atoms* compose state (see Definition 1) and will be collectively called ***fluents***. Note that a serial rule all of whose body literals are fluents is essentially a regular rule, since all the $\otimes$-connectives can be replaced with $\wedge$. Therefore, one can view the regular rules as a special case of serial rules.

The following example illustrates the above concepts. All our examples use the standard logic programming convention whereby lowercase symbols represent constants and predicate symbols, while the uppercase symbols stand for variables that are universally quantified outside of the rules. It is common practice to omit such quantifiers.

$$\begin{aligned}
move(X,Y) \;&\leftarrow (on(X,Z) \wedge clear(X) \\
&\qquad \wedge clear(Y) \wedge \neg tooHeavy(X)) \otimes \\
&\quad -on(X,Z) \otimes +on(X,Y) \otimes \\
&\quad -clear(Y). \\
tooHeavy(X) \;&\leftarrow weight(X,W) \wedge limit(L) \wedge \\
&\qquad W < L. \\
?- \;\; &move(blk1, blk15) \otimes move(SomeBlk, blk1).
\end{aligned}$$

Here *on*, *clear*, *tooHeavy*, and *weight* are fluents and the rest of atoms represent actions. The predicate *tooHeavy* is an intensional fluent, while *on*, *clear*, and *weight* are extensional fluents. The actions $+on(...)$, $-clear(...)$, and $-on(...)$ are elementary and the intensional predicate *move* represents a complex action. This example illustrates several features of Transaction Logic. The first rule is a serial rule defining of a complex action of moving a block from one place to another. The second rule defines the intensional fluent *tooHeavy*, which is used in the definition of *move* (under the scope of default negation). As the second rule does not include any action, it is a regular rule.

The last statement above is a *request to execute* a composite action, which is analogous to a query in logic programming. The request is to move block *blk1* from its current position to the top of *blk15* and then find some other block and move it on top of *blk1*. Traditional logic programming offers no logical semantics for updates, so if after placing *blk1* on top of *blk15* the second operation $(move(SomeBlk, blk1))$ fails (say, all available blocks are too heavy), the effects of the first operation will persist and the underlying database becomes corrupted. In contrast, Transaction Logic gives update operators the logical semantics of an *atomic database transaction*. This means that if any part of the transaction fails, the effect is as if nothing was done at all. For example, if the second action in our example fails, all actions are "backtracked over" and the underlying database state remains unchanged.

$\mathcal{TR}$'s semantics is given in purely model-theoretic terms and here we will only give an informal overview. The truth of any action in $\mathcal{TR}$ is determined over sequences of states—***execution paths***—which makes it possible to think of truth assignments in $\mathcal{TR}$'s models as executions. If an action, ψ, defined by a set of serial rules, $\mathbb{P}$, evaluates to true over a sequence of states $\mathbf{D}_0, \ldots, \mathbf{D}_n$, we say that it can *execute* at state $\mathbf{D}_0$ by passing through the states $\mathbf{D}_1$, ..., $\mathbf{D}_{n-1}$, ending in the final state $\mathbf{D}_n$. This is captured by the notion of ***executional entailment***, which is written as follows:

$$\mathbb{P}, \mathbf{D}_0 \ldots \mathbf{D}_n \models \psi \tag{2}$$

Various inference systems for serial-Horn $\mathcal{TR}$ [11] are similar to the well-known SLD resolution proof strategy for Horn clauses plus some $\mathcal{TR}$-specific inference rules and axioms. Given a set of serial rules, $\mathbb{P}$, and a *serial goal*, ψ (i.e., a formula that has the form of a body of a serial rule such as (1), these inference systems prove statements of the form $\mathbb{P}, \mathbf{D} \cdots \vdash \psi$, called ***sequents***. A proof of a sequent of this form is interpreted as a proof that action ψ defined by the rules in $\mathbb{P}$ can be successfully executed starting at state $\mathbf{D}$.

An inference succeeds iff it finds an execution for the transaction ψ. The execution is a sequence of database states $\mathbf{D}_1, \ldots, \mathbf{D}_n$ such that $\mathbb{P}, \mathbf{D}\, \mathbf{D}_1 \ldots \mathbf{D}_n \vDash \psi$. We will use the following inference system in our planning application. For simplicity, we present only the version for ground facts and rules. The inference rules can be read either top-to-bottom (if *top* is proved then *bottom* is proved) or bottom-to-top (to prove *bottom* one needs to prove *top*).

Definition 5 ($\mathcal{TR}$ inference System). *Let $\mathbb{P}$ be a set of rules (serial or regular) and $\mathbf{D}$, $\mathbf{D}_1$, $\mathbf{D}_2$ denote states.*

- *Axiom: $\mathbb{P}, \mathbf{D} \cdots \vdash ()$, where $()$ is an empty clause (which is true at every state).*
- *Inference Rules*
  1. *Applying transaction definition: Suppose $t \leftarrow body$ is a rule in $\mathbb{P}$.*

$$\frac{\mathbb{P}, \mathbf{D} \cdots \vdash body \otimes rest}{\mathbb{P}, \mathbf{D} \cdots \vdash t \otimes rest} \tag{3}$$

2. *Querying the database: If* $\mathbf{D} \models t$ *then*

$$\frac{\mathbb{P}, \mathbf{D} \cdots \vdash rest}{\mathbb{P}, \mathbf{D} \cdots \vdash t \otimes rest} \tag{4}$$

3. *Performing elementary updates: If the elementary update* t *changes the state* $\mathbf{D}_1$ *into the state* $\mathbf{D}_2$ *then*

$$\frac{\mathbb{P}, \mathbf{D}_2 \cdots \vdash rest}{\mathbb{P}, \mathbf{D}_1 \cdots \vdash t \otimes rest} \tag{5}$$

A ***proof*** of a sequent, seq_n, is a series of sequents, seq_1, seq_2, ... , seq_{n-1}, seq_n, where each seq_i is either an axiom-sequent or is derived from earlier sequents by one of the above inference rules. This inference system has been proven to be sound and complete with respect to the model theory of $\mathcal{TR}$ [11]. This means that if ϕ is a serial goal, the executional entailment $\mathbb{P}, \mathbf{D}\, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi$ holds if and only if there is a proof of $\mathbb{P}, \mathbf{D} \cdots \vdash \phi$ over the execution path $\mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n$, i.e., $\mathbf{D}_1, \ldots, \mathbf{D}_n$ is the sequence of intermediate states that appear in the proof and $\mathbf{D}$ is the initial state. In this case, we will also say that such a proof proves the statement $\mathbb{P}, \mathbf{D}\, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash \phi$.

3 $\mathcal{TR}$ Planners

The informal encoding of *STRIPS* and forward state space planning as sets of $\mathcal{TR}$ rules first appeared in an unpublished report [11]. To use $\mathcal{TR}$ as a planning formalism, we formally show how a planning problem specification can be transformed into a set of $\mathcal{TR}$ rules that represent *STRIPS* and *Forward State Space* planning techniques. From now on, we call Forward State Space planning technique *naive planning*, as it is one of the simplest possible state space planning techniques. We also show that $\mathcal{TR}$ inference system uses those sets of $\mathcal{TR}$ rules to construct a plan. To highlight the correspondence between these sets of $\mathcal{TR}$ rules and original *STRIPS* and naive planning techniques, we first briefly review these planning techniques in terms of imperative pseudo codes.

The original *STRIPS* planning algorithm, proposed by [19], maintains a *stack* of goals and tries to achieve the goals from the top of the stack until the stack gets empty. We can simply implement this technique using recursive functions as depicted in Figure 1. Naive planning algorithm is based on *depth first search*. As illustrated in Figure 2, it starts from initial state, iteratively chooses actions, and moves to a new state until eventually finds a goal state.

The following definitions encode the aforesaid planning techniques as a set of $\mathcal{TR}$ rules.

Definition 6 (Enforcement operator). *Let* G *be a set of extensional literals. We define* $Enf(G) = \{+p | p \in G\} \cup \{-p | \neg p \in G\}$. *In other words,* $Enf(G)$ *is the set of elementary updates that makes* G *true.*

Next we introduce a natural correspondence between *STRIPS* actions and $\mathcal{TR}$ rules.

```
function STRIPS(ℝ, 𝔸, S, G)
    σ ← []
    loop
        if G ⊆ S then
            return σ
        else
            A ← {αθ|α ∈ 𝔸, θ(E(α)) ⊆ G}
            if A = ∅ then
                reutrn failure
            else
                Choose non-deterministically α ∈ A
                σ' ← STRIPS(ℝ, 𝔸, S, Pre(α))
                if σ' = failure then
                    reutrn failure
                else
                    S ← exec(S, σ')
                    S ← (S \ ¬θ(E(α))) ∪ θ(E(α))
                    σ ← [σ|σ'|αθ]
                end if
            end if
        end if
    end loop
    return σ
end function
```

Fig. 1. *STRIPS* Planning

```
function NAIVE(ℝ, 𝔸, S_0, G)
    S ← S_0
    σ ← []
    loop
        if G ⊆ S then
            return σ
        else
            A ← {αθ|α ∈ 𝔸, θ(Pre(α)) ⊆ S}
            if A = ∅ then
                reutrn failure
            else
                Choose non-deterministically α ∈ A
                S ← (S \ ¬θ(E(α))) ∪ θ(E(α))
                σ ← [σ|αθ]
            end if
        end if
    end loop
    return σ
end function
```

Fig. 2. Naive Planning

Definition 7 (Actions as $\mathcal{TR}$ rules). *Let $\alpha = \langle p_\alpha(\overline{X}), Pre(\alpha), E(\alpha)\rangle$ be a STRIPS action. We define its* ***corresponding $\mathcal{TR}$ rule****, $tr(\alpha)$, to be a rule of the form*

$$p_\alpha(\overline{X}) \leftarrow (\wedge_{\ell \in Pre(\alpha)} \ell) \otimes (\otimes_{u \in Enf(E(\alpha))} u). \tag{6}$$

Note that in (6) the actual order of action execution in the last component, $\otimes_{u \in Enf(E(\alpha))} u$, is immaterial, since all such executions happen to lead to the same state.

We now give a set of $\mathcal{TR}$ clauses that simulates naive planning for *STRIPS* planning problems [19]. For convenience, we use $a \widehat{\otimes} b$ as a shorthand for $a \otimes b \vee b \otimes a$. This connective is called the *shuffle* operator in [11]. We define it to be commutative and associative and thus extend it to arbitrary number of operands.

Definition 8 (Naïve planning rules). *Given a STRIPS planning problem $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$ (see Definition 4), we define a set of $\mathcal{TR}$ rules, $\mathbb{P}(\Pi)$, which simulate naive planning technique to provide a planning solution to the planning problem. $\mathbb{P}(\Pi)$ has two parts, $\mathbb{P}_{general}$, $\mathbb{P}_{\mathbb{A}}$, described below.*

- *The $\mathbb{P}_{general}$ part: contains a couple of rules as follows;*

$$\begin{array}{l} plan \leftarrow . \\ plan \leftarrow execute\_action \otimes plan. \end{array} \tag{7}$$

  These rules construct a sequence of actions and bind them to the plan.
- *The $\mathbb{P}_{actions}$ part: for each $\alpha \in \mathbb{A}$, $\mathbb{P}_{actions}$ has a couple of rules as follows;*

$$\begin{array}{l} p_\alpha(\overline{X}) \leftarrow (\wedge_{\ell \in Pre(\alpha)} \ell) \otimes (\otimes_{u \in Enf(E(\alpha))} u). \\ execute\_action \leftarrow p_\alpha(\overline{X}). \end{array} \tag{8}$$

  This is the $\mathcal{TR}$ rule that corresponds to the action α, introduced in Definition 7 and generally links an action to a plan.

Given a *STRIPS* planning problem $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$, Definition 8 gives a set of $\mathcal{TR}$ rules that specify a naive planning strategy for that problem. To find a solution for that planning problem, one simply needs to place the request

$$? - plan \otimes (\wedge_{g_i \in G} g_i). \tag{9}$$

and use the $\mathcal{TR}$'s inference system to find a proof. As mentioned before, a solution plan for a *STRIPS* planning problem is a sequence of actions leading to a state that satisfies the planning goal. Such a sequence can be extracted by picking out the atoms of the form p_α from a successful derivation branch generated by the $\mathcal{TR}$ inference system. Since each p_α uniquely corresponds to a *STRIPS* action, this provides us with the requisite sequence of actions that constitutes a plan.

Suppose $seq_0, \ldots, seq_m$ is a deduction by the $\mathcal{TR}$ inference system. Let $i_1, \ldots, i_n$ be exactly those indexes in that deduction where the inference rule

(3) was applied to some sequent using a rule of the form $tr(\alpha_{i_r})$ introduced in Definition 7. We will call $\alpha_{i_1}, \ldots, \alpha_{i_n}$ the ***pivoting sequence of actions***. The corresponding **pivoting sequence of states** $\mathbf{D}_{i_1}, \ldots, \mathbf{D}_{i_n}$ is a sequence where each $\mathbf{D}_{i_r}$, $1 \leq r \leq n$, is the state at which α_{i_r} is applied. One can show that the pivoting sequence of actions generated from a deduction of (9) is a solution to the planning problem. *Completeness* of a planning strategy means that, for any *STRIPS* planning problem, if there is a solution, the planner will find at least one plan. Based on the completeness of $\mathcal{TR}$'s inference system, one can show that the planner in Definition 8 is complete.

Definition 9 (*STRIPS* planning rules). *Let $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S} \rangle$ be a STRIPS planning problem (see Definition 4). We define a set of $\mathcal{TR}$ rules, $\mathbb{P}(\Pi)$, which simulate STRIPS planning technique to provide a planning solution to the planning problem. $\mathbb{P}(\Pi)$ has three disjoint parts, $\mathbb{P}_{\mathbb{R}}$, $\mathbb{P}_{\mathbb{A}}$, and $\mathbb{P}_G$, described below.*

- *The $\mathbb{P}_{\mathbb{R}}$ part: for each rule $p(\overline{X}) \leftarrow p_1(\overline{X}_1) \wedge \cdots \wedge p_k(\overline{X}_n)$ in $\mathbb{R}$, $\mathbb{P}_{\mathbb{R}}$ has a rule of the form*

$$achieve\_p(\overline{X}) \leftarrow \widehat{\otimes}_{i=1}^{n} achieve\_p_i(\overline{X}_i). \tag{10}$$

  Rule (10) is an extension to the classical STRIPS planning algorithm. It captures intensional predicates and ramification of actions, and it is the only major aspect of our $\mathcal{TR}$-based rendering of STRIPS that was not present in the original in one way or another.
- *The part $\mathbb{P}_{\mathbb{A}} = \mathbb{P}_{actions} \cup \mathbb{P}_{atoms} \cup \mathbb{P}_{achieves}$ is constructed out of the actions in $\mathbb{A}$ as follows:*
  - *$\mathbb{P}_{actions}$: for each $\alpha \in \mathbb{A}$, $\mathbb{P}_{actions}$ has a rule of the form*

$$p_\alpha(\overline{X}) \leftarrow (\wedge_{\ell \in Pre(\alpha)} \ell) \ \otimes \ (\otimes_{u \in Enf(E(\alpha))} u). \tag{11}$$

    This is the $\mathcal{TR}$ rule that corresponds to the action α, introduced in Definition 7.
  - *$\mathbb{P}_{atoms} = \mathbb{P}_{achieved} \cup \mathbb{P}_{enforced}$ has two disjoint parts as follows:*
    - *$\mathbb{P}_{achieved}$: for each extensional predicate $p \in \mathcal{P}_{ext}$, $\mathbb{P}_{achieved}$ has the rules*

$$\begin{array}{l} achieve\_p(\overline{X}) \leftarrow p(\overline{X}). \\ achieve\_not\_p(\overline{X}) \leftarrow \neg p(\overline{X}). \end{array} \tag{12}$$

      These rules say that if an extensional literal is true in a state then that literal has already been achieved as a goal.
    - *$\mathbb{P}_{enforced}$: for each action $\alpha = \langle p_\alpha(\overline{X}), Pre(\alpha), E(\alpha) \rangle$ in $\mathbb{A}$ and each $e(\overline{Y}) \in E(\alpha)$, $\mathbb{P}_{enforced}$ has the following rule:*

$$achieve\_e(\overline{Y}) \leftarrow \neg e(\overline{Y}) \otimes execute\_p_\alpha(\overline{X}). \tag{13}$$

      This rule says that one way to achieve a goal that occurs in the effects of an action is to execute that action.

- $\mathbb{P}_{achieves}$: *for each action* $\alpha = \langle p_\alpha(\overline{X}), Pre(\alpha), E(\alpha)\rangle$ *in* $\mathbb{A}$, $\mathbb{P}_{achieves}$ *has the following rule:*

$$execute\_p_\alpha(\overline{X}) \leftarrow (\widehat{\otimes}_{\ell \in Pre(\alpha)} achieve\_\ell) \otimes p_\alpha(\overline{X}). \tag{14}$$

  This means that to execute an action, one must first achieve the precondition of the action and then perform the state changes prescribed by the action.

– $\mathbb{P}_G$: *Let* $G = \{g_1, ..., g_k\}$. *Then* $\mathbb{P}_G$ *has a rule of the form:*

$$achieve_G \leftarrow (\widehat{\otimes}_{g_i=1}^{k} achieve\_g_i) \otimes (\wedge_{i=1}^{k} g_i). \tag{15}$$

Similar to naive planning, given a *STRIPS* planning problem $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$, Definition 9 gives a set of $\mathcal{TR}$ rules. $\mathcal{TR}$'s inference system can use those rules to simulate *STRIPS* planning strategy for that problem. If one places the request

$$? - achieve_G\,. \tag{16}$$

the $\mathcal{TR}$'s inference system will find a proof. It can be shown that the pivoting sequence of actions generated from this proof is a solution to the planning problem. One can also show that the linear planner provided by Definition 9 is complete under the set of goal-serializable planning problems. We do not further discuss these issues in this paper due to space limitations. One can also show that with the help of existing tabling methods for $\mathcal{TR}$'s inference systems, *STRIPS* planner always terminates.

Definitions 8 and 9 are transforms that convert the specification of planning problems to $\mathcal{TR}$ rules. The similarity of $\mathcal{TR}$'s inference system in Definition 5 and well-known SLD resolution algorithm shows that one can use a similar approach to encode planning algorithms in logic programming frameworks. Based on Definitions 8 and 9, we can build a simple translator that constructs $\mathcal{TR}$ rules out of planning problem specifications in PDDL.

4 Experiments

In this section we briefly report on our experiments that compare naive and *STRIPS* planning. The test environment was a tabled $\mathcal{TR}$ interpreter [20] implemented in XSB and running on Intel®Xeon(R) CPU E5-1650 0 @ 3.20GHz 12 CPU, 64GB memory running on Mint Linux 14 64-bit. We use our translator to build $\mathcal{TR}$ rules out of PDDL files. We use the generated $\mathcal{TR}$ rules and our interpreter to solve our planning problems, which are test cases taken from [1]. We do not explain our test cases as they are well explained at *http://ipc.icaps-conference.org/*. Our test cases also can be found at *http://ewl.cewit.stonybrook.edu/planning/* along with our PDDL2TR translator, $\mathcal{TR}$ interpreter, and all the necessary items needed to reproduce the results. The tests highlight how the performance of the two planning techniques varies depending on the domain of application.

Table 1. Results for the Elevator test case (4 actions)

Test Case	Naive		*STRIPS*	
	CPU	Mem	CPU	Mem
s1-0	0	19	0	17
s2-0	0.004	277	0	96
s3-0	0.092	3636	0.02	853
s4-0	1.352	54628	0.1	4152
s5-0	24.213	867806	0.348	14148
s6-0	463.908	13440627	1.032	44681
s7-0	1000>	N/A	3.14	144060
s8-0	1000>	N/A	9.564	430435
s9-0	1000>	N/A	27.425	1212350
s10-0	1000>	N/A	74.24	3115811
s11-0	1000>	N/A	217.545	9074006
s12-0	1000>	N/A	546.606	21151356

Table 2. Results for the Travelling and Purchase Problem test case (3 actions)

Test Case	Naive		*STRIPS*	
	CPU	Mem	CPU	Mem
p01	0	22	0.004	121
p02	0.004	125	0.004	215
p03	0.024	664	0.016	878
p04	0.124	4040	0.056	2067
p05	41.43	2350601	0.592	19371

The main difference between the two test cases is that the Healthcare test case has many more actions and intensional rules than the movie store case. As seen from Tables 1 and 2, for both of these test cases, *STRIPS* planning gets to about two orders of magnitude more efficient both in time and space.[1] However, *STRIPS* is not able to solve problems that are not goal serializable.

We do not compare and analyse the performances of studied planning techniques in this paper as this study would be beyond the scope of this paper. The aim of our experiments is to provide show the differences between planning techniques in different application domains and to illustrate the ability of $\mathcal{TR}$ to not only provide a theoretical framework for analysis of planning techniques, but also to implement such techniques in a declarative way. Moreover, our experiments also show that $\mathcal{TR}$ simplifies and eases the implementation of complicated planning techniques, such as *STRIPS*.

5 Compatibility with PDDL and Related Work

The representation of planning algorithms in $\mathcal{TR}$ enables us to develop a simple translator that constructs $\mathcal{TR}$ rules out of planning problems and algorithms.

[1] Time is measured in seconds and memory in kilobytes.

PDDL is a standard language intended to express planning problems [28] in AIPS planning competitions [1]. Planning problems from AIPS planning competitions are usually considered as standard benchmarks for planners. Therefore, providing an automated method of translating PDDL planning domains shows the generality of our approach.

A planning problem consists of domain predicates, possible actions, the structure of compound actions, and the effects of actions. To express a planning problem, PDDL supports several syntactic features such as basic *STRIPS*-style actions, conditional effects, universal quantifications in the effects, ADL features [41], domain axioms, safety constraints, hierarchical actions, and more. The formalism in Section 2 also supports the basic *STRIPS*-style actions and domain axioms. Clearly, it is simple to extend this formalism to include other features. For instance, it is easy to show that $\mathcal{TR}$ is can represent most of the features provided by different extensions of PDDL. The following list briefly shows how $\mathcal{TR}$ can express some of these main features.

- ADL features [28,37]: in PDDL, actions can have a first order formula in their precondition. The effect of an action can also include universal quantifications over fluents. $\mathcal{TR}$ can use Lloyd-Topor transformation to support first order formula (including universal and existential quantifiers and disjunction) in the precondition of actions. It also can simulate universal quantifications over fluents in the effects of actions.
- Numerical extensions [22]: in PDDL, one can associate actions, objects, and plans with numeric costs and use these costs in numerical expressions to compute different planning metrics. This syntactical feature also needs PDDL to include numerical operators. Since $\mathcal{TR}$ can express and encode numerical operators and expressions as a part of its model theory, it can easily handle this feature.
- Temporal extensions and durative actions [22,25,38,43]: PDDL is able to express discretised and continuous actions [22]. $\mathcal{TR}$ is also able to represent discretised and continuous actions because the notion of *time* can be encoded in $\mathcal{TR}$'s transactions.
- Plan and solution preferences and constraints [27,31]: In a planning problem, it is possible that only a subset of goals can be achieved because of the conflict between goals. In this situation, the ability to assign importance and preferences to different goals is essential. PDDL provides such ability to express such preferences among goals and planning solutions. $\mathcal{TR}$ augmented with defeasible reasoning [21] also can easily provide this feature.

Answer set programming is one of the leading logic-based planning techniques [36][24][44]. However, encoding a planning problem in answer set programming requires the addition of inertia axioms to solve frame problem [23]. Clearly, $\mathcal{TR}$-based planning does not face this problem. Our PDDL2TR translator also shows that $\mathcal{TR}$-based planning is general enough to automatically encode planning problems. Since $\mathcal{TR}$'s model theory also avoids the frame problem and no inertia axioms are required in $\mathcal{TR}$'s planning rules. Picat is another

logic programming framework that relies on tabling. It has been shown to be an efficient logic-based system for solving planning problems [4][49][3][48]. However, the techniques employed by Picat are orthogonal to the results presented here and, we believe, this work provides a natural direction for incorporation of complex planning algorithms, like *STRIPS*, into Picat.

6 Conclusion

This paper has demonstrated that Transaction Logic can bridge the gap between AI planning and logic programming. Specially, we claim that $\mathcal{TR}$ is a general framework for analysis and implementation in the area of planning, which does not depend on any particular planning strategy.

As an illustration, we have shown that different planning strategies, such as *STRIPS*, not only can be naturally represented in $\mathcal{TR}$, but that also such representations can be used to automatically translate planning problems and algorithms into declarative programming languages (e.g. Prolog). We have also shown that the use of this powerful logic opens up new possibilities for improvement of existing planning methods in logic programming. For instance, we have shown that the sophisticated *STRIPS* algorithm can be cast as a set of rules in $\mathcal{TR}$, which shows the ability of rule based systems to represent such planning techniques.

These non-trivial insights were acquired merely due to the use of $\mathcal{TR}$ and not much else. The same technique can be used to cast even more advanced strategies such as *ABSTRIPS* [40], and HTN [39] as $\mathcal{TR}$ rules, and those techniques can straightforwardly be used to solve planning problems in logic programming frameworks.

There are several promising directions to continue this work. One is to investigate other planning strategies and, hopefully, accrue similar benefits. Other possible directions include heuristics and plans with loops [33,34,46]. For instance loops are easily representable using recursive actions in $\mathcal{TR}$.

References

1. Bacchus, F.: AIPS-00 planning competition, May 2000. http://www.cs.toronto.edu/aips2000
2. Barták, R., Toropila, D.: Solving sequential planning problems via constraint satisfaction. Fundam. Inform. **99**(2), 125–145 (2010). http://dx.doi.org/10.3233/FI-2010-242
3. Barták, R., Zhou, N.-F.: On modeling planning problems: experience from the petrobras challenge. In: Castro, F., Gelbukh, A., González, M. (eds.) MICAI 2013, Part II. LNCS, vol. 8266, pp. 466–477. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-45111-9_40
4. Barták, R., Zhou, N.: Using tabled logic programming to solve the petrobras planning problem. TPLP **14**(4–5), 697–710 (2014). http://dx.doi.org/10.1017/S1471068414000295

5. Basseda, R., Kifer, M., Bonner, A.J.: Planning with transaction logic. In: Kontchakov, R., Mugnier, M.-L. (eds.) RR 2014. LNCS, vol. 8741, pp. 29–44. Springer, Heidelberg (2014)
6. Bibel, W.: A deductive solution for plan generation. New Generation Computing **4**(2), 115–132 (1986)
7. Bibel, W.: A deductive solution for plan generation. In: Schmidt, J.W., Thanos, C. (eds.) Foundations of Knowledge Base Management. Information Systems, pp. 453–473. Springer, Heidelberg (1989)
8. Bibel, W., del Cerro, L.F., Fronhfer, B., Herzig, A.: Plan generation by linear proofs: on semantics. In: Metzing, D. (ed.) GWAI-89 13th German Workshop on Artificial Intelligence, Informatik-Fachberichte, vol. 216, pp. 49–62. Springer, Heidelberg (1989)
9. Bonner, A., Kifer, M.: Transaction logic programming. In: Int'l Conference on Logic Programming, pp. 257–282. MIT Press, Budapest, June 1993
10. Bonner, A.J., Kifer, M.: Applications of transaction logic to knowledge representation. In: Gabbay, D.M., Ohlbach, H.J. (eds.) ICTL1994. LNCS, vol. 827, pp. 67–81. Springer, Heidelberg (1994)
11. Bonner, A., Kifer, M.: Transaction logic programming (or a logic of declarative and procedural knowledge). Tech. Rep. CSRI-323, University of Toronto, November 1995. http://www.cs.toronto.edu/~bonner/transaction-logic.html
12. Bonner, A., Kifer, M.: Concurrency and communication in transaction logic. In: Joint Int'l Conference and Symposium on Logic Programming, pp. 142–156. MIT Press, Bonn, September 1996
13. Bonner, A., Kifer, M.: A logic for programming database transactions. In: Chomicki, J., Saake, G. (eds.) Logics for Databases and Information Systems, ch. 5, pp. 117–166. Kluwer Academic Publishers, March 1998
14. Bonner, A.J., Kifer, M.: An overview of transaction logic. Theoretical Computer Science **133** (1994)
15. Cresswell, S., Smaill, A., Richardson, J.: Deductive synthesis of recursive plans in linear logic. In: Biundo, S., Fox, M. (eds.) Recent Advances in AI Planning. Lecture Notes in Computer Science, vol. 1809, pp. 252–264. Springer, Heidelberg (2000)
16. Dovier, A., Formisano, A., Pontelli, E.: Perspectives on logic-based approaches for reasoning about actions and change. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. LNCS, vol. 6565, pp. 259–279. Springer, Heidelberg (2011). http://dx.doi.org/10.1007/978-3-642-20832-4_17
17. Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic sat-compilation of planning problems. In: Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence, IJCAI 1997, vol. 2, pp. 1169–1176. Morgan Kaufmann Publishers Inc., San Francisco (1997). http://dl.acm.org/citation.cfm?id=1622270.1622325
18. Erol, K., Hendler, J.A., Nau, D.S.: UMCP: a sound and complete procedure for hierarchical task-network planning. In: Hammond, K.J. (ed.) AAAI 1994, pp. 249–254. University of Chicago, Chicago (1994). http://www.aaai.org/Library/AIPS/1994/aips94-042.php
19. Fikes, R.E., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence **2**(34), 189–208 (1971)
20. Fodor, P., Kifer, M.: Tabling for transaction logic. In: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP 2010, pp. 199–208. ACM, New York (2010)

21. Fodor, P., Kifer, M.: Transaction logic with defaults and argumentation theories. In: Gallagher, J.P., Gelfond, M. (eds.) Technical Communications of the 27th International Conference on Logic Programming, ICLP 2011, July 6–10, 2011, Lexington, Kentucky, USA. LIPIcs, vol. 11, pp. 162–174. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011). http://dx.doi.org/10.4230/LIPIcs.ICLP.2011.162
22. Fox, M., Long, D.: Pddl2.1: An extension to pddl for expressing temporal planning domains. J. Artif. Int. Res. **20**(1), 61–124 (2003). http://dl.acm.org/citation.cfm?id=1622452.1622454
23. Gebser, M., Kaminski, R., Knecht, M., Schaub, T.: plasp: A prototype for PDDL-based planning in ASP. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 358–363. Springer, Heidelberg (2011)
24. Gebser, M., Kaufmann, R., Schaub, T.: Gearing up for effective ASP planning. In: Erdem, E., Lee, J., Lierler, Y., Pearce, D. (eds.) Correct Reasoning. LNCS, vol. 7265, pp. 296–310. Springer, Heidelberg (2012). http://dl.acm.org/citation.cfm?id=2363344.2363364
25. Geffner, H.: Pddl 2.1: Representation vs. computation. J. Artif. Int. Res. **20**(1), 139–144 (2003). http://dl.acm.org/citation.cfm?id=1622452.1622457
26. Gelfond, M., Lifschitz, V.: Action languages. Electron. Trans. Artif. Intell. **2**, 193–210 (1998). http://www.ep.liu.se/ej/etai/1998/007/
27. Gerevini, A., Haslum, P., Long, D., Saetti, A., Dimopoulos, Y.: Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. Artif. Intell. **173**(5–6), 619–668 (2009). http://dx.doi.org/10.1016/j.artint.2008.10.012
28. Ghallab, M., Isi, C.K., Penberthy, S., Smith, D.E., Sun, Y., Weld, D.: PDDL - The Planning Domain Definition Language. Tech. rep., CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212
29. Giunchiglia, E., Massarotto, A., Sebastiani, R.: Act, and the rest will follow: exploiting determinism in planning as satisfiability. In: Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI 1998/IAAI 1998, pp. 948–953. American Association for Artificial Intelligence, Menlo Park (1998). http://dl.acm.org/citation.cfm?id=295240.295931
30. Green, C.: Application of theorem proving to problem solving. In: Proceedings of the 1st International Joint Conference on Artificial Intelligence, IJCAI 1969, pp. 219–239. Morgan Kaufmann Publishers Inc., San Francisco (1969). http://dl.acm.org/citation.cfm?id=1624562.1624585
31. Gregory, P., Long, D., Fox, M.: Constraint based planning with composable substate graphs. In: Coelho, H., Studer, R., Wooldridge, M. (eds.) Proceedings of ECAI 2010–19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16–20, 2010. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 453–458. IOS Press (2010). http://dx.doi.org/10.3233/978-1-60750-606-5-453
32. Hölldobler, S., Schneeberger, J.: A new deductive approach to planning. New Generation Computing **8**(3), 225–244 (1990)
33. Kahramanogullari, O.: Towards planning as concurrency. In: Hamza, M.H. (ed.) Artificial Intelligence and Applications, pp. 387–393. IASTED/ACTA Press (2005)
34. Kahramanogullari, O.: On linear logic planning and concurrency. Information and Computation **207**(11), 1229–1258 (2009). special Issue: 2nd International Conference on Language and Automata Theory and Applications (LATA 2008)

35. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of the 10th European Conference on Artificial Intelligence, ECAI 1992, pp. 359–363. John Wiley & Sons Inc., New York (1992). http://dl.acm.org/citation.cfm?id=145448.146725
36. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138**(12), 39–54 (2002). http://www.sciencedirect.com/science/article/pii/S0004370202001868. knowledge Representation and Logic Programming
37. McDermott, D.V.: The 1998 AI planning systems competition. AI Magazine **21**(2), 35–55 (2000). http://www.aaai.org/ojs/index.php/aimagazine/article/view/1506
38. McDermott, D.V.: PDDL2.1 - the art of the possible? commentary on fox and long. J. Artif. Intell. Res. (JAIR) **20**, 145–148 (2003). http://dx.doi.org/10.1613/jair.1996
39. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
40. Nilsson, N.: Principles of Artificial Intelligence. Tioga Publ. Co., Paolo Alto (1980)
41. Pednault, E.P.D.: Adl: Exploring the middle ground between strips and the situation calculus. In: Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning, pp. 324–332. Morgan Kaufmann Publishers Inc., San Francisco (1989). http://dl.acm.org/citation.cfm?id=112922.112954
42. Rezk, M., Kifer, M.: Transaction logic with partially defined actions. J. Data Semantics **1**(2), 99–131 (2012)
43. Smith, D.E.: The case for durative actions: A commentary on PDDL2.1. J. Artif. Intell. Res. (JAIR) **20**, 149–154 (2003). http://dx.doi.org/10.1613/jair.1997
44. Son, T.C., Baral, C., Tran, N., Mcilraith, S.: Domain-dependent knowledge in answer set planning. ACM Trans. Comput. Logic **7**(4), 613–657 (2006). http://doi.acm.org/10.1145/1183278.1183279
45. Son, T.C., Pontelli, E., Sakama, C.: Logic programming for multiagent planning with negotiation. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 99–114. Springer, Heidelberg (2009). http://dx.doi.org/10.1007/978-3-642-02846-5_13
46. Srivastava, S., Immerman, N., Zilberstein, S., Zhang, T.: Directed search for generalized plans using classical planners. In: Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS-2011). AAAI, June 2011
47. Stefik, M.J.: Planning with Constraints. Ph.D. thesis, Stanford, CA, USA (1980). aAI8016868
48. Zhou, N., Dovier, A.: A tabled prolog program for solving sokoban. In: IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7–9, 2011, pp. 896–897. IEEE Computer Society (2011). http://dx.doi.org/10.1109/ICTAI.2011.145
49. Zhou, N., Dovier, A.: A tabled prolog program for solving sokoban. Fundam. Inform. **124**(4), 561–575 (2013). http://dx.doi.org/10.3233/FI-2013-849

On Compiling Linear Logic Programs with Comprehensions, Aggregates and Rule Priorities

Flavio Cruz[1,2(✉)] and Ricardo Rocha[2]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, USA
fmfernan@cs.cmu.edu
[2] CRACS and INESC TEC and Faculty of Sciences, University of Porto,
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{flavioc,ricroc}@dcc.fc.up.pt

Abstract. Linear logic programs are challenging to implement efficiently because facts are asserted and retracted frequently. Implementation is made more difficult with the introduction of useful features such as rule priorities, which are used to specify the order of rule inference, and comprehensions or aggregates, which are mechanisms that make data iteration and gathering more intuitive. In this paper, we describe a compilation scheme for transforming linear logic programs enhanced with those features into efficient C++ code. Our experimental results show that compiled logic programs are less than one order of magnitude slower than hand-written C programs and much faster than interpreted languages such as Python.

1 Introduction

Linear Meld (LM) is a linear logic programming language aimed for the parallel implementation of graph-based algorithms [2]. LM is a high-level declarative language that offers a concise and expressive framework to define graph based algorithms that are provably correct. LM has been applied to a wide range of problems and machine learning algorithms, including: belief propagation [6], belief propagation with residual splash [6], PageRank, graph coloring, N-Queens, shortest path, diameter estimation, map reduce, quick-sort, neural network training, minimax, and many others.

Like Datalog, LM is a *forward-chaining* logic programming language since computation is driven by a set of inference rules that are used to update a database of logical facts. In Datalog, programs are monotonic and therefore the database grows in size as more facts are inferred from the logical rules. In LM, logical facts are linear and thus can be retracted when a rule is inferred. The use of linear facts greatly increases the power of the language but also increases the complexity of the implementation since database facts are retracted often.

In previous work [3], we have described the implementation of the LM virtual machine, including its data structures and how programs are parallelized. In this paper, we describe our compilation strategy and how we have refitted the

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 34–49, 2015.
DOI:10.1007/978-3-319-19686-2_3

runtime system to allow stand-alone compilation of programs by transforming logical rules into C++ code.

Our goal was to reduce the overhead of executing interpreted byte code and better understand the effectiveness and limitations of the compilation scheme. We present an algorithm that compiles logical rules, including comprehensions and aggregates, into efficient iterator-based C++ code. The compiler supports rule priorities, allowing the programmer to order rules based on their priority of inference. To the best of our knowledge, this is the first available compilation strategy for a linear logic language that supports these 3 features combined. The contributions of this paper are then three-fold: (1) a novel algorithm to compile prioritized linear logic rules with aggregates and comprehensions; (2) the interplay between the database layout and compiled code; and (3) comparison and analysis of our compilation with hand-written C programs and interpreted code. Experimental results show that our compiled programs are only 1 to 5 times slower than hand-written C programs.

The remainder of the paper is organized as follows. First, we briefly introduce the LM language. Next, we present an overview of the runtime support available to compiled rules and we discuss our contributions which include the algorithm for compiling rules into efficient iterator-based C++ code, and related work. We then present experimental results comparing our compiled programs with the old implementation and with hand-written C programs. The paper finishes with some conclusions.

2 Linear Meld

LM is a forward-chaining linear logic programming language that allows logical facts to be asserted and retracted in a structured fashion. A LM program can be seen as a graph of nodes, where each node contains a database of facts. The program is written as a set of inference rules that apply over the facts of a node.

LM rules have the form `a(X), b(Y) -o c(X, Y)` and can be read as follows: if fact `a(X)` and fact `b(Y)` exist in the database then fact `c(X, Y)` is added to the database. The expression `a(X), b(Y)` is called the *body* of the rule and `c(X, Y)` is called the *head* of the rule. A fact is a predicate, e.g., `a`, `b` or `c`, and its associated tuple of values, e.g., the concrete values of `X` and `Y`. Since LM uses linear logic as its foundation, we distinguish between *linear* and *persistent facts*. Linear facts are consumed (deleted) during the process of deriving a rule, while persistent facts are not. Program execution starts by adding the initial facts (called the axioms) to the database. Next, rules are recursively applied and the database is updated by adding new facts or deleting facts used during rule derivation. When no more rules are applicable, the program terminates. Rules have a defined priority (their position in the source file) and highest priority rules are fired first. If a new fact is derived and there is a set of applicable rules to be fired, the higher priority rule is selected before the others.

To make these ideas concrete, Fig. 1 presents a simple example for the single source shortest path (SSSP) program. The program computes the shortest distance from node `@1` to all other nodes in the graph. The SSSP program starts

```
1  type route edge(node, node, int).
2  type linear shortest(node, int, list int).
3  type linear relax(node, int, list int).
4
5  !edge(@1, @2, 3). !edge(@1, @3, 1).
6  !edge(@3, @2, 1). !edge(@3, @4, 5).
7  !edge(@2, @4, 1).
8  shortest(A, +00, []).
9  relax(@1, 0, [@1]).
10
11 shortest(A, D1, P1), D1 > D2, relax(A, D2, P2)
12    -o shortest(A, D2, P2),
13       {B, W | !edge(A, B, W) | relax(B, D2 + W, P2 ++ [B])}.
14
15 shortest(A, D1, P1), D1 <= D2, relax(A, D2, P2)
16    -o shortest(A, D1, P1).
```

Fig. 1. Single Source Shortest Path program code

(lines 1-3) with the declaration of the predicates. Predicates specify the facts used in the program. The first predicate, `edge`, is a persistent predicate that describes the relationship between the nodes of the graph, where the third argument represents the weight of the edge (the `route` modifier informs the compiler that the `edge` predicate determines the structure of the graph). The predicates `shortest` and `relax` are specified as linear facts and thus are deleted when deriving new facts. In the example, every node has a `shortest` fact that can be improved with new `relax` facts. Lines 5-9 declare the axioms of the program: `edge` facts describe the graph; `shortest(A, +00, [])` is the initial shortest distance (infinity) for all nodes; and `relax(@1, 0, [@1])` starts the algorithm by setting the distance from `@1` to `@1` to be 0.

The first rule of the program (lines 11-13) reads as follows: if the current `shortest` path `P1` with distance `D1` is larger than a new `relax` path with distance `D2`, then replace the current shortest path with `D2`, delete the new `relax` and propagate new paths to the neighbors (line 13) using a *comprehension*. The comprehension iterates over the edges of node `A` and derives a new `relax` fact for each node `B` with the distance `D2 + W`, where `W` is the weight of the edge.

The second rule of the program (lines 15-16) is read as follows: if the current shortest distance `D1` is shorter than a new `relax` distance `D2`, then delete the new `relax` fact and keep the current shortest path. Figure 2 shows a graphical representation of the application of the SSSP program rules.

2.1 LM Syntax

The abstract syntax for LM programs is presented in Fig. 3. A LM rule is written as $BE \multimap HE$ where BE is the body and HE is the head of the rule. The body may contain linear (L) and persistent (P) *fact expressions* and *constraints* (C). Fact expressions instantiate facts from the database and contain variables as arguments that may or may not be bound to concrete values or to other variables. Variables in the body of the rule can also be used in the head when instantiating facts. Constraints are essential for matching rules since they represent database

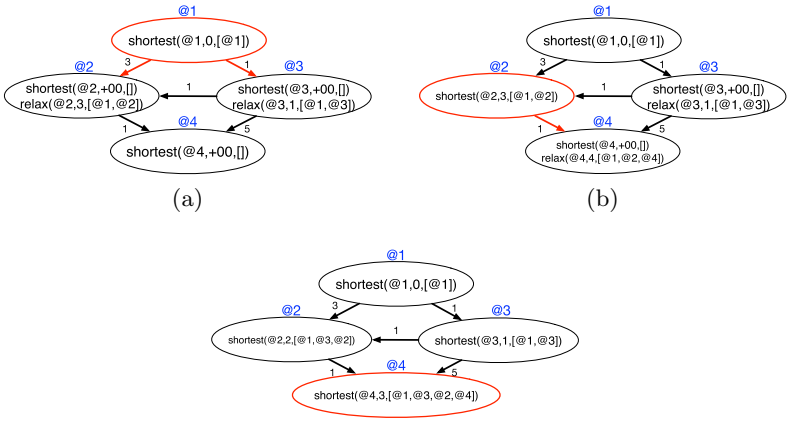


Fig. 2. Graphical representation of the SSSP program: (a) represents the program after propagating the initial distance at node `@1`, followed by (b) where the first rule is applied in node `@2` and by (c) that represents the final state of the program, where all the shortest paths have been computed.

Program	$Prog$	$::= \Sigma, D$
List Of Rules	Σ	$::= \cdot \mid \Sigma, R$
Database	D	$::= \Gamma; \Delta$
Rule	R	$::= BE \multimap HE \mid \forall_x.R$
Body Expression	BE	$::= L \mid P \mid C \mid BE, BE \mid \mathbf{1}$
Head Expression	HE	$::= L \mid P \mid HE, HE \mid EE \mid CE \mid AE \mid \mathbf{1}$
Linear Fact	L	$::= l(\hat{x})$
Persistent Fact	P	$::= !p(\hat{x})$
Constraint	C	$::= c(\hat{x})$
Selector Operation	S	$::= \texttt{min} \mid \texttt{max} \mid \texttt{random}$
Comprehension	CE	$::= \{\ \widehat{x};\ SB;\ SH\ \}$
Aggregate	AE	$::= [\ A \Rightarrow y;\ \widehat{x};\ SB;\ SH_1;\ SH_2\ ]$
Aggregate Operation	A	$::= \texttt{min} \mid \texttt{max} \mid \texttt{sum} \mid \texttt{count} \mid \texttt{collect}$
Sub-Body	SB	$::= L \mid P \mid SB, SB \mid \exists_x.SB$
Sub-Head	SH	$::= L \mid P \mid SH, SH \mid \mathbf{1}$
Known Linear Facts	Δ	$::= \cdot \mid \Delta, l(\hat{t})$
Known Persistent Facts	Γ	$::= \cdot \mid \Gamma, !p(\hat{t})$

Fig. 3. Abstract syntax of LM

joins and database *selects*. While selects filter out possible combinations from the database, body constraints (C) further restrict combinations by acting as guards using small variables from fact expressions. Constraints use a small functional language that includes mathematical operations, boolean operations, external functions and literal values.

The head of a rule, HE, contains linear (L) and persistent (P) *fact templates* which are uninstantiated facts and will derive new facts. The head can also have *comprehensions* (CE) and *aggregates* (AE). All those expressions may use all the variables instantiated in the body.

Comprehensions are similar to the functional programming construct of the same name. Comprehensions are sub-rules that are applied for all possible combinations. In a comprehension $\{\ \widehat{x};\ SB;\ SH\ \}$[1], $\widehat{x}$ is a list of variables, SB is the body of the comprehension and SH is the head. The body SB is used to generate all possible combinations for the head SH, according to the facts in the database. An example was shown in Fig. 1 (line 13), where `!edge(A, B, W)` facts are iterated over in order to derive `relax(A, D2 + W, P2 ++ [B])` facts for each combination.

Aggregates build on top of comprehensions and allow the capture of values that appear in each combination of the sub-rules. This list of values is then combined using one operator into a single value and then used to derive a set of fact expressions. In the abstract syntax $[\ A \Rightarrow y;\ \widehat{x};\ SB;\ SH_1;\ SH_2\ ]$, A is the aggregate operation, $\widehat{x}$ is the list of variables introduced in BE and SH_1 and y is the variable in the body SB that represents the values to be aggregated using A. Like comprehensions, we use $\widehat{x}$ to try all the combinations of SB, but, in addition to deriving SH_1 for each combination, we aggregate the values represented by y into a new y variable that is used inside the head SH_2. LM provides several aggregate operations, including the `min` (minimum value), `max` (maximum value), `sum` (add all numbers), `count` (count combinations) and `collect` (collect items into a list). Consider, for example, the following rule:

```
const P = ... // number of nodes
const damp = ... // probability of random jump to another page (for PageRank computation)

update(A), pagerank(A, OldRank)
      -o [sum => V | B | neighbor-pagerank(A, B, V) | neighbor-pagerank(A, B, V) |
            pagerank(A, damp/P + (1.0 - damp) * V)].
```

The rule uses an aggregate to accumulate the sum of the neighbor's PageRank into a single value `V`. This aggregate value is then assigned to a new `pagerank` fact via the expression `damp/P + (1.0 - damp)*V`, where `V` is the result of adding all the `V` values in `neighbor-pagerank(A, B, V)` facts.

3 Supporting Runtime and Database Data Structures

In this section, we review the supporting runtime that is used by the compiler. We focus mostly on the structure of the nodes since inference rules are compiled from the point of view of the node data structure.

Figure 4 presents the layout of the node data structures. Each node of the graph stores 4 main data structures: (1) the *rule matching engine*; (2) a *fact buffer* for storing incoming and temporary facts; (3) the *database of linear facts*; and (4) the *database of persistent facts*.

[1] We substitute ; for | in the abstract syntax to avoid confusion with the grammar choice operator.

The rule engine maintains a simplified view of the two fact databases and efficiently decides which rules need to be executed. For instance, if a rule r needs facts `a` and `b` to be applied and the database already contains `a` facts, once a `b` fact is derived, the rule engine schedules r to be executed. The compiler is responsible for the code that is executed when a rule is scheduled. A compiled rule contains instructions to search and match facts from the database and to derive new facts when the body of the rule is matched.

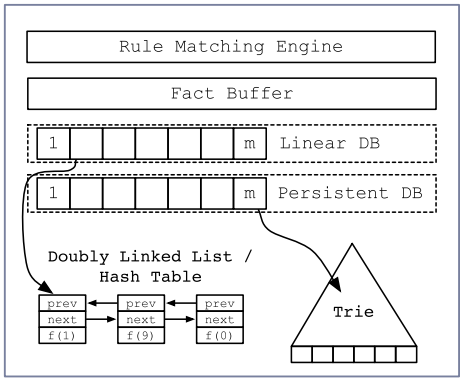


Fig. 4. Node data structures

In this context, the organization of the database structures is critical because linear facts can be retracted and asserted frequently. This means that the database needs to allow fast insertions and deletions but also needs to have reasonably fast mechanisms for lookup. The database of facts is partitioned by predicate, therefore, each predicate can have its own data structure depending on the patterns of access for that particular predicate. Linear facts are stored using the following data structures:

- *Doubly-Linked List Data Structures.* Each linear fact is a node of the linked list. Allows constant $\mathcal{O}(1)$ insertion and deletion of facts given the pointer of the target node. Although lookup operations take linear time, this is not critical since most predicates tend to have a small number of facts.
- *Hash Table Data Structures.* For predicates with many facts we use hash tables. Hash tables are efficient for repetitive lookup operations using a specific argument (i.e., searching for facts with a concrete value) and build upon lists by hashing facts using a specific argument and then using separate chaining with doubly-linked lists for collision resolution. Hash tables are, on average, $\mathcal{O}(1)$ for insertion, deletion and lookup, however they require more memory.

For persistent tuples, we use *Trie Data Structures*, which are trees where facts are indexed by a common prefix. Since persistent facts are never deleted, it's not expensive to index facts by a common prefix, which also tends to save memory in the long run.

4 Compiling Rules

In this section, we present the main algorithm of the compiler, that turns inference rules into C++ code, and we discuss the key optimizations for efficient code execution.

4.1 Constraints

After an inference rule is compiled, it must respect the *fact constraints* (facts must exist in the database) and the *join constraints* that can be represented by variable constraints and/or boolean expressions. For instance, consider again the second rule of the SSSP program presented in Fig. 1:

```
shortest(A, D1, P1), D1 <= D2, relax(A, D2, P2)
   -o shortest(A, D1, P1).
```

The fact constraints include the facts required to trigger the rule, namely `shortest(A, D1, P1)` and `relax(A, D2, P2)`, and the join constraints include the expression `D1 <= D2`. However, rules may also have other less obvious join constraints, such as variable constraints, as in the following rule:

```
new-neighbor-pagerank(A, B, New),
neighbor-pagerank(A, B, Old)
   -o neighbor-pagerank(A, B, New).
```

where variable `B` must have the same value in both body facts[2].

4.2 Iterators

The data structures for facts presented in Section 3 support the *iterator* pattern. For linked lists, the iterator goes through every fact in the list while the hash table iterator can either iterate through the whole table or iterate through a single bucket. A bucket iterator is in fact a linked list iterator that starts from a given argument. For tries, while the default iterator goes through every fact in the trie, it can be customized with a matching specification in order to reduce search. A matching specification includes argument assignments (e.g., argument $i = V$, where V is a concrete value).

Iterators are heavily used in the compiled code. For instance, the second rule of the SSSP program presented in Fig. 1 is compiled as follows:

```
1  for(auto it1(list("shortest").begin()); it1 != list("shortest").end(); ) {
2     fact *f1(*it1);
3     for(auto it2(list("relax").begin()); it2 != list("relax").end(); ) {
4        fact *f2(*it2);
5        if(f1->get_int(1) <= f2->get_int(1)) { // D1 <= D2
6           fact *new_shortest(new fact("shortest"));
7           new_shortest->set_int(1, f1->get_int(1));
8           new_shortest->set_list(2, f1->get_list(2));
9           // new fact was derived
10          list("shortest").push_back(new_shortest);
11          // deleting facts
12          it1 = list("shortest").erase(it1); // remove from list
13          it2 = list("relax").erase(it2);
14          return;
15       }
16       ++it2;
17    }
18    ++it1;
19 }
```

[2] Rule taken from an asynchronous PageRank program.

The compilation algorithm iterates through the fact expressions in the body of the rule and creates nested loops to try all the possible combinations of facts. For this rule, all pairs of `shortest` and `relax` facts must be matched until the constraint `D1 <= D2` is true. First, an iterator for `shortest` is created that will loop through all `shortest` facts in the list. Inside the loop, a nested iterator is created for predicate `relax`. This inner loop includes a check for the `D1 <= D2` constraint. If the constraint fails, another `relax` fact is then attempted by incrementing `it2`. Likewise, if the current `f1` fact fails for all `f2` facts, then `it1` is incremented in order to try the next `shortest` fact. Otherwise, if the constraint succeeds then the rule matches and a new `shortest` fact is derived. Additionally, the two used linear facts are retracted by erasing the iterators from the linked lists. Note that after the rule is derived, the code must return since there is a higher priority rule that may be triggered with the new `shortest` fact (see Fig. 2). This enforces the priority semantics of the language.

Figure 5 presents the algorithm for compiling rules into C++ code. First, we split the body of the rule into fact expressions and constraints. Fact expressions map directly to iterators while fact constraints map to *if* expressions. A possible compilation strategy is to first compile all the fact expressions and then compile the constraints. However, this may require unneeded database lookups since some constraints may fail early. Therefore, our compiler introduces constraints as soon as all the variables in the constraint are all included in the already compiled fact expressions. The order in which fact expressions are selected for compilation does not interfer with the correctness of the compiled code, thus our compiler selects the fact expressions (*RemoveBestFactExpr*) by their potential to activate constraints, therefore avoiding undesirable database lookups. If two fact expressions have the same number of new constraints, then the compiler always picks the persistent fact expression since persistent facts are not deleted.

Derivation of new facts belonging to the local node implies adding the new fact to the local node data structure. Facts that belong to other nodes are sent using an appropriate runtime API.

4.3 Persistence Checking

Not all linear facts need to be deleted. For instance, in the compiled rule above, the fact `shortest(A, D1, P1)` is re-derived in the head. Our compiler is able to turn linear loops into persistent loops for linear facts that are retracted and then asserted. The rule is then compiled as follows:

```
1  for(auto it1(list("shortest").begin()); it1 != list("shortest").end(); ) {
2     fact *f1(*it1);
3     for(auto it2(list("relax").begin()); it2 != list("relax").end(); ) {
4        fact *f2(*it2);
5        if(f1->get_int(1) <= f2->get_int(1)) {
6           it2 = list("relax").erase(it2);
7           goto next;
8        }
9        ++it2;
10 next: continue;
11    }
12    ++it1;
13 }
```

```
Data: Rule R1, Rules
Result: Compiled Code
FactExprs ⟵ FactExprsFromRule(R1);
Constraints ⟵ ConstraintsFromRule(R1);
Code ⟵ CreateFunctionForRule();
Iterators ⟵ [];
CompiledFacts = [];
while FactExprs not empty do
    Fact ⟵ RemoveBestFactExpr(FactExprs);
    CompiledFacts.push(Fact);
    Iterator ⟵ Code.InsertIterator(Fact);
    Iterators.push(Iterator);
    /* Select constraints that are covered by CompiledFacts.       */
    NextConstraints ⟵ RemoveConstraints(Constraints, CompiledFacts);
    Code.InsertConstraints(NextConstraints);
end
HeadFacts = HeadTemplatesFromRule(R1);
while HeadFacts not empty do
    Fact ⟵ RemoveFact(HeadFacts);
    Code.InsertDerivation(Fact);
end
for Iterator ∈ Iterators do
    if IsLinear(Iterator) then
        Code.InsertRemove(Iterator);
    end
end
/* Enforce rule priorities.                                        */
if FactsDerivedUsedBefore(Head, Program, R1) then
    Code.InsertReturn();
else
    Code.InsertGoto(FirstLinear(Iterators));
end
return Code
```

Fig. 5. Compiling LM rules into C++ code

In this new version of the code, only the `relax` facts are deleted, while the `shortest` facts remain untouched. In the SSSP program, each node has one `shortest` fact and this compiled code simply filters out the `relax` facts with the distances that are equal or greater than the current best distance. Note that now we have a *goto statement* (line 7) that is executed when the rule is fired. In this case, since no new `shortest` fact was derived, we avoid returning to enforce rule priorities and continue to try to fire the rule as many times as possible.

All the rule combinations are attempted in cases where a rule does not derive any facts or the facts derived do not appear before the rule, that is, the new facts are only used in lower priority rules. This is specified in the final *if statement* in Fig. 5. If the rule does not return, then we always jump to the first loop that

uses linear facts. We must jump to the first linear loop because we cannot use the next fact from the deepest loop since we may have constraints between the first linear loop and the deepest loop that were previously validated using facts that were deleted in the meantime.

4.4 Updating Facts

Many inference rules retract and then derive the same predicate but with different arguments. The compiler recognizes those cases and instead of retracting the fact from its linked list or hash table, it updates the fact in-place. As an example, consider the following rule:

```
new-neighbor-pagerank(A, B, New),
neighbor-pagerank(A, B, Old)
   -o neighbor-pagerank(A, B, New).
```

Assuming that `neighbor-pagerank` is stored in a hash table and indexed by the second argument, the code for the rule above is as follows:

```
1   for(auto it1(list("new-neighbor-pagerank").begin()); it1 !=
2      list("new-neighbor-pagerank").end(); )
3   {
4      fact *f1(*it1);
5      // hash table for neighbor-pagerank is indexed by the second argument therefore
6      // we search for the bucket using the second argument of new-neighbor-pagerank
7      hash_bucket bucket(hash_table("neighbor-pagerank").find(f1->get_node(1));
8      for(auto it2(bucket.begin()); it2 != bucket.end(); ) {
9         fact *f2(*it2);
10        if(f1->get_node(1) == f2->get_node(1)) {
11           f2->set_float(2, f1->get_float(2)); // update neighbor-pagerank
12           it1 = list("new-neighbor-rank").erase(it1);
13           goto next;
14        }
15        ++it2;
16     }
17     ++it1;
18  next: continue;
19  }
```

Note that `neighbor-pagerank` is updated using `set_float`. The rule also does not return since this is the highest priority rule. If there was a higher priority rule using `neighbor-pagerank`, then the code would have to return since an updated fact represents a new fact.

4.5 Enforcing Linearity

We have already introduced the *goto* statement as a way to avoid reusing retracted linear facts. However, this is not enough in order to enforce linearity of facts. Consider the following inference rule:

```
add(A, N1), add(A, N2) -o add(A, N1 + N2).
```

Using the standard compilation algorithm, two nested loops are created, one for each `add` fact. However, notice that there is an implicit constraint when

creating the iterator for `add(A, N2)` since this fact cannot be the same as the first one. That would invalidate linearity since a single linear fact would be used to prove two linear facts. This is easily solved by adding a constraint for the inner loop that ensures that the two facts are different (line 5).

```
1  for(auto it1(list("add").begin()); it1 != list("add").end(); ) {
2     fact *f1(*it1);
3     for(auto it2(list("add").begin()); it2 != list("add").end(); ) {
4        fact *f2(*it2);
5        if(f1 != f2) {
6           f1->set_int(1, f1->get_int(1) + f2->get_int(1));
7           it2 = list("add").erase(it2);
8           goto next;
9        }
10       ++it2;
11    }
12    ++it1;
13 next: continue;
14 }
```

Figure 6 presents the steps for executing this rule when the database contains three facts. Initially, the two iterators point to the first and second facts and the former is updated while the latter is retracted. The second iterator then moves to the next fact and the first fact is updated again, now to the value `6`, the expected result.

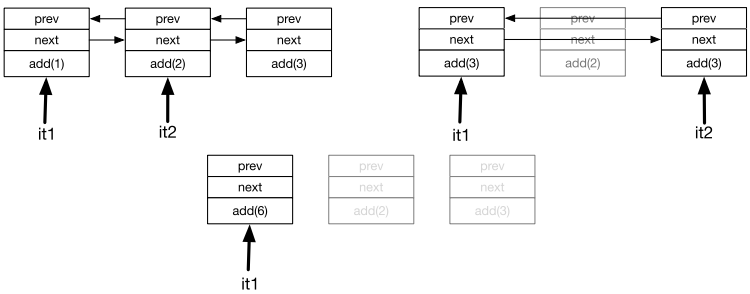


Fig. 6. Executing the add rule

4.6 Comprehensions

Comprehensions were initially presented in the first rule of the SSSP program.

```
shortest(A, D1, P1), D1 > D2, relax(A, D2, P2)
   -o shortest(A, D2, P2), {B, W | !edge(A, B, W) | relax(B, D2 + W, P2 ++ [B])}.
```

The attentive reader will remember that comprehensions are sub-rules, therefore they should be compiled like normal rules. However, they do not need to return due to rule priorities since all the combinations of the comprehension must be derived. However, the rule itself must return if any of its comprehensions has derived a fact that is used by a higher priority rule. In the case of the

above example, the rule does not need to return since it has the highest priority and the `relax` facts derived in the comprehension are all sent to other nodes. The code for the rule is shown below:

```
for(auto it1(list("shortest").begin()); it1 != list("shortest").end(); ) {
   fact *f1(*it1);
   for(auto it2(list("relax").begin()); it2 != list("relax").end(); ) {
      fact *f2(*it2);
      if(f1->get_int(1) > f2->get_int(1)) {
         // comprehension code
         for(auto it3(trie("edge").begin()); it3 != trie("edge").end(); ) {
            fact *f3(*it3);
            fact *new_relax(new fact("relax"));
            new_relax->set_int(1, f2->get_int(1) + f3->get_int(2));
            new_relax->set_list(append(f2->get_list(2), list(f3->get_node(1))));
            send_fact(new_relax, f3->get_node(1));
            ++it3;
         }
         f1->set_int(1, f2->get_int(1));
         f1->set_list(2, f2->get_list(2));
         it2 = list("relax").erase(it2);
         goto next;
      }
      ++it2;
   }
   ++it1;
next: continue;
}
```

Special care must be taken when the comprehension's sub-rule uses the same predicates that are derived by the main rule. Rule inference must be atomic in the sense that after a rule matches, the comprehensions in the head of the rule can use the facts that were present before the body of the rule was matched. Consider a rule with n comprehensions or aggregates, where CB_i and CH_i are the body and head of the comprehension/aggregate, respectively, and H represents the fact templates found in the head of the rule. The formula used by the compiler to detect conflicts between predicates is the following:

$$\bigcup_{i}^{n}[CB_i \cap H] \cup \bigcup_{i}^{n}[CB_i \cap \bigcup_{j}^{n}[CH_j]]$$

If the result of the formula is not empty, then the compiler disables optimizations for the conflicting predicates and derives the corresponding facts into the fact buffer that are then added back into the database. Fortunately, most rules in LM programs do not show conflicts and thus can be fully optimized.

4.7 Aggregates

Aggregates are similar to comprehensions. They are also sub-rules but a value is accumulated for each combination of the sub-rule. After all the combinations are inferred, a final head term is derived with the accumulated term. Consider again the following PageRank rule:

```
update(A), pagerank(A, OldRank)
      -o [sum => V | B | neighbor-pagerank(A, B, V) | neighbor-pagerank(A, B, V) |
            pagerank(A, damp/P + (1.0 - damp) * V)].
```

The variable `V` is initialized to `0.0` and sums all the PageRank values of the neighbors as seen in the code below. The aggregate value is then used to update the second argument of the initial `pagerank` fact.

```
1  for(auto it1(list("pagerank").begin()); it1 != list("pagerank").end(); ) {
2     fact *f1(*it1);
3     for(auto it2(list("update").begin()); it2 != list("update").end(); ) {
4        fact *f2(*it2);
5        double acc(0.0); // aggregate accumulator.
6        for(auto it3(list("neighbor-pagerank").begin()); it3 !=
7              list("neighbor-pagerank").end(); ) {
8           fact *f3(*it3);
9           acc += f3->get_float(2);
10          ++it3; // the sub-rule has no head since neighbor-pagerank is re-derived
11       }
12       // head of the aggregate
13       f1->set_float(1, damp / P + (1.0 - damp) * V);
14       goto next;
15    }
16    ++it1;
17 next: continue;
18 }
```

5 Related Work

LM shares many similarities [1] with Constraint Handling Rules (CHR) [5]. CHR is a concurrent committed-choice constraint language used to write constraint solvers. A CHR program is a set of rules and a set of constraints. The constraint store can be seen as a database of facts and rules manipulate the constraint store. Many basic optimizations used in the LM compiler such as join optimizations and the use of different data structures for indexing facts were inspired in work done on CHR [7]. Wuille et al. [9] have described a CHR to C compiler that follows some of the ideas presented here and De Koninck et al. [4] showed how to compile CHR programs with dynamic priorities into Prolog. Our work distinguishes itself from these two works by supporting a novel combination of comprehensions, aggregates and rule priorities. Compilation of LM programs is also novel due to the implicit parallelism of rules, allowing for programs to be parallelized [2].

6 Experimental Results

This section presents experimental results for our compilation strategy. We compare the execution speed of our new compiled code against hand-written implementations in C of the same programs. We also compare the results against interpreted execution in order to help us understand the limitations of the compilation scheme when removing the interpretation overhead.

For our experimental setup, we used a computer with a 24 (4x6) Core AMD Opteron(tm) Processor 8425 HE @ 800 MHz with 64 GBytes of RAM memory running the Linux kernel 3.15.10-201.fc20.x86_64. The C++ compiler used is GCC 4.8.3 (g++) with the flags: `-O3 -std=c+0x -march=x86-64`. We run all experiments 3 times and averaged the execution time.

We have implemented 5 different LM programs and their corresponding C versions. The programs are the following:

- Shortest Path (SP): a slightly modified version of the program presented in Fig. 2, where the shortest distance is computed from all nodes to all nodes.
- N-Queens: the classic puzzle for placing queens on a chess board so that no two queens threaten each other.
- Belief Propagation: a machine learning algorithm to denoise images.
- Heat Transfer: an asynchronous program that performs transfer of heat between nodes.
- MiniMax: the AI algorithm for selecting the best player move in a Tic-Tac-Toe game. The initial board was augmented in order to provide a longer running benchmark.

Table 1 presents experimental results comparing the compiled and interpreted code versions against the C program versions. Comparisons to other systems are shown under the **Other** column. Note that for some programs, we present different program sizes shown in ascending order.

Table 1. Experimental results comparing different programs against hand-written versions in C. For the C versions, we show the execution time in seconds (column **C Time** (s)). For the other approaches, we show the overhead ratio compared with the corresponding C version. The overhead numbers (**lower is better**) are computed by dividing the execution time of the approach on that column by the execution time of the similar hand-written version in C.

Program	**Size**	**C Time** (s)	**Compiled**	**Interpreted**	**Other**
Shortest Path	US Airports	0.1	3.9	13.9	13.3 (python)
	OCLinks	0.4	5.6	14.2	11.2 (python)
	Powergrid	0.9	3.5	11.3	10.6 (python)
N-Queens	11	0.2	1.4	3.9	20.8 (python)
	12	1.3	3.2	5.3	24.1 (python)
	13	7.8	3.8	6.6	26.0 (python)
	14	49	4.5	8.9	28.0 (python)
Belief Propagation	50	2.8	1.3	1.4	1.1 (GL)
	200	51	1.3	1.4	1.1 (GL)
	300	141	1.3	1.4	1.1 (GL)
	400	180	1.3	1.4	1.1 (GL)
Heat Transfer	80	7.3	4.6	9.9	-
	120	32	5.3	10.5	-
MiniMax	-	7.3	3.2	7.1	9.3 (python)

The Shortest Path program shows good improvements from the interpreted version, since the run time is reduced between 61% and 72%. The good performance results come from the fact that the program performs repeated comparisons between integer numbers, which tend to be slower in interpreted code, and from the fact that the program has only two rules where the shortest distance

fact is updated or kept. The distance facts are also indexed by the source node, which helps the code filter through the candidate distances faster. This is helpful since the program computes the shortest distance between pairs of nodes.

N-Queens presents some scalability issues for our compilation scheme due to the exponential increase of facts as the problem size increases. The same behavior can be observed for the Python programs. Regarding the comparison with the interpreted version, the compiled version reduces the interpreted run time by almost 50% which indicates that there are more database operations in N-Queens than in Shortest Path.

The Belief Propagation program is made of many expensive floating point calculations. The interpreted version used external functions written in C to implement those operations because otherwise it would be too slow. Therefore, and since the rules tend to manipulate a small number of facts, the interpreted and compiled versions perform about the same. This program has also the best results which proves that the program spends a huge amount of time performing floating point calculations. For comparison purposes, we used GraphLab [8] (GC in the table), an efficient machine learning framework for writing parallel graph-based machine learning algorithms in C++. GraphLab's version of the algorithm is slightly slower than the C version.

The Heat Transfer program also performs floating point operations but in a much smaller scale than Belief Propagation. This is noticiable from the results since the slowdown is much larger than Belief Propagation. The program also needs to compute many sum aggregates, which makes the interpreted version incur in some overhead due to the integer operations.

While all the other programs perform computations on a pre-defined set of nodes, the MiniMax program creates the nodes of the graph dynamically. Creating new nodes requires creating new databases which tends to take a considerable fraction of the run time. However, we have seen a good reduction in run time when compared to the interpreted version, which we think is the result of low-level optimizations that were applied in the compiled version.

It should be noted that in these programs there is a parallelization overhead since LM's supporting runtime is designed to explore parallelism implicitly. For instance, we measured a 20% overhead for N-Queens, a program that needs to reference count many lists during run time. Fortunately, if the programmer takes advantage of the parallel facilities of LM, she will be able to run most of these programs faster than C by using between 2 and 4 threads.

7 Conclusions

In this paper, we have presented a compilation strategy for linear logic programs with comprehensions, aggregates and rule priorities. Rule priorities allow the programmer to assign priorities to rules so that higher priority rules are applied before lower priority rules, while comprehensions and aggregates allow a more expressive way for the programmer to iterate through the database to derive new facts or aggregate data. To the best of our knowledge, our compilation strategy

is the first to consider programs with these three important features and the first efficient compilation strategy for forward-chaining linear logic programs. We have also implemented and described important optimizations such as fact updates and persistence checking and the importance of choosing the right data structures for the needs of linear logic programs. Our experimental results show that LM is competitive when compared to hand-written C programs.

Acknowledgments. This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program and project SIBILA (NORTE-07-0124-FEDER-000059). Flavio Cruz is funded by the FCT grant SFRH/BD/51566/2011.

References

1. Betz, H., Frühwirth, T.: A linear-logic semantics for constraint handling rules. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 137–151. Springer, Heidelberg (2005)
2. Cruz, F., Rocha, R., Goldstein, S., Pfenning, F.: A Linear Logic Programming Language for Concurrent Programming over Graph Structures. Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue pp. 493–507 (July 2014)
3. Cruz, F., Rocha, R., Goldstein, S.C.: Design and implementation of a multithreaded virtual machine for executing linear logic programs. In: International Symposium on Principles and Practice of Declarative Programming, pp. 43–53. ACM Press, September 2014
4. De Koninck, L., Stuckey, P.J., Duck, G.J.: Optimizing compilation of chr with rule priorities. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 32–47. Springer, Heidelberg (2008)
5. Frühwirth, T.: Constraint handling rules. In: Podelski, A. (ed.) Constraint Programming: Basics and Trends. LNCS, vol. 910, pp. 90–107. Springer, Heidelberg (1995)
6. Gonzalez, J., Low, Y., Guestrin, C.: Residual splash for optimally parallelizing belief propagation. In: Artificial Intelligence and Statistics (2009)
7. Holzbaur, C., de la Banda, M.J.G., Stuckey, P.J., Duck, G.J.: Optimizing compilation of constraint handling rules in HAL. CoRR cs.PL/0408025 (2004)
8. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: a new framework for parallel machine learning. In: Conference on Uncertainty in Artificial Intelligence (UAI), pp. 340–349 (2010)
9. Wuille, P., Schrijvers, T., Demoen, B.: CCHR: the fastest CHR implementation, in C. In: Workshop on Constraint Handling Rules, pp. 123–137 (2007)

Declaratively Solving Google Code Jam Problems with Picat

Sergii Dymchenko(✉) and Mariia Mykhailova

Independent Researcher, Bellevue, WA, USA
sdymchenko@progopedia.com, michaylova@gmail.com

Abstract. In this paper we present several examples of solving algorithmic problems from the Google Code Jam programming contest with Picat programming language using declarative techniques: constraint logic programming and tabled logic programming. In some cases the use of Picat simplifies the implementation compared to conventional imperative programming languages, while in others it allows to directly convert the problem statement into an efficiently solvable declarative problem specification without inventing an imperative algorithm.

1 Introduction

Google Code Jam[1] (GCJ) is one of the biggest programming competitions in the world: almost 50,000 participants registered in 2014, and 25,462 of them solved at least one task.

GCJ competitors are allowed to use any freely available programming language or system (including Picat [2] described in this paper). We show examples of solving GCJ problems with Picat using constraint logic programming and tabled logic programming.

Picat is a new logic-based multi-paradigm programming language. Picat shares many features with Prolog, especially B-Prolog [4], but also has many distinct features: optional destructive assignments, functions in addition to predicates, explicit non-determinism, list comprehensions [5].

Picat Implementation of TPK Algorithm

To give an idea of Picat syntax to a reader unfamiliar with the language we present an implementation of TPK algorithm. TPK is an algorithm proposed by D. E. Knuth and L. T. Pardo [2] used to show basic syntax of a programming language. The algorithm allows a user to input 11 real numbers $(a_0 \dots a_{10})$. After that for $i = 10 \dots 0$ (in that order) the algorithm computes value $y = f(a_i)$, where $f(t) = \sqrt{|t|} + 5t^3$, and outputs a pair (i, y) if $y \leq 400$, or (i, TOO LARGE) otherwise.

[1] https://code.google.com/codejam
[2] http://picat-lang.org/

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 50–57, 2015.
DOI:10.1007/978-3-319-19686-2_4

```
1 f(T) = sqrt(abs(T)) + 5 * T**3.
2 main =>
3     N = 11,
4     As = to_array([read_real() : I in 1..N]),
5     foreach (I in N..-1..1)
6         Y = f(As[I]),
7         if Y > 400 then
8             printf("%w TOO LARGE\n", I - 1)
9         else
10            printf("%w %w\n", I - 1, Y)
11        end
12    end.
```

Listing 1.1. TPK algorithm in Picat

Line 1 defines a function to calculate the value of f (a function in Picat is a special kind of a predicate that always succeeds with a return value). Lines 2–12 define the `main` predicate. Line 4 uses list comprehension to read 11 space-separated real numbers into array `As`. Line 5 defines a header of `foreach` loop: `I` goes from 11 to 1 with the step -1 (in Picat array indices are 1-based). Lines 6–11 calculate the value of y and print the result using an 'if-then-else' construct. `printf` is similar to the corresponding C language function; `%w` can be seen as a "wildcard" control sequence to output values of different types.

2 The Problems

For this section we have chosen a set of GCJ problems from different years to demonstrate different useful aspects of Picat: constraint programming, top-down dynamic programming with tabling, and the `planner` module.

Triangle Areas[3]

"Triangle Areas" is a problem from the round 2 of GCJ 2008. The problem gives integers N, M and A and asks to find any triangle with vertices in integer points with coordinates $0 \le x_i \le N$ and $0 \le y_i \le M$ that has an area $S = \frac{A}{2}$, or to decide that it does not exist.

"Triangle Areas" is almost perfect for solving with constraint logic programming. Variables are discrete, constraints are non-linear, and we are looking for any feasible solution. To come up with an effective model we need to notice that one vertex of the triangle can be chosen arbitrarily. With this observation, the most convenient way to calculate the doubled triangle area is to place one vertex in $(0, 0)$; then $2S = A = |x_2 y_3 - x_3 y_2|$. (The same formula can be used in an imperative solution.)

For this problem we present complete source code of the solution. For subsequent problems we omit the `main` predicate to save space.

[3] Problem link: http://goo.gl/enHWlq

```
1 import cp.
2 import util.
3 model(N, M, A, Points) =>
4     [X2, X3] :: 0..N,
5     [Y2, Y3] :: 0..M,
6     A #= abs(X2 * Y3 - X3 * Y2),
7     Points = [X2, Y2, X3, Y3].
8 do_case(Case_num, N, M, A) =>
9     printf("Case #%w: ", Case_num),
10    if model(N, M, A, Points), solve(Points) then
11        printf("0 0 %s\n", join([to_string(V) : V in Points]))
12    else
13        println("IMPOSSIBLE")
14    end.
15 main =>
16    C = read_int(),
17    foreach (Case_num in 1..C)
18        N = read_int(), M = read_int(), A = read_int(),
19        do_case(Case_num, N, M, A)
20    end.
```

Listing 1.2. Complete Picat program for the "Triangle Areas" problem

Lines 1–2 load Picat modules for constraint programming and utility functions. Lines 3–7 define the model with input parameters `N`, `M`, `A` and a list of output parameters `[X2, Y2, X3, Y3]`. `::` and `#=` are from the 'cp' module. With `::` we define possible domains for `X2, X3, Y2, Y3` variables, and `#=` constrains both left and right parts to be equal. After model evaluation `X2, X3, Y2, Y3` variables will not necessarily be instantiated to concrete values, but they will have reduced domains with possible delayed constraints and will be instantiated later with `solve`.

Lines 8–14 define the `do_case` predicate to process a single input case. Line 9 outputs case number according to the problem specification. Lines 10–14 are an 'if-then-else' construct that outputs point coordinates if it is possible to satisfy our `model` predicate and solve (assign concrete values from the domain to every variable) the constraint satisfaction problem, or "IMPOSSIBLE" otherwise. Line 11 uses an interesting Picat feature – list comprehension – which is very similar to what Python and many other modern programming languages have.

Lines 15–20 define the `main` predicate that reads the number of test cases `C` and for each test case reads `N, M, A` parameters and executes `do_case`.

This Picat program is very similar to our constraint programming solution in ECL$^{\text{i}}$PS$^{\text{e}}$ CLP [1].

A possible imperative solution in a mainstream programming language requires a more in-depth analysis of the problem. First observations will be the same. We will also note that it is impossible to find required triangle if $A > M \times N$, and for $A = M \times N$ triangle $(0,0),(N,0),(0,M)$ is a valid answer. Now, for $A < M \times N$ we can represent A as $M(A \operatorname{div} M) + (A \bmod M)$, $0 < A \operatorname{div} M < N, 0 <$

$A \bmod M < M$. If we match this representation with the area formula, we can see that points $(0,0), (1,M)$, and $(-A \operatorname{div} M, A \bmod M)$ form a triangle with area $\frac{A}{2}$. If we shift this triangle $A \operatorname{div} M$ units in positive direction along the x axis, we will get a triangle $(A \operatorname{div} M, 0), (A \operatorname{div} M + 1, M), (0, A \bmod M)$ that will match all the requirements.

Arguably, our declarative solution in Picat is simpler and leaves less space for a possible mistake.

Interestingly, in our tests the running time of our solution in Picat 0.9 on small input is about 2.5 times larger than on the large input (table 1). This is probably related to the implementation details and could change in the future versions.

Welcome to Code Jam[4]

"Welcome to Code Jam" is a problem from the qualification round of GCJ 2009. The task is to calculate the last 4 digits of the number of times the string "welcome to code jam" (S) appears as a subsequence of the given string (T).

This is a typical dynamic programming problem. The problem state $dp[i][j]$ is the number of times the substring of T of length i contains the substring of S of length j (modulo 10000). The recurrence relation is: if $T[i] = S[j]$, $dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$, otherwise $dp[i][j] = dp[i-1][j]$.

Our Picat program uses tabling [3] (a kind of memoization) to implement the described dynamic programming solution in a top-down fashion.

```
1  s() = to_array("welcome to code jam").
2  table
3  ways(_, _, 0) = 1.
4  ways(_, 0, _) = 0.
5  ways(T, I, J) = W =>
6      S = s(),
7      if T[I] == S[J] then
8          W = (ways(T, I - 1, J) + ways(T, I - 1, J - 1)) mod 10000
9      else
10         W = ways(T, I - 1, J)
11     end.
12 do_case(Case_num, T) =>
13     W = ways(T, length(T), length(s())),
14     printf("Case #%w: %04d\n", Case_num, W).
```

Listing 1.3. Picat solution for the "Welcome to Code Jam" problem

Line 1 defines the string S from the problem statement as a functional fact. Lines 2–11 defines recursive `ways` function for dynamic programming. The calls to this function are automatically tabled (memoized) because of the `table` declaration. Lines 3 and 4 describe base cases for the recursion. Lines 5–11 specify the

[4] Problem link: http://goo.gl/qeLls4

recurrence relation. The `do_case` predicate in lines 12–14 calls the `ways` function and prints the results according to the problem specification.

An imperative solution can rely on the same recurrence relation, but might require more code to implement it either as a bottom-up dynamic programming or as a top-down recursion with memoization.

Bribe the Prisoners[5]

"Bribe the Prisoners" was the hardest problem from the round 1C of GCJ 2009. In it we have an array of P prison cells, each cell is either empty or contains a prisoner. Every time a prisoner from one of the cells is released, all prisoners housed on either side of that cell until cell 1, cell P, or an empty cell get one coin each. Initially all cells contain prisoners. Given a list of indices of prisoners to be released, find the minimum total number of coins that will be spent if the prisoners will be released in an optimal order.

This is an another dynamic programming problem. For each pair of cells $A \leq B$, $dp[A][B]$ is the best answer if prisoners occupy only cells from A to B, inclusive. If the first prisoner between A and B to be released is in cell X, $(B - A)$ coins are to be paid out immediately after his release, and then the smaller subproblems $dp[A][X-1]$ and $dp[X+1][B]$ have to be solved. The final answer $dp[1][P]$ corresponds to the initial state of all cells occupied.

Our Picat program uses mode-directed tabling [6].

```
1 table (+, +, +, min)
2 cost(A, B, FreeList, Cost) ?=>
3     foreach(X in FreeList)
4         (X < A ; X > B)
5     end,
6     Cost = 0.
7 cost(A, B, FreeList, Cost) =>
8     member(X, FreeList),
9     X >= A, X =< B,
10    cost(A, X - 1, FreeList, CostLeft),
11    cost(X + 1, B, FreeList, CostRight),
12    Cost = B - A + CostLeft + CostRight.
13 do_case(Case_num, P, FreeList) =>
14    cost(1, P, FreeList, Cost),
15    printf("Case #%w: %w\n", Case_num, Cost).
```

Listing 1.4. Picat solution for the "Bribe the Prisoners" problem

The first line declares the tabling mode for the `cost` predicate: first 3 parameters are input parameters, and the last parameter is an output parameter (the cost of releasing all prisoners in `FreeList` that occupy cells in the $[A; B]$ range) that must be minimized. Lines 2–12 define two clauses of the `cost` predicate; the predicate is non-deterministic, and the first rule is declared backtrackable using

[5] Problem link: http://goo.gl/pSbrTk

`?=>` syntax instead of `=>`. The first clause states that if no prisoners in `FreeList` occupy cells between A and B, the cost of their release will be 0. The second clause calculates the release cost of prisoner X as described by the recurrence relation. The `do_case` predicate in lines 13–15 calls the `cost` function for the whole range of cells and prints the result according to the problem specification.

As with the previous problem, an imperative solution can use the same recurrence relation, but might require more code for a bottom-up or top-down approach implementation, including explicit comparison of release costs of different prisoners to find the minimum. Our Picat solution replaces most of the auxiliary code with a single `table` declaration.

Osmos[6]

"Osmos" is a problem from the round 1B of GCJ 2013. The problem describes "motes" of different integer sizes. One mote (Armin) is controlled by a player, the rest are passive. If Armin is of size X, it can absorb any passive mote of size $Y < X$ and grow to size $X+Y$ as a result. You are given the initial size of Armin and the sizes of passive motes. You can add a passive mote of any positive size, or you can remove any existing passive mote. Minimize the number of addition and removal operations required for Armin to be able to absorb all passive motes.

Our Picat program uses the `planner` module [7].

To come up with an effective planning solution we need to notice that there always exists an optimal solution in which Armin absorbs passive motes in order from smallest to largest (if there is a pair of motes absorbed in different order, they can be swapped without increasing the number of operations needed).

```
1  import planner.
2  final([_, []]) => true.
3  action([Armin, Others], NewState, Action, Cost) ?=>
4      Others = [Min | Rest],
5      Armin > Min,
6      NewArmin is Armin + Min,
7      Action = absorb,
8      Cost = 0,
9      NewState = [NewArmin, Rest].
10 action([Armin, Others], NewState, Action, Cost) ?=>
11     Others = [Min | _Rest],
12     Armin =< Min,
13     append(NewOthers, [_], Others),
14     Action = remove,
15     Cost = 1,
16     NewState = [Armin, NewOthers].
17 action([Armin, Others], NewState, Action, Cost) =>
18     Others = [Min | _Rest],
```

[6] Problem link: http://goo.gl/0N5zB8

```
19      Armin =< Min,
20      NewItem is Armin - 1,
21      NewOthers = [NewItem | Others],
22      Action = add,
23      Cost = 1,
24      NewState = [Armin, NewOthers].
25  do_case(Case_num, Armin, Others) =>
26      Limit = length(Others),
27      best_plan([Armin, sort(Others)], Limit, _Plan, Cost),
28      printf("Case #%w: %w\n", Case_num, Cost).
```

Listing 1.5. Picat solution for the "Osmos" problem

Solving a planning problem in Picat requires a `final` predicate and an `action` predicate. Line 2 defines the `final` predicate which has one parameter – the current state – and succeeds if the state is final. In our program a state is represented as a 2-element list: the first item is the Armin size, and the second item is a sorted list of the sizes of passive motes (`Others`). A state is final if the `Others` list is empty.

Lines 3–24 define the `action` predicate which has three clauses – one for `absorb`, `remove` and `add` actions – and has four parameters: current state, new state, action name, and action cost. Lines 3–9 define the `absorb` action which can be used if Armin is bigger than the first of the other motes at the cost of 0. Lines 10–16 define the `remove` action which removes the last (the largest) mote from `Others` at the cost of 1. The `append` predicate and the `[ | ]` syntax for getting the head and the tail of a list work exactly the same way as in Prolog. Lines 17–24 define the `add` action which adds a mote of size $Armin - 1$ to the beginning of the `Others` (so it can be absorbed by the next absorb action) at the cost of 1.

Picat's predicate for finding an optimal plan – `best_plan` – has two input parameters: the initial state and the resource limit, and two output parameters: the best plan and its cost. To find an optimal plan the system uses tabling and iterative deepening depth-first search-like algorithm. If no plan was found and the maximum resource limit was reached, the predicate fails. In this problem the resource limit for `best_plan` is the initial number of passive motes, because there is an obvious plan of this cost to remove all the motes.

This solution is a declarative specification of the problem statement which relies on just a few observations about the problem. An imperative solution would require much more insight into the problem. One could notice that in an optimal solution if a mote is removed, all motes of equal or greater sizes are also removed (if one of larger motes is absorbed, so can be the mote in question). Thus, a greedy solution is: keep absorbing passive motes from smallest to largest while absorbing the next one is possible. After this, either remove all passive motes left or keep adding motes of size one less than Armin's current size and immediately absorbing them until Armin can absorb the smallest passive mote left. Repeat until Armin absorbs the last of the given passive motes.

Table 1. Running times for small (4 minutes time limit) and large (8 minutes) inputs[7]

Problem	Technique	Small	Large
Triangle Areas	constraint programming	2.4s	0.9s
Welcome to Code Jam	dynamic programming	0.0s	0.3s
Bribe the Prisoners	dynamic programming	0.0s	4.8s
Osmos	planning	0.0s	0.1s

3 Conclusions

We have given several examples of declarative solutions for GCJ problems with Picat using constraint logic programming and tabled logic programming.

We considered using Picat's mixed integer programming module which might be useful for solving many GCJ problems [1], but currently there is no easy way to suppress log messages written to standard output by the underlying solver.

Compared to Prolog, Picat code can be more compact because of functions (function calls can be nested, so there is no need for intermediate variables), list comprehensions, and more convenient console input/output. Also, while many modern Prolog systems have loop constructs, Picat loop syntax looks much cleaner because neither global nor local variables need to be explicitly declared.

Running times of our Picat programs are several orders of magnitude smaller than the time limit imposed by GCJ rules (table 1).

We also have found that GCJ problems can be complex and large enough to exercise many different aspects of a programming language implementation: we discovered and reported two serious bugs in the version 0.8 of the Picat system while working on this paper (the bugs were promptly fixed for Picat 0.9).

References

1. Dymchenko, S., Mykhailova, M.: Declaratively solving tricky Google Code Jam problems with Prolog-based ECLiPSe CLP system (2014). http://arxiv.org/abs/1412.2304
2. Knuth, D.E., Pardo, L.T.: The Early Development of Programming Languages. Stanford University, Computer Science Department (1976)
3. Warren, D.S.: Memoing for logic programs. Commun ACM **35**(3) (1992)
4. Zhou, N.-F.: The language features and architecture of B-Prolog. Theory and Practice of Logic Programming **12**(1–2) (2012)
5. Zhou, N.-F.: Combinatorial search with Picat (2014). http://arxiv.org/abs/1405.2538
6. Zhou, N.-F., Kameya, Y., Sato, T.: Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In: 2010 22nd IEEE International Conference on Tools with Artificial Intelligence, vol. 2 (2010)
7. Zhou, N.-F., Kjellerstrand, H.: Solving several planning problems with Picat. In: Control and Decision Conference (2014 CCDC), The 26th Chinese (2014)

[7] Results were obtained on a 64-bit Linux machine with Intel Core i7-4900MQ CPU @ 2.80GHz and 16GB RAM using Picat 0.9.

Reactive Single-Page Applications with Dynamic Dataflow

Simon Fowler[1(✉)], Loïc Denuzière[2], and Adam Granicz[2]

[1] University of Edinburgh, Edinburgh, Scotland
simon.fowler@ed.ac.uk
[2] IntelliFactory, Budapest, Hungary
{loic.denuziere,granicz.adam}@intellifactory.com
http://www.intellifactory.com

Abstract. Modern web applications are heavily dynamic. Several approaches, including functional reactive programming and data binding, allow a presentation layer to automatically reflect changes in a data layer. However, many of these techniques are prone to unpredictable memory performance, do not make guarantees about node identity, or cannot easily express *dynamism* in the dataflow graph.

We identify a point in the design space for the creation of statically-typed, reactive, dynamic, single-page web applications for the WebSharper framework in the functional-first language F#. We provide an embedding abstraction to link a dynamic dataflow graph to a DOM presentation layer in order to implement dynamic single-page applications, and show how the technique can be used to support declarative animation.

Keywords: Functional programming · Reactive web applications · F#

1 Introduction

The web has grown from a collection of static, textual websites to a platform allowing complex, fully-fledged applications to run in a browser. A key advance has been the ability of page content to change, in particular as a result of changes to underlying data.

Changing the DOM via callback functions is adequate for small applications, but the inversion of control introduced by callback functions makes it difficult to maintain larger applications, and the code to update the presentation layer invariably becomes entangled with application logic. Techniques such as data binding allow mutable data to be inserted into the DOM, with the presentation layer automatically reflecting these changes. Functional reactive programming (FRP) [7] introduces *Signals* and *Behaviours*, where values can be treated as a function of time. Several successful implementations exist: React [1] provides an efficient data-binding system, and Elm [5] is a popular language designed for creating reactive web applications using FRP.

The design space, however, is vast. FRP, while having an extremely clean and expressive semantics, is prone to memory leaks when using higher-order signals.

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 58–73, 2015.
DOI:10.1007/978-3-319-19686-2_5

As a result, Elm's type system forbids higher-order signals and the creation of new signals, using signal transformers from arrowised FRP [13] to achieve dynamism. Applications written with React are not statically-typed and make few guarantees about the preservation of the *identity*, including internal state, such as focus, of DOM nodes.

WebSharper[1] is a framework allowing web applications to be written entirely in the functional-first language F# [17]. This is achieved by compiling quoted F# expressions to JavaScript, with raw DOM elements and events encapsulated using a functional interface. Designing a framework, `UI.Next`, for *reactive single-page applications* in WebSharper required us to identify a point in the design space fulfilling the following key properties.

Dynamism. It must be possible for the dataflow graph to consist of dynamic sub-graphs, where the structure of these sub-graphs may change over the course of the application's execution.

Predictable Memory Usage. Purely monadic FRP systems must sometimes retain the entire history of a value in order to use higher-order signals. The framework must not mandate such memory leaks in order to preserve the semantics of the reactive system.

Composability. It should be simple to compose elements in both the dataflow and presentation layers. Layers in the dataflow layer should compose using applicative and monadic abstractions, and it should be simple to integrate the dataflow and presentation layers.

Standard Type Systems. The system should not require any non-standard type system features in order to fulfil the above properties.

Control over Node Identity. The user should be able to explicitly specify whether DOM nodes are shared or regenerated upon changes in data.

1.1 Contributions

As a result of our design and implementation guided by the above principles, we report on the following scientific contributions.

- We describe a dynamic dataflow graph consisting of parameterised views of data sources, connected in a weak fashion by `Snap`s, a specialised extension of the `IVar` primitive [15]. This connects parameterised views of data sources in the dataflow graph, supporting asynchronous loading of variables, preventing glitches, and ensuring the graph is amenable to garbage collection (Section 3).
- We introduce a monoidal API for specifying DOM elements, provide abstractions to integrate this reactive DOM layer with the dynamic dataflow graph, and describe the implementation of this integration (Section 4).
- We demonstrate how a declarative animation API can be integrated with the DOM layer, making use of limited history-dependence, and can be backed by the dataflow graph (Section 5).

[1] http://www.websharper.com

`UI.Next` is freely available online at http://www.github.com/intellifactory/websharper.ui.next. Example applications can be found at http://intellifactory.github.io/websharper.ui.next.samples; the samples site itself is also written using `UI.Next`.

2 UI.Next by Example

`UI.Next` focuses on the creation of reactive, single-page applications. Before describing the implementation in detail, we provide an example of a calculator application, with standard and scientific modes.

We begin by defining data types for modes, a set of binary and unary operations, and a record to model the calculator. The calculator has one number in its memory in order to support binary operations, and a current operand and operation. We also define functions to execute the numerical operations.

```
type Mode = Standard | Scientific
type BinOp = Add | Sub | Mul | Divide | Exp | Mod
type UnOp = Sin | Cos | Tan | Squared
type Op = BinaryOp of BinOp | UnaryOp of UnOp
type Calculator = { Memory : float ; Operand : float ; Operation : Op }

let binOpFn = function
    | Add -> (+)     | Sub -> (-)
    | Mul -> ( * )   | Divide -> (/)
    | Exp -> ( ** ) | Mod -> (%)

let unOpFn = function
    | Squared -> fun x -> pown x 2
    | Sin -> sin     | Cos -> cos
    | Tan -> tan
```

There are two main reactive primitives in `UI.Next`: `Var`s, which can be thought of as observable mutable reference cells, and `View`s, which are read-only projections of `Var`s in the dataflow graph, and can be combined using applicative and monadic functional abstractions. In the following functions, `rvCalc` is a `Var` containing the current calculator state. `Var.Update` updates a variable based on its current value.

When a number button is pressed, the number is added to the current operand multiplied by 10 (`pushInt`). Pressing a unary operation button applies it to the current operand. When a binary operation is pressed, the number is placed into the memory, the operation is stored, and the operand is set to zero (`shiftToMem`). The user can then type another number, and pressing the equals button will apply the operation to the number in memory and current operand (`calculate`).

```
let pushInt x rvCalc =
  Var.Update rvCalc (fun c -> { c with Operand = c.Operand * 10.0 + x})
let shiftToMem op rvCalc =
  Var.Update rvCalc (fun c ->
    { c with Memory = c.Operand; Operand = 0.0; Operation = op })


let calculate rvCalc =
  Var.Update rvCalc (fun c ->
    let ans =
      match c.Operation with
        | BinaryOp op -> binOpFn op c.Memory c.Operand
        | UnaryOp op -> unOpFn op c.Operand
  { c with Memory = 0.0 ; Operand = ans ; Operation = BinaryOp Add } )
```

The next step is to create a view for the model, allowing it to be embedded into a web page. In order to do this, we create and combine elements of type `Doc`, a monoidally-composable representation of a DOM tree, which may contain both static and reactive fragments.

The "screen" of the calculator should display the current operand. This is done by mapping a serialisation function onto the current operand, converting it to a string (resulting in a type of `View<string>`), and creating a `Doc.TextView` representing a DOM text node which will update every time the `View` updates. We make use of the F# 'pipe' operator (`a |> f = f a`).

```
let numberDisplay rvCalc =
  let rviCalc = View.FromVar rvCalc
  View.Map (fun c -> string c.Operand) rviCalc |> Doc.TextView
```

We next define the "keypad" of the calculator. We define several button creation functions using the `Doc.Button` function, which takes as its arguments a caption, list of attributes, and a callback function to update the calculator state. `Div0` constructs a `Doc` representing a `<div>` tag, without attributes.

```
let calcBtn i rvCalc = Doc.Button (string i) [] (fun _ -> pushInt i rvCalc)
let cbtn rvCalc = Doc.Button "C" [] (fun _ -> Var.Set rvCalc initCalc)
let eqbtn rvCalc = Doc.Button "=" [] (fun _ -> calculate rvCalc)
let uobtn o rvCalc = Doc.Button (showOp o) [] (fun _ -> setOp o rvCalc; calculate
     rvCalc)
let bobtn o rvCalc = Doc.Button (showOp o) [] (fun _ -> shiftToMem o rvCalc)
let keypad rvCalc =
  let btn num = calcBtn num rvCalc
  Div0 [
    Div0 [btn 1.0 ; btn 2.0 ; btn 3.0 ; bobtn (BinaryOp Add) rvCalc]
    Div0 [btn 4.0 ; btn 5.0 ; btn 6.0 ; bobtn (BinaryOp Sub) rvCalc]
    Div0 [btn 7.0 ; btn 8.0 ; btn 9.0 ; bobtn (BinaryOp Mul) rvCalc]
    Div0 [btn 0.0 ; cbtn rvCalc; eqbtn rvCalc; bobtn (BinaryOp Divide) rvCalc]
  ]
```

We may then declare the operations which are present in scientific mode, and two rendering functions, `standardCalc` and `scientificCalc`, composing each set of components.

```
let scientificOps rvCalc =
  Div0 [
    bobtn (BinaryOp Exp) rvCalc ; bobtn (BinaryOp Mod) rvCalc
    uobtn (UnaryOp Sin) rvCalc  ; uobtn (UnaryOp Cos) rvCalc
    uobtn (UnaryOp Tan) rvCalc  ; uobtn (UnaryOp Squared) rvCalc
  ]
let standardCalc rvCalc = Div0 [ numberDisplay rvCalc; keypad rvCalc ]
let scientificCalc rvCalc =
  Div0 [ numberDisplay rvCalc; scientificOps rvCalc; keypad rvCalc ]
```

Finally, we create two radio buttons to switch between standard and scientific modes, which set the `rvMode` variable accordingly, and create a `View` of `rvMode`. We then map the appropriate rendering function to create a `View<Doc>`, which can be embedded using the `Doc.EmbedView: View<Doc> -> Doc` function.

```
let calcView rvCalc rvMode =
  let modeButtons =
    [Div0 [Doc.Radio [] Standard rvMode ; Doc.TextNode "Standard"]
    Div0 [Doc.Radio [] Scientific rvMode ; Doc.TextNode "Scientific"]] |> Doc.Concat
  View.FromVar rvMode
```

```
|> View.Map (fun mode ->
  let body =
    match mode with | Standard -> standardCalc | Scientific -> scientificCalc
  [body rvCalc; modeButtons] |> Doc.Concat
) |> Doc.EmbedView
```

3 Dataflow Layer

The dataflow layer exists to model data dependencies and consequently to perform change propagation. The layer is specified completely separately from the reactive DOM layer, and as such may be treated as a render-agnostic data model.

The dataflow layer consists of two primitives: reactive variables, `Var`s, and reactive views, `View`s. A `Var` is a data source, parameterised over a type: this is equivalent to a mutable reference cell with the notable exception that it may be *observed* by `View`s. A `View` represents a snapshot of a `Var`, and may be composed using applicative and monadic functional combinators.

In terms of the dataflow graph, a `Var` is a source node, and can have no incoming edges. A `View` is a node which must have at least one incoming edge. Edges in the graph are *not* direct pointers between nodes: nodes can be abstractly considered as communicating processes using a `Snap`, a novel, specialised variation of the Concurrent ML IVar primitive. As a result, the dataflow layer is amenable to garbage collection: if a `Var` or `View` becomes eligible for garbage collection, all dependent `View`s in the dataflow graph will be automatically garbage collected without the need for explicit unsubscription.

```
type View =
  static member Const : 'T -> View<'T>
  static member FromVar : Var<'T> -> View<'T>
  static member Sink : ('T -> unit) -> View<'T> -> unit
  static member Map : ('A -> 'B) -> View<'A> -> View<'B>
  static member MapAsync : ('A -> Async<'B>) -> View<'A> -> View<'B>
  static member Map2 : ('A -> 'B -> 'C) -> View<'A> -> View<'B> -> View<'C>
  static member Apply : View<'A -> 'B> -> View<'A> -> View<'B>
  static member Join : View<View<'T>> -> View<'T>
  static member Bind : ('A -> View<'B>) -> View<'A> -> View<'B>
```

`Var`s can be initialised, their values can be set, or they can be marked as finalised if their value no longer changes. `FromVar` creates a `View` which observes a `Var`, and `Const` creates a `View` which consists of a static, non-changing value. The `Sink` function acts as an *imperative observer* of the `View` – that is, the possibly side-effecting callback function of type `('T -> unit)` is executed whenever the value being observed changes. We use the `Sink` function to integrate the dataflow layer with the reactive DOM layer, which is further explained in Section 4.

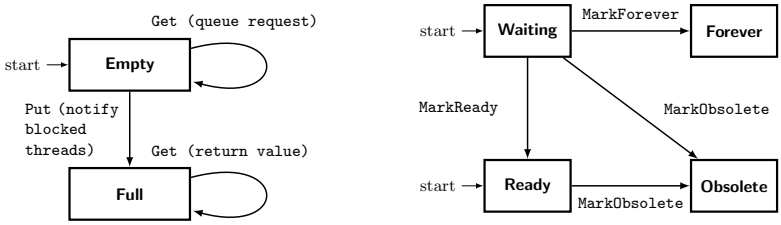
The remaining abstractions are standard combinators for applicative and monadic composition. Monadic composition allows dynamism in the dataflow graph, which is crucial for implementing dynamic single-page applications.

3.1 Implementation

In this section, we describe the implementation of the dataflow layer. A `Var` consists of a current value, a flag describing whether or not the `Var` is finalised and will not change, and a `Snap`.

```
type Var<'T> = { mutable Const : bool; mutable Current : 'T; mutable
Snap : Snap<'T> }
```

A `Snap` can be thought of as an observable and stateful snapshot of the contents of a `Var`. At its core, a `Snap` is based on the notion of an *immutable variable*, or IVar [15]. An IVar is created as an empty cell, which can be written to only once. Attempting to read from a 'full' IVar will immediately yield the value contained in the cell, whereas attempting to read from an 'empty' IVar will result in the thread blocking until such a variable becomes available. This is shown in Figure 1a.



(a) `IVar` **(b)** `Snap`

Fig. 1. State Transition Diagrams for `IVar`s and `Snap`s

A simple way of implementing change propagation using `IVar`s instead of pointers is to associate `Var`s and `View`s with an IVar `obsolete` of unit type. Dependent nodes read an initial value from the data source, attempt to perform the `Get` operation on `obsolete`, and block since `obsolete` is empty[2]. Upon changing the value, `Put` is called on `obsolete`, and all dependent nodes are notified and can fetch the latest value. Finally, `obsolete` is replaced by a fresh IVar, and the process repeats.

This model is intuitive and conveys the essence of the approach. The realisation of this technique in `UI.Next`, a `Snap`, is slightly more complex in order to support applicative and monadic combinators, perform certain optimisations, prevent certain classes of leaks, and to better support asynchronously populating a `View` from an external data source using the `MapAsync` operation. A `Snap` can be thought of as a state machine consisting of four states:

Ready: A `Snap` containing an up-to-date value, and a list of threads to notify when the value becomes obsolete.

Waiting: A `Snap` without a current value. Contains a list of threads to notify when the value becomes available, and a list of threads to notify should the `Snap` become obsolete prior to receiving a value. This is required for the

[2] We make use of the F# asynchronous programming model on the client by using a custom scheduler built into the WebSharper runtime: creating threads is done by queueing functions for execution, which are executed in a round-robin style.

implementation of the `MapAsync` combinator, and represents a `Snap` wherein a request has been made for a value, but it has not yet been received.

Forever: A `Snap` containing a value that will never change. This prevents nodes waiting for the `Snap` to become obsolete when this will never be the case.

Obsolete: A `Snap` containing obsolete information, signifying that the `View` should obtain a new snapshot.

The state transition diagram for a `Snap` is shown in Figure 1b. `Snap`s can be modified by three operations. `MarkForever` updates the `Snap` with a value which will never change, transitioning to the `Forever` state. `MarkReady` marks the `Snap` as containing a new value, notifying all waiting threads. Finally, `MarkObsolete` marks the `Snap` as obsolete. An additional operation `MarkDone` checks if the `Snap` is in the `Forever` state, and if not, transitions to the `Ready` state. `Snap`s support a variety of applicative and monadic combinators in order to implement the operations provided by `View`s: to implement `Map2` for example, a `Snap` must be created which is marked as obsolete as soon as either of the two dependent `Snap`s becomes obsolete.

`Var`s support an operation, `SetFinal`, which marks the value as finalised, preventing further writes to the variable. This prevents a class of leaks wherein a `Var` which remains static is continually observed.

`Snap`s are used to drive change propagation. When the value of a `Var` is updated, the current `Snap` is marked as obsolete and replaced by a new `Snap` in the `Ready` state.

```
type View<'T> = V of (unit -> Snap<'T>)
static member FromVar var = V (fun () -> var.Snap)
static member Set var val =
  if var.Const then () // Invalid
  else Snap.MarkObsolete var.Snap;
    var.Current <- val; var.Snap <- Snap.CreateWithValue val
```

At its core, a `View` consists of a function `observe` to return a `Snap`. The simplest `View` directly observes a single `Var`: this simply accesses the current `Snap` associated with that `Var`, updating whenever the `Snap` becomes obsolete.

At a high level, implementing `View` combinators for applicative and monadic composition involves creating a `View` with an observation function which uses the underlying `Snap` combinators. `View`s are created lazily, and results are cached for efficiency. When a `Snap` becomes obsolete, the observation functions are called to yield new `Snap`s.

```
static member Map fn (V observe) =
  View.CreateLazy (fun () -> observe () |> Snap.Map fn)
static member Map2 fn (V o1) (V o2) =
  View.CreateLazy (fun () -> let s1 = o1 (); let s2 = o2 () Snap.Map2 fn s1 s2)

static member CreateLazy observe =
 let cur = ref None
 let obs () =
   match !cur with
   | Some s when not (Snap.IsObsolete s) -> s
   | _ -> let sn = observe (); cur := Some sn; sn
 V obs
```

In order to react to lifecycle events and trigger change propagation through the dataflow graph, the `When` eliminator function is used.

```
val When : Snap<'T> -> ready: ('T -> unit) -> obsolete: (unit ->
unit) -> unit
```

The `When` function takes a `Snap` and two callbacks: `ready`, which is invoked when a value becomes available, and `obsolete`, which is invoked when the `Snap` becomes obsolete. This is implemented by matching on the state of the `Snap`, and adding the callback to the appropriate queue.

```
let Map fn sn =
  let res = Create ()
  When sn (fn >> MarkDone res sn) (fun () -> MarkObsolete res) ; res

let Map2 fn sn1 sn2 =
  let res = Create (); let v1 = ref None; let v2 = ref None
  let obs () = v1 := None; v2 := None; MarkObsolete res
  let cont () =
    match !v1, !v2 with
      | Some x, Some y -> MarkReady res (fn x y)    | _ -> ()
  When sn1 (fun x -> v1 := Some x; cont ()) obs
  When sn2 (fun y -> v2 := Some y; cont ()) obs ; res
```

The `Snap.Map` function takes a dependent `Snap` `sn` and a function `fn` to apply to the value of `sn` when it becomes available. Firstly, an empty `Snap`, `res`, is created. This is passed to the `When` eliminator along with two callbacks: the first, called when `sn` is ready, marks `res` as ready, containing the result of `fn` applied to the value of `sn`. The second, called when `sn` is obsolete, marks `res` as obsolete.

The `Snap.Map2` function applies a function to multiple arguments, which can in turn be used to implement applicative combinators. In order to do this, a `Snap` `res` and two mutable reference cells, `v1` and `v2`, are used. When either of the dependent `Snaps` `sn1` or `sn2` update, the corresponding reference cell is updated and the continuation function `cont` is called. If both of the reference cells contain values, then the continuation function marks `res` ready, containing the result of `fn` applied to `sn1` and `sn2`. If either of the dependent `Snaps` become obsolete, then `res` is marked as obsolete. This avoids glitches, which are intermediate states present during the course of change propagation, and avoids such intermediate states being observed by the reactive DOM layer.

3.2 Identity-Preserving Conversion Functions

We provide several transformation functions on reactive collections, which allow stateful conversion by using shallow memoisation: that is, where inputs are equal, previous outputs are re-used. Only one previous value for each entry in the sequence is stored, meaning that the memory usage of these functions is linear in the size of the longest sequence in the `View`. This allows *identity* to be preserved: this is particularly useful for sharing `Docs` upon updates, preventing needless DOM node regeneration and loss of internal DOM node state. This allows the transformations to have an amount of history-dependence: this is important when incorporating the notion of identity into animations, for example, as described in Section 5.2. Conversion functions are parameterised over

either two or three type parameters; `'A` and `'B` are the input and output types respectively, while `'K` is the type of an equality key. The `when 'A : equality` constraint specifies that the `'A` type must support equality testing.

```
static member Convert<'A,'B when 'A : equality>:
  ('A -> 'B) -> View<seq<'A>> -> View<seq<'B>>
static member ConvertBy<'A,'B,'K when 'K : equality>:
  ('A -> 'K) -> ('A -> 'B) -> View<seq<'A>> -> View<seq<'B>>
static member ConvertSeq<'A,'B when 'A : equality>:
  (View<'A> -> 'B) -> View<seq<'A>> -> View<seq<'B>>
static member ConvertSeqBy<'A,'B,'K when 'K : equality>:
  ('A -> 'K) -> (View<'A> -> 'B) -> View<seq<'A>> -> View<seq<'B>>
```

The `Convert` function can be thought of as converting a sequence of values, and re-using output values from the previous step should the inputs be determined to be equal. The `ConvertSeq` function is an extension of this notion, wherein the conversion function accepts a reactive view: changes to each individual item of the collection (as detected by either a machine- or user-specified notion of equality) are propagated on the item-level using this `View`.

4 Reactive DOM Layer

The Reactive DOM layer exists as a presentation layer for the dynamic dataflow graph, allowing changes in the dataflow graph to be automatically propagated to the DOM. In this section, we detail the design and implementation of the reactive DOM layer, showing how an in-memory representation of the DOM can be linked with the dataflow graph. We show how this can be used to batch updates, prevent visual glitches, and preserve the *identity* (internal state such as focus) of nodes. The simplest example of the integration of the dataflow and DOM layers is a text label which mirrors the contents of an input text box.

```
let rvText = Var.Create "" ; let inputField = Doc.Input [] rvText
let label = Doc.TextView rvText.View ; Div0 [ inputField; label ]
```

We begin by declaring a variable `rvText` of type `Var<string>`, which is a reactive variable to hold the contents of the input box. Secondly, we create an input box which is associated with `rvText`, meaning that whenever the contents of the input field changes, `rvText` will be updated accordingly. Next, we create a label using `Doc.TextView`, which we associate with a view of `rvText`. Finally, we construct a `<div>` tag using a monoidal DOM API.

Another example is that of a to-do list, where the item should be rendered with a strikethrough if the task has been completed. Arguably the most important function within the Reactive DOM layer is the `Doc.EmbedView` function:

```
static member EmbedView : View<Doc> -> Doc
```

Semantically, this allows us to embed a *reactive* DOM fragment into a larger DOM tree. This is the key to creating reactive DOM applications using the dataflow layer: by using `View.Map` to map a rendering function onto a variable, we can create a value of type `View<Doc>` to be embedded using `EmbedView`.

We begin by defining a simple type, with a reactive variable of type `Var<bool>` which is set to true if the task has been completed. An item can

be rendered by mapping a rendering function onto a `View` of this variable; note that in the code listing below, `Del0` is a notational shorthand for an HTML `<del>` element without any attributes, and `Doc.TextNode` creates a DOM text node.

```
type TodoItem = { Done : Var<bool> ; TodoText : string }
View.FromVar todo.Done |> View.Map (fun isDone ->
  if isDone then Del0 [ Doc.TextNode todo.TodoText ] else Doc.TextNode todo.TodoText)
|> Doc.EmbedView
```

4.1 Design

Reactive elements are created using the `Doc.Element` function, which takes as its arguments a tag name, a sequence of attributes, and a sequence of child elements.

```
static member Element : name: string -> seq<Attr> -> seq<Doc> -> Doc
```

Reactive attributes have type `Attr` and can be static, dynamic, or animated. Static attributes correspond to simple key-value pairs, as found in traditional static sites, whereas dynamic attributes are instead backed by a `View<string>`. We defer discussion of animation attributes to Section 5.

A key design decision is to use a *monoidal interface* for both DOM elements and attributes. All DOM elements in the reactive DOM layer are of type `Doc`. To form a monoid, `Doc`s support `Empty`, and `Append` and `Concat` functions. Reactive attributes of type `Attr` support the same interface.

4.2 Implementation

The Reactive DOM layer consists of a skeleton representation of the DOM tree in memory. Each node in this skeleton representation contains a `View` of unit type, and updates are propagated upwards through the tree. When the DOM skeleton is marked as changed, a message is sent to an update process, which applies the changes to the DOM.

DOM Skeleton Representation. The internal structure of a `Doc` is a pair of a `DocNode`, which indicates what the `Doc` represents, and a `View updates` of type `View<unit>`, which is used to notify the update process that part of the tree has changed.

```
type DocNode =
  | AppendDoc of DocNode * DocNode   | ElemDoc of DocElemNode
  | EmbedDoc of DocEmbedNode | EmptyDoc | TextDoc of DocTextNode
type DocTextNode = {Text:TextNode; mutable Dirty:bool;mutable Value:string}
type DocElemNode = {Attr:Attrs.Dyn;Children:DocNode;El:Element;ElKey:int}
type DocEmbedNode = {mutable Current:DocNode;mutable Dirty:bool}
```

Moreover, `DocNode` is a discriminated union consisting of five possible types of node. To support the monoidal interface, `AppendDoc` denotes two sibling nodes, and `EmptyDoc` denotes the absence of an element.

An `ElemNode` represents a DOM element, consisting of the attributes associated with the elements, the skeleton representation of the children of the element, the DOM element itself, and a key which is used for equality testing.

A `TextNode` represents a DOM text node, consisting of the current value, the current in-memory DOM node, and a `Dirty` flag used for DOM synchronisation. Finally, an `ElemNode` is used to represent a reactive `View` embedded into the tree. This consists of a *mutable* `DocNode` to represent the changes, and `Dirty` flag to specify that either the entire subtree, or an element within the subtree has changed.

Integration with Dataflow Layer. The main entry point to a reactive application is the `Doc.Run` function, which attaches a reactive DOM fragment of type `Doc` with a standard DOM element. The `Doc.Run` function is implemented by spawning an update process providing actor-like concurrency. Whenever a message is received by this update process, the update process firstly performs any animations that may be necessary (described further in Section 5.1), and synchronises the in-memory DOM representation with the physical DOM.

The key to the integration between the dataflow and reactive DOM layers is the `Updates View` associated with each `Doc`. The key idea for the integration of these two layers is that a notification for an update is propagated *upwards* through the tree. Once the notification propagates to the top of the `Doc` tree, the update process is notified in order to trigger any animations and synchronise the virtual and physical DOM representations.

Combining the `Views` associated with each `Doc` is done through the use of the standard `View` combinators. As an example, consider the `Doc.Append` function, which appends two `Docs` as siblings. The `AppendDoc` node requires an update either of the two contained `Docs` require an update: this can be achieved using the `Map2` combinator. `Docs.Mk` is simply a constructor for `Doc`. The `||>` operator is similar to `|>`, but takes a tupled argument, applying both arguments to the function.

```
static member Append a b =
  (a.Updates, b.Updates) ||> View.Map2 (fun () () -> ())
  |> Docs.Mk (AppendDoc (a.DocNode, b.DocNode))
```

EmbedView Implementation. `EmbedView` allows a reactive DOM segment to be embedded within the DOM tree, with any updates in this segment being reflected within the DOM.

```
static member EmbedView view =
  let node = Docs.CreateEmbedNode ()
  view |> View.Bind (fun doc -> Docs.UpdateEmbedNode node doc.DocNode; doc.Updates)
  |> View.Map ignore |> Docs.Mk (EmbedDoc node)
```

`EmbedView` works by creating a new entry in the dataflow graph, depending on the reactive DOM segment. Conceptually, this can be thought of as a `View<View<Doc>>`, which would not be permissible in many FRP systems. Here, the monadic `Bind` operation provided by the dynamic dataflow layer is crucial in allowing us to observe not only changes *within* the `Doc` subtree (using

`doc.Updates`), but changes *to* the `Doc` itself: when either change occurs, the `DocEmbedNode` is marked as dirty, and the update is propagated upwards through the tree.

Synchronisation. The synchronisation algorithm recursively checks whether any child nodes have been marked as dirty.

In the case of `EmbedNodes`, it is not only necessary to check whether the `EmbedNode` itself is dirty but also whether the current subtree value represented by the `EmbedNode` is dirty: this ensures that both global (entire subtree changes) and local (changes within the subtree) changes have been taken into account. If so, then the updates are propagated atomically to the DOM.

An important consideration of the synchronisation algorithm is the preservation of node *identity* – that is, the internal state associated with an element such as the current input in a text box, and whether the element is in focus. For this reason, when updating the children of a node, simply removing and reinserting all children of an element marked dirty is not a viable solution: instead we associate a *key* with each item, which is used for equality checking, and perform a set difference operation to calculate the nodes to be removed.

As the synchronisation process is only triggered when updates are required, the synchronisation process applies updates in a *batched* fashion, meaning that there is no visible 'cascade' of updates.

5 Declarative Animation

Animations in web applications are typically implemented as an interpolation between attribute values over time. CSS has some native animation functionality, but the approach founders when animations depend explicitly on dynamic data and cannot be determined statically. The D3 library [4] provides more powerful animation functionality, with a particular focus on data visualisation, but targets a more imperative style of programming.

`UI.Next` animations can be attached directly to elements and therefore react directly to changes within the dataflow graph. An animation is defined using the `Anim<'T>` type, where the `'T` type parameter defines the type of value to be interpolated during the animation. An `Anim<'T>` type is internally represented as a function `Compute`, mapping a normalised time to a value, and the duration of the animation.

```
type Anim<'T> = { Compute : Time -> 'T; Duration : Time }
```

An animation can be constructed using the `Anim.Simple` function, which takes as its arguments an interpolation strategy, an easing function, the duration of the animation, the delay of the animation in milliseconds, and the start and end values. Collections of animations can be described using a monoidal interface.

```
static member Anim.Simple :
  Interpolation<'T> -> Easing -> duration: Time -> delay: Time ->
      startValue: 'T -> endValue: 'T -> Anim<'T>
```

Transitions are specified using the `Trans` type.

```
static member Create : ('T -> 'T -> Anim<'T>) -> Trans<'T>
static member Trivial : unit -> Trans<'T>
static member Change : ('T -> 'T -> Anim<'T>) -> Trans<'T> -> Trans<'T>
static member Enter : ('T -> Anim<'T>) -> Trans<'T> -> Trans<'T>
static member Exit : ('T -> Anim<'T>) -> Trans<'T> -> Trans<'T>
```

A transition can either be created with the `Trivial` function, meaning that no animation occurs on changes, or with an animation. Enter and exit transitions, which occur when a node is added or removed from the DOM tree respectively, can be specified using the `Enter` and `Exit` functions.

An animation is embedded within the reactive DOM layer as an attribute through the `Attr.Animated` function:

```
static member Animated : string -> Trans<'T> -> View<'T> -> ('T -> string) -> Attr
```

This function takes the name of the attribute to animate, a transition, a view of a value upon which the animation depends (for example, an item's rank in an ordered list), and a projection function from that value to a string, in such a way that it may be embedded into the DOM.

5.1 Implementation

Animations are triggered as a result of transitions. In order to support transitions, a set of nodes from the previous update is kept at each invocation of the update process. The update process can perform the appropriate set difference operations on these two sets in order to ascertain the sets of animations which must be played as a result of nodes being added or removed.

The JavaScript `requestAnimationFrame` notifies the browser of the intent to perform an animation, and schedules a callback to be performed upon the next browser redraw cycle. The argument provided to this callback is the current timestamp: by calculating the difference between this timestamp and the timestamp at the beginning of the animation, the current point in the animation can be passed to the `Compute` function to calculate the new attribute value.

Animated attributes have an `Updates View`, which is triggered whenever an animation updates the current value of the attribute. This is linked with the remainder of the DOM synchronisation function in the `ElemNode` to which the `Attr` is attached, as the `Updates View` of the element is triggered whenever the element or any of its attributes are updated.

5.2 Example: Object Constancy

Object Constancy is a technique for allowing an object representing a particular datum to be tracked through an animation: consider the case where the underlying data does not change, but can be filtered or sorted. In such a case, the objects representing the data remaining in the visualisation should not be removed and re-added, but instead should transition to their new positions: this relies crucially on the preservation of node identity. Bostock [3] discusses an example displaying the top ten US states for a particular age bracket, sorted by population percentage. We begin by defining a data model.

```
type AgeBracket = AgeBracket of string; type State = State of string
type StateView = {
  MaxValue : double; Position : int; State : string; Total : int; Value : double }
type DataSet =
   { Brackets : AgeBracket []; Population : AgeBracket -> State -> int;
     States : State [] }
```

Here, `AgeBracket` and `State` are representations of age brackets and states respectively, and `DataSet` represents data read from an external source. The `StateView` record specifies details about how a state should be displayed based on other visible items.

```
let SimpleAnimation x y =
  Anim.Simple Interpolation.Double Easing.CubicInOut 300.0 x y
let SimpleTransition = Trans.Create SimpleAnimation let
InOutTransition = SimpleTransition
  |> Trans.Enter (fun y -> SimpleAnimation Height y)
  |> Trans.Exit (fun y -> SimpleAnimation y Height)
```

Using this, it is possible to define an animation lasting for 300ms between 2 given values. With the animation, we can then create two transitions: an unconditional transition `SimpleTransition`, and a transition `InOutTransition` which is triggered when a DOM entry is added (`Enter`) and removed (`Exit`). The `Enter` and `Exit` transitions interpolate the `y` co-ordinate of a bar between the bottom of the SVG graphic (`Height`) and a given position. The element will transition from the origin position to the desired position on, and to the origin on exit.

We now specify a rendering function taking a `View<StateView>` and returning a `Doc` to be embedded within the tree.

```
let Render (state: View<StateView>) =
  let anim name kind (proj: StateView -> double) =
    Attr.Animated name kind (View.Map proj state) string
  let x st = Width * st.Value / st.MaxValue
  let y st = Height * double st.Position / double st.Total
  let h st = Height / double st.Total - 2.
  S.G [Attr.Style "fill" "steelblue"] [
    S.Rect [
      "x" ==> "0"; anim "y" InOutTransition y; anim "width" SimpleTransition x
      anim "height" SimpleTransition h ] []
   ]
```

We specify three projection functions for the width, Y position, and height of the bar, and animated attributes for each. Finally, we create a selection box to allow the user to modify the age bracket. To implement object constancy, we use a key which uniquely identifies the data [9]. For `StateView`, this is `State`, used when embedding the current set of visible elements using `ConvertSeqBy`. The `shownData` argument is a `View` of the data to be displayed, of type `View<seq<StateView>>`.

```
S.Svg ["width" ==> string Width; "height" ==> string Height] [
  shownData |> View.ConvertSeqBy (fun s -> s.State) Render
  |> View.Map Doc.Concat |> Doc.EmbedView ]
```

6 Related Work

Functional Reactive Programming [7] provides *Behaviours* or *Signals*, representing values as a function of time. Early implementations of FRP [7] supported

higher-order signals by storing every signal value, creating a memory leak. *Arrowised* FRP [13] allows only *combinators* on primitive signals, manipulated using the Arrow abstraction [10], but avoids memory leaks as a result. The lack of first-class signals makes many GUI programming patterns difficult to implement.

Elm [5] is an FRP-based web programming language. Higher-order signals are forbidden by Elm's type system, allowing history-dependent transformations and avoiding memory leaks. In order to achieve dynamism, Elm implements arrowised FRP. Elm's history-dependence allows the elegant implementation of applications such as games, but without first-class signals and monadic composition, does not support our dynamic SPA pattern. `UI.Next` does not implement FRP signals, but retains first-class dataflow nodes and monadic composition as a result.

Krishnaswami [11] describes a language implementing FRP semantics while guaranteeing leak freedom by dividing expressions into those which may be evaluated immediately, and those which depend on future values; obsolete behaviour values are aggressively deleted. The approach relies on a specialised type system.

React [1] is a reactive DOM library which uses an automated 'diff' algorithm driven by browser redraw cycles instead of the approach we have described. We decided on a dataflow-backed system instead of a diff algorithm to retain complete control over DOM node identity. Flapjax [12] provides similar functionality to `UI.Next`, but has an entirely different approach to the dataflow graph and integrates with the DOM layer differently: signals are instead inserted manually.

The `iTask` framework [14] allows applications to be developed using *workflows*. Interconnected forms are combined using a rich set of combinators. Task-oriented programming is high-level, but is not our target in the design space; abstractions such as Flowlets [2] can handle scenarios such as dependent sequential forms.

SMLtoJS [8] also compiles an ML language (SML) to JavaScript and provides an interface to the DOM API.

7 Conclusion and Future Work

In this paper, we have presented a framework in F#, `UI.Next`, facilitating the creation of reactive applications backed by a dynamic dataflow graph. `Snap`s, an extension `IVar`s, are used as weak links within the dataflow graph to make the graph more amenable to garbage collection and prevent glitches. The DOM layer allows reactive DOM fragments to be embedded using the `EmbedView` function, and uses a monoidal interface. Finally, we have presented an interface for declarative animation which integrates directly into the reactive DOM layer as reactive attributes. We are currently investigating the use of an F# type provider [16] for reactive templating, and are working on formalising the semantics of `UI.Next`, to give a semantics to reactive abstractions such as Flowlets [2] and Piglets [6].

Acknowledgments. Fowler is supported by EPSRC grant EP/L01503X/1 (University of Edinburgh School of Informatics CDT in Pervasive Parallelism). Thanks to Anton Tayanovskyy, Adam Harries, and the anonymous reviewers for their useful comments.

References

1. React — A JavaScript Library for Building User Interfaces (2014). http://facebook.github.io/react/
2. Bjornson, J., Tayanovskyy, A., Granicz, A.: Composing reactive GUIs in F# using websharper. In: Hage, J., Morazán, M.T. (eds.) IFL. LNCS, vol. 6647, pp. 203–216. Springer, Heidelberg (2011)
3. Bostock, M.: Object Constancy (2012). http://bost.ocks.org/mike/constancy/
4. Bostock, M., Ogievetsky, V., Heer, J.: D3: Data-Driven Documents. IEEE Transactions on Visualization and Computer Graphics **17**(12), 2301–2309 (2011)
5. Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for GUIs. In: PLDI 2013, New York, NY, USA (2013)
6. Denuzière, L., Rodriguez, E., Granicz, A.: Piglets to the Rescue. In: IFL 2013, Nijmegen, The Netherlands (2013)
7. Elliott, C., Hudak, P.: Functional reactive animation. In: ICFP 1997, vol. 32, no. 8, pp. 263–273. ACM, New York (1997)
8. Elsman, M.: SMLtoJs: Hosting a Standard ML Compiler in a Web Browser. In: PLASTIC 2011, Portland, OR, USA (2011)
9. Heer, J., Bostock, M.: Declarative Language Design for Interactive Visualization. IEEE Transactions on Visualization and Computer Graphics **16**(6), 1149–1156 (2010)
10. Hughes, J.: Generalising monads to arrows. Science of Computer Programming **37**(1–3), 67–111 (2000)
11. Krishnaswami, N.R.: Higher-order functional reactive programming without spacetime leaks. In: ICFP 2013, New York, NY, USA (2013)
12. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for ajax applications. In: OOPSLA 2009, New York, NY, USA (2009)
13. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Haskell 2002, New York, NY, USA (2002)
14. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In: ICFP 2007, New York, NY, USA (2007)
15. Reppy, J.H.: Concurrent programming in ML. Cambridge University Press (2007)
16. Syme, D., Battocchi, K., Takeda, K., Malayeri, D., Fisher, J., Hu, J., Liu, T., McNamara, B., Quirk, D., Taveggia, M., Chae, W., Matsveyeu, U., Petricek, T.: Strongly-typed language support for internet-scale information sources. Technical report, Technical Report MSR-TR-2012-101, Microsoft Research (2012a)
17. Syme, D., Granicz, A., Cisternino, A.: Expert F# 3.0. APress (2012b)

CHR(Curry): Interpretation and Compilation of Constraint Handling Rules in Curry

Michael Hanus(✉)

Institut für Informatik, CAU Kiel, Kiel 24098, Germany
mh@informatik.uni-kiel.de

Abstract. Constraint Handling Rules (CHR) is a rule-based language to specify application-oriented constraint solvers. CHR requires a host language that provides the basic constraints used in a CHR program. In this paper, we argue that an integrated functional logic language like Curry is an appropriate host language for CHR since it supports a natural formulation of constraint handling rules and a seamless integration into a typed environment. As a proof of concept, we describe CHR(Curry), an integration of CHR into Curry, together with two implementations. The first is an interpreter of CHR's refined operational semantics implemented in Curry, and the second compiles CHR rules into Prolog which can be directly used in Prolog-based Curry implementations, such as PAKCS.

1 Motivation

Functional logic languages [4,15] integrate the most important features of functional and logic languages in order to provide a variety of programming concepts. They support functional concepts like higher-order functions and lazy evaluation as well as logic programming concepts like non-deterministic search and computing with partial information. This combination allows better abstractions for application programming and has also led to new design patterns [1,5] as well as better abstractions in application programs such as implementing graphical user interfaces [12] or web frameworks [17]. The declarative multi-paradigm language Curry [11,18] is a modern functional logic language with advanced concepts for application programming [2,3].

An important application area of declarative, and in particular, logic programming languages is constraint programming [19,22]. Since logic programming is a subset of functional logic programming, there exist various attempts to extend functional logic languages with constraint solving facilities (see [24] for a survey). For instance, Lux [21] describes an implementation of a solver for real arithmetic constraints for Curry, and the inclusion of finite domain constraints in the functional logic language TOY [20] is described in [9].

An alternative to using a fixed set of constraint solvers are Constraint Handling Rules (CHR) [10]. CHR is a declarative language for specifying application-oriented constraint systems. They are useful for applications that require specific constraints for which no standard solvers (like solvers for finite domain or real

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 74–89, 2015.
DOI:10.1007/978-3-319-19686-2_6

arithmetic constraints) exist. CHR defines the processing of multisets of constraints by the specification of multi-headed simplification or propagation rules. Thus, CHR is a high-level language to specify and implement constraint solvers for various application domains (see [10,27] for more detailed surveys).

Since CHR consists only of rewrite rules, CHR programs require a host language $\mathcal{H}$. On the one hand, the results of CHR computations are intended to be used in some application program, written in $\mathcal{H}$, that interacts with users, databases etc. On the other other hand, CHR is based on the existence of a set of basic constraints and data types that are used inside CHR rules. In order to make the reference to the host language $\mathcal{H}$ explicit, the notation CHR($\mathcal{H}$) is used. Most CHR systems implement CHR(Prolog) so that Prolog predicates can be used as basic constraints in CHR programs.

Example 1. The following CHR(Prolog) program [10] defines a generic less-than-or-equal relation `leq`.

```
reflexivity  @ leq(X,Y) <=> X=Y | true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X=Y.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

The first rule uses the Prolog predicate "`=`" to check the equality of the `leq` arguments, i.e., if both arguments are equal, then the CHR constraint `leq(X,Y)` can be omitted (or replaced by `true`). The second rule uses the same predicate as a constraint that unifies the arguments `X` and `Y` in order to enforce the antisymmetry property of `leq`. The detailed meaning of these rules will be explained in Section 3.

Most implementation and research efforts have been done for CHR(Prolog). Nevertheless, Prolog does not seem the most natural host language since non-Prolog features, like evaluable expressions, are sometimes used in example programs.

Example 2. The following simple CHR program, presented in [8], calculates the greatest common divisor (gcd) of two integers:

```
gcd1 @ gcd(0)           <=> true.
gcd2 @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

The intended use of this program is to put two CHR constraints `gcd(A)` and `gcd(B)` into the initial store. The second rule replaces the larger value by smaller ones (if `N` is positive) so that, after removing one CHR constraint by the first rule, the remaining CHR constraint contains the greatest common divisor.

Although the authors of [8] use the general notation of CHR(Prolog), they remark that the term `M-N` occurring in the second rule is not treated as in Prolog but it is "automatically evaluated" (as in functional programming). Since such functional notations occur also in many other examples (and they are translated in the actually implemented examples into non-declarative Prolog features), it seems that a functional logic language is a more appropriate host language than Prolog. In order to show that this idea is feasible, we propose in this paper CHR(Curry). Curry as a host language for CHR has the following advantages:

- The natural functional notation can be used in CHR rules.
- All functions defined in a Curry program as well as all predicates or constraints can be used in CHR rules.
- CHR constraints can be used in Curry programs. In particular, one can define application-oriented constraint solvers as high-level CHR rules and use them as any other predefined constraint.
- One can use high-level APIs developed in functional logic style to visualize the results of CHR computations, e.g., in graphical user interfaces [12], interactive web pages [13], or web frameworks [17].
- If CHR is embedded into a strongly typed host language, such as Curry, one gets type safety and (polymorphically) typed CHR constraints for free.

We develop CHR(Curry) as follows. In a first step, we show how CHR rules can be written in Curry without any language extension, i.e., we basically develop an eDSL (embedded domain specific language) for CHR in Curry. In a second step, we sketch two implementations of this eDSL: an interpreter oriented towards the refined operational semantics of CHR [8], and a compiler that translates CHR rules into an existing CHR(Prolog) implementation.

In the next section, we introduce some concepts of functional logic programming and the language Curry. Section 3 reviews the basic ideas of CHR. Section 4 contains our proposal to integrate CHR in Curry. Sections 5 and 6 sketches the implementations of this proposal before we conclude with a review of related work in Sections 7 and 8.

2 Basic Elements of Curry

We briefly review those elements of Curry which are necessary to understand the contents of this paper. More details can be found in recent surveys on functional logic programming [4,15] and in the language report [18].

Curry is a multi-paradigm declarative language that combines in a seamless way features from functional, logic, and concurrent programming and supports application-oriented programming (with types, modules, encapsulated search, monadic I/O [29]). The syntax of Curry is close to Haskell [23], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. Functional types are "curried," i.e., $\alpha \to \beta$ denotes the type of all functions mapping elements of type α into elements of type β, and the application of an operation f to an argument e is denoted by juxtaposition ("$f\ e$").

In addition to Haskell, Curry allows *free* (*logic*) *variables* in rules and initial expressions. Function calls with free variables are evaluated by a possibly nondeterministic instantiation of demanded arguments.

Example 3. The following Curry program defines the data type of polymorphic lists and operations to concatenate two lists and compute the last element of a list:

```
data List a  = [] | a : List a

(++) :: [a]  →  [a]  →  [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a]  →  a
last xs | _ ++ [x] =:= xs
        = x
  where x free
```

The `data` type declaration defines `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is written as `[a]` for conformity with Haskell). The (optional) type declaration ("`::`") of the operation "`++`" specifies that "`++`" takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type. Since "`++`" can be called with free variables in arguments, the equation "`_ ++ [x] =:= xs`" in the condition of `last` is solved by instantiating the anonymous free variable `_` to the list `xs` without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

The (optional) condition of a program rule is a constraint, where a *constraint* is any expression of the built-in type `Success`. Each Curry system provides at least *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms. "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2, i.e., this expression is evaluated by proving both argument constraints concurrently. Some Curry systems also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [16]. The purpose of this paper is to provide a mechanism to specify application-oriented constraint solvers on the level of Curry programs.

3 Constraint Handling Rules

In this section we review the basic ideas of the language CHR. More details about the concept and implementation of CHR can be found in the surveys [10,27] and the CHR website[1].

A CHR program describes the processing of a multiset of user-defined constraints (also called the *constraint store*) by two kinds of rules. *Simplification rules* specify the replacement of several constraints by a multiset of constraints. *Propagation rules* specify the propagation of new constraints from several existing constraints, i.e., the new constraints are added to the constraint store. In order to restrict the applicability of rules, rules can contain guards that consist of predefined (built-in) primitive constraints. Such primitive constraints can also occur in the right-hand sides of simplification or propagation rules.

[1] http://dtai.cs.kuleuven.be/CHR/

For instance, the CHR program shown in Example 1 contains two simplification rules (reflexivity, antisymmetry) and one propagation rule (transitivity). Simplification and propagation rules are denoted by "`<=>`" and "`==>`", respectively. The primitive constraints to the left of the symbol "`|`" constitute the guard of a rule. Multiple constraints are separated by commas which are interpreted as logical conjunction. The rule

```
reflexivity  @ leq(X,Y) <=> X=Y | true.
```

specifies that an occurrence of a constraint `leq(X,Y)` can be eliminated provided that `X=Y` holds, i.e., both arguments are syntactically identical. The rule

```
antisymmetry @ leq(X,Y), leq(Y,X) <=> X=Y.
```

specifies that occurrences of both `leq(X,Y)` and `leq(Y,X)` in the constraint store can be replaced by `X=Y` that enforces the syntactic identity of `X` and `Y`. Note the different rôles of the primitive constraint `X=Y` in both rules. This constraint acts in rule `reflexivity` as a condition (test) to determine the applicability of the rule, whereas in rule `antisymmetry` it enforces the equality by manipulating the constraint store. In general, the applicability of a rule is tested without modifying the constraint store (in contrast to predicates in logic programming that are applied by instantiating the actual arguments), i.e., the left-hand side and the condition must be entailed by the constraint store before the constraints in the right-hand side are added to the store. The rule

```
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

propagates a new constraint, i.e., `leq(X,Z)` is added to the constraint store if the store already contains the constraints `leq(X,Y)` and `leq(Y,Z)`. The redundancy in the constraint store caused by propagation is useful to enable the application of further simplification rules. For instance, if the constraint store contains

```
leq(X1,X2), leq(X3,X1), leq(X2,X3)
```

the application of rule `transitivity` adds the new constraint `leq(X1,X3)` so that the application of rule `antisymmetry` deletes the constraints `leq(X3,X1)` and `leq(X1,X3)` and enforces the syntactic equality between `X1` and `X3`. As a consequence, the remaining two constraints can be deleted by enforcing the equality between `X1` and `X2`.

Since the uncontrolled application of propagation rules might lead to nonterminating derivations, the operational semantics of CHR (see Section 5) defines conditions to restrict the application of such rules. Sometimes it is useful to combine a simplification and a propagation rule into one rule, called *simpagation rule*, where the left-hand side contains two parts separated by "\", as shown in Example 2:

```
gcd2 @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

The part to the left of "\" is kept like in a propagation rule and the right part is deleted like in a simplification rule. Actually, simpagation rules can also be seen as the most general form of CHR rules. This is further discussed in the following section where we present our syntactic embedding of CHR in Curry so that CHR rules become regular Curry expressions.

4 Constraint Handling Rules in Curry

As already mentioned in Section 1, instead of extending the syntax of Curry in order to deal with CHR, we want to embed CHR rules into Curry programs. For this purpose, we represent CHR rules as data objects in Curry. Remember that the most general form of a CHR rule is the simpagation rule

$$r \ @ \ \ H_1 \setminus H_2 \iff g \mid B$$

where r is a name of the rule, H_1 and H_2 are sequences of CHR (user-defined) constraints, the guard g is a sequence of built-in (primitive) constraints and B is a sequence of CHR and built-in constraints. Simplification and propagation rules are special cases of the simpagation rule with $H_1 = \varnothing$ and $H_2 = \varnothing$, respectively. Hence, it suffices to specifiy a data structure to represent simpagation rules.

In order to abstract from the set of CHR constraints used in actual programs, we assume that the type variable `chr` denotes the type of CHR constraints, which is usually an enumeration of the various CHR constraints occurring in a CHR program. Furthermore, the variables occurring in CHR rules have a distinct domain (e.g., `Int` in case of the gcd rules shown in Example 2) which we denote by the type variable `dom`. Using a single domain in CHR rules is not a restriction since this domain could also be a union type. Therefore, we can specify the structure of a CHR rule by the following data type:

```
data CHR dom chr =
  SimpaRule [chr] [chr] [PrimConstraint dom] (Goal dom chr)
```

The four arguments of `SimpaRule` correspond to the components H_1, H_2, g, and B of a simpagation rule. We do not include the name r of the rule since we will identify rules by program objects. The type `Goal` denotes sequences of user-defined and primitive constraints and is defined as follows:

```
data Goal dom chr = Goal [CHRconstr dom chr]

data CHRconstr dom chr = PrimCHR (PrimConstraint dom)
                       | UserCHR chr
```

Hence, `CHRconstr` is the union of primitive and user-defined constraints.

Finally, the type `PrimConstraint` contains the primitive (built-in) CHR constraints such as equality, disequality, etc. Moreover, one can also embed any constraint defined in a Curry program as a primitive constraint. For this purpose, we define this type as follows:

```
data PrimConstraint a =
    Eq a a                          -- equality
  | Neq a a                         -- disequality
  | Fail                            -- always unsatisfiable
  | Compare (a → a → Bool) a a      -- ordering constraint
  | Ground a                        -- ground value?
  | Nonvar a                        -- bound variable?
  | AnyPrim (() → Success)          -- user-defined primitive
```

Although constraints like `Nonvar` and `Ground` have a non-declarative flavor, they are often used in CHR rules to control the application of rules. The argument type of `AnyPrim` reflects the fact that any constraint abstraction available in Curry can be used as a primitive constraint.

Although these type definitions cover the essential structure of CHR rules, it would be tedious to use them for writing concrete rules. Therefore, we define a bunch of operations as syntactic sugar for writing CHR rules. Since some special characters (comma, vertical bar) belong to the syntax of Curry and are not allowed as operators, we can not provide the exact Prolog-oriented syntax of CHR. Nevertheless, we want to be very close to this syntax. For this purpose, we use a goal-oriented syntax to define CHR rules. For instance, to define simplification rules, we will define an operator of type

```
(<=>) :: Goal dom chr → Goal dom chr → CHR dom chr
```

where the left- and right-hand sides are goals. To construct goals in a readable manner, we define the operator "`/\`" for the conjunction of two goals:

```
(/\) :: Goal dom chr → Goal dom chr → Goal dom chr
(/\) (Goal c1) (Goal c2) = Goal (c1 ++ c2)
```

Similarly, we define `true` as the always satisfiable (empty) goal:

```
true :: Goal dom chr
true = Goal []
```

To support a nice notation for primitive constraints, we define a generic embedding of primitive constraints into goals by

```
primToGoal :: PrimConstraint dom → Goal dom chr
primToGoal pc = Goal [PrimCHR pc]
```

and introduce some operators[2] to denote the various primitive constraints:

```
fail     = primToGoal Fail
x .=.  y = primToGoal (Eq x y)
x ./=. y = primToGoal (Neq x y)
x .>=. y = primToGoal (Compare (>=) x y)
...
```

Finally, we introduce operators to write CHR rules in the usual way:

```
(<=>) :: Goal dom chr → Goal dom chr → CHR dom chr
g1 <=> g2 | null (primsOfGoal g1)
          = SimpaRule [] (uchrOfGoal g1) [] g2

(==>) :: Goal dom chr → Goal dom chr → CHR dom chr
g1 ==> g2 | null (primsOfGoal g1)
          = SimpaRule (uchrOfGoal g1) [] [] g2
```

Here we use operations `primsOfGoal` and `uchrOfGoal` that extract the list of primitive and user-defined CHR constraints from a goal. The condition expresses the

[2] We omit in this paper the definition of the operator priorities since they should be clear from the context.

fact that primitive constraints are not allowed in the left-hand sides of CHR rules.[3] To denote simpagation rules, we introduce the operator "\\":

```
(\\) :: Goal dom chr  → CHR dom chr  → CHR dom chr
g \\ (SimpaRule h1 h2 c b) | null (primsOfGoal g) && null h1
                           = SimpaRule (uchrOfGoal g) h2 c b
```

To attach a condition to a CHR rule, we define a guard operator "|>" (note that the right-hand side of the already existing rule becomes the condition of the new rule by the use of this operator):

```
(|>) :: CHR dom chr  → Goal dom chr  → CHR dom chr
(SimpaRule h1 h2 _ c) |> b | null (uchrOfGoal c)
                           = SimpaRule h1 h2 (primsOfGoal c) b
```

In order to exploit the strong type system of the host language in CHR programs, we introduce user-defined CHR constraints as a data type. For instance, the CHR program of Example 1 contains rules for a single CHR constraint `leq`. Since the arguments of `leq` are compared by equality in the reflexivity and antisymmetry rule, they can be arbitrary but have to be of the same type.[4] Thus, we define the following data type to represent this CHR constraint:

```
data LEQ a = Leq a a
```

Since user-defined CHR constraints should be embedded into CHR goals, our CHR implementation defines a generic embedding of binary constraints (actually, it defines a family of embeddings for various arities):

```
toGoal2 :: (a  → b  → chr)  → a  → b  → Goal dom chr
toGoal2 c x y = Goal [UserCHR (c x y)]
```

Hence, we define `leq` as a goal corresponding to the CHR constraint `Leq`:

```
leq = toGoal2 Leq
```

With this preparation and our CHR operators introduced above, we can write the rules of Example 1 as the following Curry program:

```
reflexivity  [x,y]   = leq x y <=> x .=. y  |> true
antisymmetry [x,y]   = leq x y /\ leq y x  <=> x .=. y
transitivity [x,y,z] = leq x y /\ leq y z  ==> leq x z
```

Apart from small syntactic differences, this is the "standard" notation for CHR rules. Note that all variables occurring in a CHR rule have to be introduced at some point. In Curry, they could be declared either as free variables or as parameters. In our eDSL for CHR, we decided to introduce these variables as parameters. The name of each CHR rule is represented by the name of the operation defining this rule. Thus, a CHR program consists of a list of operations (not a set, which is relevant for the refined operational semantics of CHR, see below)

[3] Our actual implementation yields also a sensible error message if this condition is not satisfied.

[4] In Haskell, they should have the type class context `Eq`, but the current version of Curry does not support type classes so that equality is syntactically defined on any type.

defining the various rules. As shown later, such a list is the input parameter to our implementations.

In a well typed CHR program, all rules have the same type, i.e., they operate over the same domain type and specify the semantics of user-defined constraints of the same type. For instance, the reflexivity rule (as well as all other `leq` rules) has the type:

```
reflexivity :: [a]  →  CHR a (LEQ a)
```

It should be noted that the polymorphic type system of Curry automatically yields a polymorphic type system for CHR. This is in contrast to [6] where a separate (monomoprhic) type system and type checker for CHR has been developed. The soundness of our typing of CHR rules will be an immediate consequence of our well-typed interpreter (see below).

Example 4. As a final example of this section, we show the implementation of Example 2 in our framework. First, we define the type of gcd constraints

```
data GCD = GCD Int
```

and embed them into goals by

```
gcd = toGoal1 GCD
```

Then, we can easily write the two rules:

```
gcd1 []     = gcd 0 <=> true
gcd2 [m,n] = gcd n \\ gcd m <=> m .>=. n |> gcd (m-n)
```

Thanks to our embedding into Curry, we can actually use the functional notation `(m-n)` for the argument of `gcd` in rule `gcd1` without any further transformation, in contrast to CHR(Prolog).

5 Interpretation

In order to provide a first implementation of our embedded CHR language, we implement an interpreter for CHR in Curry. Since the interpreter is written in a strongly typed language, it also ensures the type correctness of CHR rules: since it manipulates a typed constraint store, the type system of Curry (which is a Hindley-Milner like polymorphic type system [7]) ensures that the constraint store always contains type-correct constraints.

The implementation of the interpreter is oriented towards the operational semantics of CHR. The original operational semantics of CHR [10] is defined as a transition system that describes the application of the different kinds of CHR rules. Since simpagation rules are the most general kind of CHR rules, it suffices to consider such kind of rules only. A state of the transition system is a triple $\langle G, S, B\rangle$ where the goal G and the store S are multi-sets of constraints and B consists of built-in constraints. The initial state has the form $\langle G, \varnothing, true\rangle$ and is reduced according to the following transition steps ($A \uplus B$ denotes the disjoint union of the multi-sets A and B):

1. Solve: $\langle \{c\} \uplus G, S, B\rangle \longmapsto \langle G, S, c \wedge B\rangle$ if c is a built-in constraint

2. Introduce: $\langle \{c\} \uplus G, S, B \rangle \longmapsto \langle G, \{c\} \uplus S, B \rangle$ if c is not a built-in constraint
3. Apply: $\langle G, H_1 \uplus H_2 \uplus S, B \rangle \longmapsto \langle C \uplus G, H_1 \uplus S, H_1' = H_1 \wedge H_2' = H_2 \wedge B \rangle$ where $r \ @ \ H_1' \setminus H_2' \iff g \mid C$ is a renamed CHR rule and $B \rightarrow \exists \overline{x}(H_1' = H_1 \wedge H_2' = H_2 \wedge g)$ (i.e., the rule heads match and the condition is satisfied w.r.t. B)

Although these transition rules specify a superset of all possible evaluations, they are too weak to be used in practice. First of all, they do not include any mechanism to avoid trivial infinite propagations. This can be improved by adding a propagation history so that a rule is not applied again to the same literals [8]. Even with this improvement, the semantics is still a "theoretical only" semantics and not used in practice (i.e., not implemented by CHR systems). For instance, consider the gcd rules of Example 2 (which is a popular CHR example and one of the first appearing on the CHR website). With this theoretical semantics, the program is non-terminating since rule `gcd2` can always be applied to the constraints `gcd(0)` and `gcd(2)` so that the constraint `gcd(2)` is added to the goal in every application step. In practice, this is avoided by ordering rules and constraints and considering CHR constraints as procedure calls or active constraints that try to find matching partners constraints to apply a rule. For instance, the gcd solver immediately removes the constraint `gcd(0)` with the first rule `gcd1` so that the infinite loop is avoided.

A refined operational semantics covering these issues has been precisely defined in [8] by a refined set of transition rules. Due to lack of space, we do not recapitulate them here. In a declarative programming language, the transition rules can be implemented with reasonable effort. Hence, we have written a simple interpreter (approximately 50 lines of code) based on these transition rules in Curry. Since the standard evaluation mode of Curry is narrowing (i.e., unification + functional reduction), it cannot be directly used to implement CHR rules since the application of a rule requires the check for the applicability of a rule *without* instantiating free variables in a goal. Therefore, our implementation exploits the predefined operation `rewriteSome` of the library `Findall`[5] which evaluates an expression by term rewriting, i.e., without binding free variables.

The basic interface to our CHR interpreter has the following type:

```
runCHR :: [[dom] → CHR dom chr] → Goal dom chr → [chr]
```

Hence, it takes as input a list of CHR rules and a goal and returns, in case of a successful evaluation, the list of remaining user-defined constraints. For instance, the evaluation of the expression

```
runCHR [gcd1,gcd2] (gcd 16 /\ gcd 28)
```

yields the result `[GCD 4]`. In order to embed CHR constraints into Curry as predefined constraints, there is also an operation

```
solveCHR :: [[dom] → CHR dom chr] → Goal dom chr → Success
```

This solver succeeds in case of a successful evaluation and, in addition, it issues a warning if there are some remaining (suspended) constraints. Using `solveCHR`,

[5] http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/Findall.html

we can use CHR constraints as any other constraint in Curry programs, e.g., we can write CHR constraints in conditions of defined operations in order to restrict their applicability.

As already mentioned, the type system of Curry ensures that well-typed CHR rules yield well-typed CHR computations, i.e., we obtain a polymorphic CHR type system for free. In particular, we can also define CHR rules for polymorphic constraints.

Example 5. The union-find algorithm is an interesting example to demonstrate the power of CHR [26]. The algorithm maintains a collection of disjoint subsets with canonical elements (representatives) and operations `union` and `find`. Since the type of the elements is not important, the sets can be modeled as a polymorphic data type. Thus, the CHR(Prolog) program presented in [26] can be defined in a type-safe manner in CHR(Curry) as follows:

```
data UF a = Root a | Arrow a a | Make a
          | Union a a | Find a a | Link a a

root  = toGoal1 Root    (~>) = toGoal2 Arrow   make = toGoal1 Make
union = toGoal2 Union   find = toGoal2 Find    link = toGoal2 Link

makeI    [a]       = make a <=> root a
unionI   [a,b,x,y] = union a b <=> find a x /\ find b y /\ link x y
findNode [a,b,x]   = a ~> b \\ find a x <=> find b x
findRoot [a,x]     = root a \\ find a x <=> x .=. a
linkEq   [a]       = link a a <=> true
linkTo   [a,b]     = link a b /\ root a /\ root b <=> b ~> a /\ root a
```

Since the type `UF` is polymorphic, this union-find algorithm can be applied to sets of various types (e.g., sets containing integers, characters, or strings) and the type system ensures that sets of different types can not be mixed.

6 Compilation

Since our CHR interpreter is parameterized over the list of CHR rules, it is useful to develop and test CHR programs. For instance, one can evaluate CHR goals with different sets of rules or rules in various orders. However, due to the interpretive approach and purely declarative implementation without any side effects or global state, the implementation is quite inefficient compared to native CHR implementations. As an alternative, one can reuse existing CHR implementations to which we can compile our CHR(Curry) programs. For instance, there are good CHR(Prolog) implementations available for SICStus- or SWI-Prolog [25]. Since the Curry system PAKCS [16] compiles Curry programs into SICStus- or SWI-Prolog programs, it is reasonable to compile CHR(Curry) programs into CHR(Prolog) programs. For this purpose, our CHR library contains an operation

```
compileCHR :: String → [[dom] → CHR dom chr] → IO ()
```

The first argument is the name of the target Curry module into which the CHR rules, specified in the second argument, are compiled. Actually, the generated

Curry module only contains an interface to access the compiled CHR constraints from Curry programs. The generated CHR(Prolog) constraints are accessed from this module by the usual foreign function interface provided by PAKCS.

As an example, consider the CHR(Curry) program to compute the greatest common divisor (Example 4). The call "`compileCHR "GCDC" [gcd1,gcd2]`" generates the following Curry module:

```
module GCDC where

import CHRcompiled

gcd :: Int  →  Goal GCD
gcd x1 = Goal (prim_gcd $!! x1)

prim_gcd external ⟨internal code to call the CHR Prolog code⟩
```

The imported module `CHRcompiled` contains some definitions that are required to handle (typed!) CHR goals also in combination with compiled CHR programs. For instance, there is the definition

```
data Goal chr = Goal Success
```

Hence, the argument of the data constructor `Goal` is a constraint, which is reasonable since it is a container for the compiled CHR(Prolog) constraints. However, the type is parameterized by a phantom type `chr` in order to avoid a mixture of CHR constraints with incompatible types. For instance, the conjunction of constraints is defined in the module `CHRcompiled` by

```
(/\) :: Goal chr  →  Goal chr  →  Goal chr
(/\) (Goal g1) (Goal g2) = Goal (g1 & g2)
```

so that only goals over the same domain can be combined. Hence, mixing union-find constraints (Example 5) over sets of integers and sets of characters in the same goal would be rejected by Curry's type system. In order to embed the CHR(Prolog) solver as a Curry constraint, `CHRcompiled` also defines the operation

```
solveCHR :: Goal chr  →  Success
```

which solves the CHR goal and issues a warning if there are some remaining (suspended) constraints.

It should be noted that the generated operation `gcd` evaluates its argument (by the strictly evaluating application operator "`$!!`") before putting the constraint into the constraint store. This is necessary to interface the functional features of Curry with CHR. Since the CHR semantics (see Section 5) does not evaluate arguments but consider them as free Herbrand terms as in logic programming, defined functions need to be evaluated before passing the CHR constraints to the CHR solver. Hence, we can write in the application program "`gcd (6+9*4)`" which is passed as the constraint `gcd(42)` to CHR(Prolog).

The actual CHR(Prolog) program is generated by a straightforward transformation of the CHR(Curry) rules. The only interesting aspect is the interfacing between the CHR(Prolog) solver and Curry, because CHR(Curry) rules can also contain calls to operations defined in Curry programs (e.g., calls to the greater-or-equal or subtraction operations in rule `gcd2`). Since CHR(Prolog) allows the use of any Prolog predicate inside rules and Curry operations are compiled into

Prolog predicates by PAKCS, interfacing CHR(Prolog) and Curry is not difficult. For instance, rules `gcd1` and `gcd2` of Example 4 are translated into the following CHR(Prolog) rules (the code is simplified since the actual code requires additional control information for PAKCS):

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=> eq('Prelude.>='(M,N),'Prelude.True')
                         | eq(X,'Prelude.-'(M,N)), gcd(X).
```

The Prolog predicate `eq` implements the strict unification operator "`=:=`", i.e., both arguments are evaluated to normal form and unified. Thus, the original argument `(m-n)` of `gcd` in rule `gcd2` is evaluated by applying the subtraction operation defined in the standard prelude of Curry (`Prelude.-`) and `X` is bound to the result before the constraint `gcd(X)` is activated. In this way any (type-correct) operation implemented in Curry can be used in CHR rules.

Compiled CHR constraints can be solved by `solveCHR` as any other Curry constraint, e.g., in initial goals or conditions of defined operations. In contrast to the interpreter "`runCHR`", remaining (suspended) CHR constraints are not returned but it is intended that all user-defined constraints should be removed at the end. This can usually be obtained by adding rules and specific constraints to access information contained in the constraint store. For instance, to retrieve the value of the greatest common divisor that would remain in the constraint store, we replace rule `gcd1` by the following new rule (we omit here the simple extension of the data type `GCD`):

```
gcda [n,x] = gcd 0 /\ gcd n /\ gcdanswer x <=> x .=. n
```

With this rule, the constraint `gcd 0` is not simply discarded but, at the same time, the argument of the constraint `gcdanswer` is unified with the remaining value and all three constraints are discarded. If we compile the rules `[gcda,gcd2]`, we yield for the Curry goal

```
solveCHR (gcdanswer x & gcd 16 & gcd 28)  where x free
```

the answer substitution {`x=4`}.

The concrete implementation of our compiler is rather technical so that we omit a more detailed description here. The complete implementation of CHR(Curry), i.e., the eDSL operations shown in Section 4, the interpreter and the compiler, is freely available as a Curry module (`CHR`) in recent distributions of PAKCS [16]. As shown by the examples above, operations defined in Curry can be used inside CHR(Curry) rules and CHR constraints can be used in Curry programs so that we obtained a thorough embedding of CHR in Curry. In addition to the examples presented in this paper, various constraint solvers have been implemented in CHR(Curry), like Boolean constraints, finite domain constraints, prime numbers, Gaussian elimination to solve linear equalities, or computing Fibonacci numbers (as shown in [8]). The latter example also demonstrates the improved efficiency of the compilation approach: our CHR interpreter needs 1.4/9.7 seconds to compute the 50./100. Fibonacci number, whereas the compiled CHR code computes these numbers in less than 10 milliseconds (with an Intel Core i7-4790/3.60Ghz processor).

7 Related Work

Since there are a lot of publications related to CHR ([10,27] provide good surveys on different stages of the CHR development), we compare our work to some closely related work only.

HaskellCHR[6] is an implementation of CHR in Haskell. It mainly emphasizes on the implementation of the operational semantics of CHR in Haskell but does not provide a deeper embedding of CHR rules in Haskell programs, e.g., neither a specific syntactical embedding nor a type system for CHR. It has been successfully used in the Chameleon system [28] to implement advanced type systems.

HCHR [6] is a deeper embedding of CHR into Haskell. Although HCHR implements a monadic interpreter for CHR in Haskell (including an implementation of logic variables and unification), HCHR is more restricted and less flexible than our approach. Since HCHR uses a specific syntactic extension to write CHR rules, it does not use Haskell's type system for CHR. Actually, it implements a monomorphic type system for CHR and transforms rules into Haskell operations so that the Haskell type checker is used to detect type errors.

The CHR(Prolog) implementation distributed with SICStus-/SWI-Prolog [25] also supports the declaration of type annotations to CHR constraints. Although one can introduce polymorphic data structures like lists, the type annotations to CHR constraints are restricted to monomorphic types.

An early predecessor of this work [14] contained a first proposal to integrate CHR into Curry. This implementation was much more restricted than the current approach. Only goals of a predefined set of types were supported, user-defined Curry operations were not allowed inside CHR rules, and the implementation was only a compiler into untyped CHR(Prolog) so that it was not clear that type correct CHR rules do not yield type errors at run time. All these restrictions are removed in our new framework.

8 Conclusion

In this paper we presented CHR(Curry), an embedding of CHR into the functional logic host language Curry. To avoid a CHR-specific language extension of Curry, we presented an eDSL to embed CHR rules into Curry programs with a notation closely related to "standard" CHR programs. This representation has the advantage that one can use functional notation in CHR rules, and Curry's type system can be exploited to check the well-typedness of CHR rules. Since we implemented the refined operational semantics of CHR in Curry, the strong type system of Curry ensures that well-typed CHR programs do not yield ill-typed constraints at run time. Since Curry's type system supports parametric polymorphism, one can also specify polymorphic constraints, as shown in the less-or-equal or union-find solvers. Due to the thorough embedding of CHR into Curry, one can use operations defined in Curry programs inside CHR rules and

[6] http://www.comp.nus.edu.sg/~gregory/haskellchr/

one can use CHR constraints in conditions of rules defining Curry operations. Hence, one can exploit the advantage of CHR to write application-specific constraint solvers.

The use of a functional logic host language instead of a purely logic host language for CHR has various advantages. For instance, the natural functional notation can be directly applied in CHR rules. This notation is often used in examples in papers about CHR but then manually translated into a flat relational notation in case of Prolog as a host language. Since our host language Curry comes with a polymorphic type system, we obtain a polymorphic type system for CHR for free.

We presented two implementations of CHR(Curry), an interpreter implemented in Curry and a compiler to CHR(Prolog). Whereas the interpreter is useful to develop and test various constraint solvers, the compiler is necessary to use CHR(Curry) in practice. For future work, it might be interesting to explore methods to improve the efficiency of the interpreter, e.g., advanced data structures, states, monadic computations, in order to get a more efficient implementation to quickly test also larger CHR systems.

References

1. Antoy, S., Hanus, M.: Functional logic design patterns. In: Hu, Z., Rodríguez-Artalejo, M. (eds.) FLOPS 2002. LNCS, vol. 2441, pp. 67–87. Springer, Heidelberg (2002)
2. Antoy, S., Hanus, M.: Declarative programming with function patterns. In: Hill, P.M. (ed.) LOPSTR 2005. LNCS, vol. 3901, pp. 6–22. Springer, Heidelberg (2006)
3. Antoy, S., Hanus. M.: Set functions for functional logic programming. In: Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009), pp. 73–82. ACM Press (2009)
4. Antoy, S., Hanus, M.: Functional logic programming. Communications of the ACM **53**(4), 74–85 (2010)
5. Antoy, S., Hanus, M.: New functional logic design patterns. In: Kuchen, H. (ed.) WFLP 2011. LNCS, vol. 6816, pp. 19–34. Springer, Heidelberg (2011)
6. Chin, W.-N., Sulzmann, M., Wang, M.: A type-safe embedding of constraint handling rules into Haskell. Honors thesis, School of Computing, Natnioanl University of Singapore (2003)
7. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proc. 9th Annual Symposium on Principles of Programming Languages, pp. 207–212 (1982)
8. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 90–104. Springer, Heidelberg (2004)
9. Fernández, A.J., Hortalá-González, T., Sáenz-Pérez, F.: Solving combinatorial problems with a constraint functional logic language. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 320–338. Springer, Heidelberg (2002)
10. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37**(1–3), 95–138 (1998)

11. Hanus, M.: A unified computation model for functional and logic programming. In: Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris), pp. 80–93 (1997)
12. Hanus, M.: A functional logic programming approach to graphical user interfaces. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, pp. 47–62. Springer, Heidelberg (2000)
13. Hanus, M.: High-level server side web scripting in Curry. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 76–92. Springer, Heidelberg (2001)
14. Hanus, M.: Adding constraint handling rules to Curry. In: Proc. 20th Workshop on Logic Programming (WLP 2006), pp. 81–90. INFSYS Research Report 1843–06-02 (TU Wien) (2006)
15. Hanus, M.: Functional logic programming: from theory to Curry. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics. LNCS, vol. 7797, pp. 123–168. Springer, Heidelberg (2013)
16. Hanus, M., Antoy, S., Braßel, B., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R., Steiner, F.: PAKCS: The Portland Aachen Kiel Curry System (2015). http://www.informatik.uni-kiel.de/~pakcs/
17. Hanus, M., Koschnicke, S.: An ER-based framework for declarative web programming. Theory and Practice of Logic Programming **14**(3), 269–291 (2014)
18. Hanus, M. (ed.): Curry: An integrated functional logic language (vers. 0.8.3) (2012). http://www.curry-language.org
19. Jaffar, J., Lassez, J.-L.: Constraint logic programming. In: Proc. of the 14thACM Symposium on Principles of Programming Languages, Munich, pp. 111–119 (1987)
20. Fraguas, F.J.L., Hernández, J.S.: *TOY*: a multiparadigm declarative system. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
21. Lux, W.: Adding linear constraints over real numbers to Curry. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 185–200. Springer, Heidelberg (2001)
22. Marriott, K., Stuckey, P.J.: Programming with constraints. MIT Press (1998)
23. Jones, S.P. (ed.): Haskell 98 Language and Libraries-The Revised Report. Cambridge University Press (2003)
24. Rodríguez-Artalejo, M.: Functional and constraint logic programming. In: Comon, H., Marché, C., Treinen, R. (eds.) CCL 1999. LNCS, vol. 2002, pp. 202–270. Springer, Heidelberg (2001)
25. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: implementation and applications. In: First Workshop on Constraint Handling Rules: Selected Contributions, pp. 8–12 (2004)
26. Schrijvers, T., Frühwirth, T.: Optimal union-find in constraint handling rules. Theory and Practice of Logic Programming **6**(1–2), 213–224 (2006)
27. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint handling rules. Theory and Practice of Logic Programming **10**(1), 1–47 (2010)
28. Stuckey, P.J., Sulzmann, M.: A theory of overloading. ACM Transactions on Programming Languages and Systems **27**(6), 1216–1269 (2005)
29. Wadler, P.: How to declare an imperative. ACM Computing Surveys **29**(3), 240–263 (1997)

Implementation and Performance of Probabilistic Inference Pipelines

Dimitar Shterionov(✉) and Gerda Janssens

Department of Computer Science, KU Leuven,
Celestijnenlaan 200A, bus 2402, 3001 Heverlee, Belgium
{dimitar.shterionov,gerda.janssens}@cs.kuleuven.be

Abstract. In order to handle real-world problems, state-of-the-art probabilistic logic and learning frameworks, such as ProbLog, reduce the expensive inference to an efficient Weighted Model Counting. To do so ProbLog employs a sequence of transformation steps, called an *inference pipeline*. Each step in the probabilistic inference pipeline is called a *pipeline component*. The choice of the mechanism to implement a component can be crucial to the performance of the system. In this paper we describe in detail different ProbLog pipelines. Then we perform a empirical analysis to determine which components have a crucial impact on the efficiency. Our results show that the Boolean formula conversion is the crucial component in an inference pipeline. Our main contributions are the thorough analysis of ProbLog inference pipelines and the introduction of new pipelines, one of which performs very well on our benchmarks.

1 Introduction

Probabilistic Logic and Learning (PLL) software such as ProbLog [7,12] provides a machinery to derive new knowledge from uncertain data. Performing probabilistic inference or learning efficiently is a challenging task. In order to handle real-world problems state-of-the-art PLL frameworks employ knowledge compilation that reduces the initial inference or learning task into a weighted model counting (WMC) problem. Knowledge compilation converts a Boolean formula into another formula with special properties. These properties allow efficient weighted model counting on the compiled formula.

The inference mechanism of ProbLog encompasses a sequence of transformation steps in order to first compile the initial ProbLog program together with a set of query and evidence atoms and second to perform WMC on the compiled form. We call this transformation sequence an *inference pipeline* and the transformation steps – *pipeline components*. There are four components in a ProbLog pipeline – *Grounding*, *Boolean formula conversion*, *Knowledge compilation* and *Evaluation*. Each of them can be implemented with a different tool or algorithm, as long as the input/output requirements are respected. For example, ProbLog1 [7] uses knowledge compilation to ROBDDs while ProbLog2 [8] uses knowledge compilation to sd-DNNFs. In order to comply with these requirements

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 90–104, 2015.
DOI:10.1007/978-3-319-19686-2_7

it may be the case that an intermediate data formatting is needed. For example, the Boolean formula that needs to be compiled to ROBDD or sd-DNNF needs to be formatted as a BDD script or a CNF accordingly.

The performance of ProbLog pipelines depends on (i) how components are implemented, i.e., what tools or algorithms are used in order to convey the necessary transformations; and (ii) how they are linked together, i.e., how the output from one component is used as input for the next one. In this paper we investigate different implementations of each component in order to not only determine the optimal pipelines but also the components with crucial impact on the overall performance.

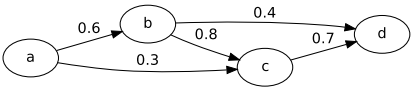
We compose 14 inference pipelines by substituting one algorithm by another for the same component when this is feasible. Then we evaluate their performance on 7 benchmark sets in order to determine the crucial component(s). These benchmarks can be considered as standard ProbLog benchmarks as they have been used in previous research to test different aspects of ProbLog inference and they cover different kinds of ProbLog programs. Our contribution is twofold – on the one hand it is the extensive analysis of ProbLog inference pipelines; and on the other the introduction of new inference pipelines, one of which performs very well on our benchmarks.

The paper is structured as follows. Section 2 gives background information on the ProbLog language as well as on weighted model counting for ProbLog inference. Section 3 presents our analysis of the different components. In Section 4 we present our experiments and discuss the results. Section 5 concludes our paper and discusses some possibilities for future research.

2 Background

2.1 The Probabilistic Logic and Learning Language ProbLog

ProbLog [7,12] is a general purpose Probabilistic Logic and Learning (PLL) programming language. It extends Prolog with probabilistic facts which encode uncertain knowledge. Probabilistic facts have the form $p_i :: f_i$, where p_i is the probability label of the fact f_i. Prolog rules define the logic consequences of the probabilistic facts. Fig. 1 shows a probabilistic graph and its encoding as a ProbLog program. The fact `0.6::e(a, b).` expresses that the edge between nodes `a` and `b` exists with probability 0.6.



a) A probabilistic graph.

```
0.6::e(a, b).  0.3::e(a, c).  0.8::e(b, c).
0.4::e(b, d).  0.7::e(c, d).
p(X, Y):- e(X, Y).
p(X, Y):- e(X, X1), p(X1, Y).
```

b) A ProbLog program.

Fig. 1. A probabilistic graph and its encoding as a ProbLog program. The `p/2` predicate defines the ("path") relation between two nodes: a path exists, if two nodes are connected by an edge or via a path to an intermediate node.

An atom which unifies with a probabilistic fact, called a *probabilistic atom* can be either true with the probability of the corresponding fact or false with (1−the probability). The choices of the truth values of all probabilistic atoms define a unique model of the ProbLog program called a *possible world*.

Let $\Omega = \{\omega_1, .., \omega_N\}$ be the set of possible worlds of a ProbLog program. Given that only probabilistic atoms have probabilities we see a single possible world ω_i as the tuple (ω_i^+, ω_i^-), where ω_i^+ is the set of probabilistic atoms in ω_i which are true and ω_i^- the set of probabilistic atoms which are false[1]. Probabilistic atoms are seen as independent random variables. A ProbLog program defines a distribution over possible worlds as given in Equation 1 where p_i denotes the probability of the atom a_i.

$$P(\omega_i) = \prod_{a_j \in \omega_i^+} p_j \prod_{a_j \in \omega_i^-} (1 - p_j) \tag{1}$$

A query q is true in a subset of the possible worlds: $\Omega^q \subseteq \Omega$. Each $\omega_i^q \in \Omega^q$ has a corresponding probability, computed by Equation 1. The (success or *marginal*) probability of q is the sum of the probabilities of all worlds in which q is true:

$$P(q) = \sum_{\omega_i \in \Omega^q} P(\omega_i) \tag{2}$$

Example 1. The query `p(a, d)` for the program in Fig. 1 is true if there is at least one path between nodes `a` and `d`. This holds in 15 out of the $2^4 = 32$ possible worlds each of them associated with a probability. Using Equation 2 gives the marginal probability $P(\texttt{p(a, d)}) = 0.54072$.

The task of computing the marginal probability of a query (i.e. the *MARG* task) is the most basic inference task of ProbLog. ProbLog can also compute the conditional probability of the query given evidence (the *COND* task).

Example 2. For the program in Fig.1, the query `p(a,d).` and evidence `e(a,b)`$= false$ ProbLog computes the conditional probability $P(\texttt{p(a,d)}|\texttt{e(a,b)} = false) = 0.21$.

2.2 Weighted Model Counting by Knowledge Compilation

Enumerating the possible worlds of a ProbLog program and computing the (marginal) probability of a query according to Equation 2 is a straightforward approach for probabilistic inference. Because the number of possible worlds grows exponentially with the increase of the number of probabilistic facts in a ProbLog program, this approach is considered impractical.

[1] The union $\omega_i^+ \cup \omega_i^-$ is the set of all possible ground probabilistic atoms of the ProbLog program with the truth value assignments specific for the possible world ω_i; the intersection $\omega_i^+ \cap \omega_i^-$ is the empty set.

In order to avoid the expensive enumeration of possible worlds the inference mechanism of ProbLog uses knowledge compilation and an efficient weighted model counting method. Model Counting is the process of determining the number of models of a formula φ. The *Weighted* Model Count (WMC) of a formula φ is the sum of the weights that are associated with each model of φ. For a given ProbLog program L with a set of possible worlds Ω the WMC of a formula φ coincides with Equation 2 when there is a bijection between the models (and their weights) of φ and the possible worlds (and their probabilities) in Ω.

The task of Model Counting (and also its specialization Weighted Model Counting) is in general a $\#P$-complete problem. Its importance in SAT and in the Statistical Relational Learning and Probabilistic Logic and Learning communities has lead to the development of efficient algorithms [5] which have found their place in ProbLog. By using knowledge compilation the actual WMC can be computed linearly to the size of the compiled (arithmetic) circuit [5, Chapter12].

3 Inference Pipeline

In order to transform a ProbLog inference task into a WMC problem ProbLog uses a sequence of transformation steps, called an *inference pipeline*. The starting point of the inference pipeline is a ProbLog program together with a (possibly empty) set of query and evidence atoms. The four main transformation steps, i.e. *components* that compose an inference pipeline are: *Grounding*, *Boolean formula conversion*, *Knowledge compilation* and *Evaluation*. The grounding generates a propositional instance of the input ProbLog program. It ignores the probabilistic information of that program, i.e. the probability label of each probabilistic fact. Second, the propositional instance is converted to a Boolean formula. The Boolean formula and the propositional instance have the same models. Third, the Boolean formula is compiled into a *negation normal form* (NNF) with certain properties which allow efficient model counting. Finally, this NNF is converted to an arithmetic circuit which is associated with the probabilities of the input program and weighted model counting is performed.

Each component can be implemented by different tools or algorithms, as long as the input/output requirements between components are respected. For example, ProbLog1 [7] uses knowledge compilation to *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [1] in order to reduce the inference task to a tractable problem. Later, [9] illustrates an approach for ProbLog inference by compilation to a *smooth, deterministic, Decomposable Negation Normal Form* (sd-DNNF) [6]. Fig. 2 gives an overview of the different approaches that can be used to implement a component and how they can be linked to form an inference pipeline. In the remaining of this section we present in detail each pipeline component and the underlying algorithms used to accomplish the necessary transformations.

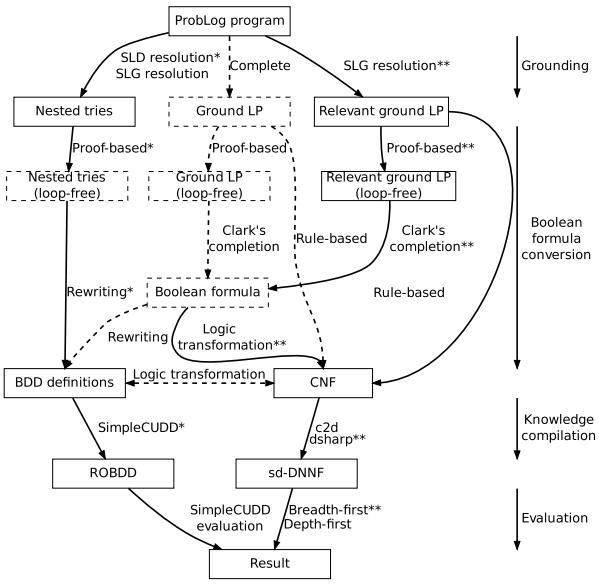


Fig. 2. ProbLog pipelines. Nodes represent Input/output formats. Each edge states a transformation and points from the output to the input. Solid edges define an existing pipeline. Default pipelines are indicated by (*) for MetaProbLog/ProbLog1 and (**) for ProbLog2. Dashed edges indicate a nonexistent pipeline. Dashed nodes indicate intermediate data formats. The input ProbLog program may contain query and evidence atoms. Vertical arrows alongside the graph indicate the components.

3.1 Grounding

A naive grounding approach is to generate the *complete* set of possible instances of the initial ProbLog program according to the values a variable can be bound to. Such a complete grounding may result in extremely big ground programs. It is more efficient with respect to the size of the grounding and the time for its generation to focus on the part of the ProbLog program which is relevant to an atom of interest. A ground ProbLog program is relevant to an atom q if it contains only relevant atoms and rules. An atom is relevant if it appears in some *proof* of q. A ground rule is relevant with respect to q if its head is a relevant atom and its body consists of relevant atoms. It is safe to confine to the ground program relevant to q because the models of the relevant ground program are the same as the models of the initial ProbLog program that entail the atom q. That is, the relevant ground program captures the distribution $P(q)$ entirely (proof of correctness can be found in [8], Theorem 1).

To determine the relevant grounding a natural mechanism is SLD resolution. Each successful SLD derivation for a query q determines one proof of q – a conjunction of ground literals. Naturally, all proofs to a query form a disjunction and therefore, can be represented as a Boolean formula in DNF. An SLD derivation may be infinite, e.g., in case of cyclic programs. In order to detect cycles (i) auxiliary code can be introduced to the input ProbLog program in order to store and compare intermediate results or (ii) SLG resolution [2] (that is, SLD with tabling) can be used instead. Adding auxiliary code as in (i) can slow down inference and is susceptible to user errors. That is why (ii), i.e. SLG resolution, is preferable for ProbLog inference.

We distinguish between two representations of the relevant grounding of a ProbLog program. ProbLog1 uses the **nested trie** structure as an intermediate representation of the collected proofs. If SLD resolution is used (that is, no tabling is invoked)[2] there is only one trie. ProbLog2 considers the **relevant ground logic program** with respect to a set of query and evidence atoms.

3.2 Boolean Formula Conversion

Logic Programs (LP) use the Closed World Assumption (CWA), which basically states that if an atom cannot be proven to be true, it is false. In contrast, First-Order logic (FOL) has different semantics: it does not rely on the CWA. Consider the (FOL) theory $\{q \leftarrow p\}$ which has three models: $\{\neg q, \neg p\}$, $\{q, \neg p\}$ and $\{q, p\}$. Its syntacticly equivalent LP (`q :- p.`) has only one model, namely $\{\neg q, \neg p\}$. In order to generate a Boolean formula from nested tries (ProbLog1, MetaProbLog) or a relevant ground LP (ProbLog2), it is required to make the transition from LP semantics to FOL semantics. When the grounding does not contain cycles it suffices to take the Clark's completion of that program [10,11]. When the grounding contains cycles it is proven that the Clark's completion does not result in an equivalent Boolean formula [11]. To handle cyclic groundings ProbLog employs one of two methods. The **proof-based** approach [14] basically removes proofs containing cycles as they do not contribute to the probability. This approach is query-directed, i.e. it considers a set of queries and traverses their proofs. The **rule-based** approach is inherited from the field of Answer Set Programming. It rewrites a rule with cycles to an equivalent rule and introduces additional variables in order to disallow cycles [11].

Once the cycles are handled, ProbLog1 rewrites the Boolean formula encoded in the nested tries as **BDD definitions**. A BDD definition [14] is a formula with a head and a body, linked with equivalence. The body of a BDD definition contains literals and/or heads of other BDD definitions combined by conjunctions or disjunctions. The logic operators are translated to arithmetic functions. A BDD script is a set of BDD definitions.

In the case of ProbLog2, the Clark's completion of the loop-free relevant ground LP is used to generate a Boolean formula. This Boolean formula is then rewritten in **CNF**. It can also be rewritten to **BDD definitions**. It is important to exploit the structure of this Boolean formula during the rewrite, otherwise the BDD script may blow up in size.

Example 3. For the ProbLog program in Fig 1 b) and the query `p(b, d)` the Boolean formula associated with the completion of the relevant ground LP is: $(p_{bd} \iff (e_{bd} \vee (e_{bc} \wedge p_{cd}))) \wedge (p_{cd} \iff e_{cd})$, where p_{xy} and e_{xy} denote `p(x, y)` and `e(x, y)` respectively. Following are its equivalent representations as a CNF and BDD definitions where $a0$ stands for an auxiliary Boolean variable:

CNF:	$(\neg p_{bd} \vee e_{bd} \vee a0) \wedge (p_{bd} \vee \neg e_{bd}) \wedge (p_{bd} \vee \neg a0) \wedge (a0 \vee \neg e_{bc} \vee \neg p_{cd}) \wedge$ $(\neg a0 \vee e_{bc}) \wedge (\neg a0 \vee p_{cd}) \wedge (p_{cd} \vee \neg e_{cd}) \wedge (\neg p_{cd} \vee e_{cd})$
BDD definitions:	p_{bd} `=` e_{bd} `+ a0` `a0 =` e_{bc} `*` p_{cd} p_{cd} `=` e_{cd}

[2] ProbLog1 allows the user to select whether to use tabling or not. ProbLog2 always uses tabling.

Example 4. A CNF can be rewritten as BDD definitions and vice-versa by a set of logical transformations. The following BDD definitions are generated from the CNF in Example 3 and are equivalent to the formula in Example 3:

BDD definitions:	a1 = p_{bd} + e_{bd} + a0	a2 = p_{bd} + ~e_{bd}	a3 = p_{bd} + ~a0	a4 = a0 + ~e_{bc} + ~p_{cd}
	a5 = a0 + e_{bd}	a6 = ~a0 + p_{cd}	a7 = p_{cd} + ~e_{cd}	a8 = ~p_{cd} + e_{cd}
	a9 = a1 * a2 * a3 * a4 * a5 * a6 * a7 * a8			

Example 3 shows how a Boolean formula that originates from Clark's completion of the relevant ground LP can easily be rewritten in CNF as well as in BDD definitions. It also shows that a CNF representation of such a formula is less succinct ([6]) than the representation as BDD definitions. If though a CNF formula is converted to BDD definitions as in Example 4 the BDD script blows up in size. For the overall performance of a pipeline it is crucial to avoid such a transformation. This phenomenon is discussed among others in [17]. In [8,9] the authors consider a ProbLog pipeline in which a CNF formula is transformed into BDD definitions as shown in Example 4, i.e. a relevant ground LP is first converted to a Boolean formula in CNF which subsequently is converted to a BDD script. Their experiments confirm that such an approach is inefficient for ProbLog inference. We do not consider further inference pipelines which include a transformation from CNF to BDD definitions. To the contrary, we introduce a *new pipeline* which transforms the relevant ground program directly into BDD definitions avoiding the blow up of the BDD script (see Table 1, pipeline $P4$).

3.3 Knowledge Compilation and Evaluation

ProbLog uses knowledge compilation to compile the Boolean formula to a negation normal form (NNF) that has the properties *determinism*, *decomposability* and *smoothness* [6]. Such an NNF is then used for efficient WMC. In ProbLog's inference pipelines two target compilation languages have been exploited so far: (i) **ROBDDs** [1] common for ProbLog1 (and MetaProbLog [13, Chapter6]) and (ii) **sd-DNNFs** [6] employed by ProbLog2.

To compile a Boolean formula to a ROBDD ProbLog implementations use **SimpleCUDD** (www.cs.kuleuven.be/~theo/tools/simplecudd.html). Compiling to sd-DNNF is done with the **c2d** [3,4] or **dsharp** [15] compilers.

After the knowledge compilation step, the compiled formula is traversed in order to compute the probabilities (i.e. the WMC) for the given query(ies) – the evaluation step. ProbLog employs two approaches to traverse sd-DNNFs: **breadth-first** and **depth-first**[3]) and one to traverse ROBDDs.

Sections 3.1 to 3.3 describe the components of the two mainstream ProbLog pipelines – ProbLog1 and ProbLog2. The subprocesses which are used in these pipelines constitute a set of interchangeable components which may form other working pipelines. Fig. 2 gives an overview of the possible ProbLog pipelines. The

[3] To invoke one of these two options in ProbLog2 one specifies either the *fileoptimized* (default) for the breadth-first implementation or *python* for the depth-first implementation as evaluation options.

link between different components depends on the compatibility of the output of a preceding subprocess with the input requirements of the next one. For example, c2d cannot compile BDD definitions but requires CNFs. Earlier it was shown that some pipelines are certain to perform worse than others: pipelines with (naive) complete grounding; pipelines in which a CNF is converted to BDD definitions (cf. Section 3.2). In addition, we prefer using SLG resolution for grounding instead of SLD resolution in order to avoid possible cycles. This leaves the 14 pipelines shown in Table 1. $P4$ and $P9..P12$ are previously unexploited pipelines for ProbLog inference.

Table 1. Pipelines used in the experiments. $X \rightarrow Y$ stands for a transformation X and the output representation Y (see Fig. 2).

	Grounding	Boolean formula conversion	Knowledge compilation	Evaluation	New Pipeline
$P0$	SLG→Rel. gr. LP	Proof-based→CNF	c2d→sd-DNNF	Breadth-first	No
$P1$	SLG→Rel. gr. LP	Proof-based→CNF	c2d→sd-DNNF	Depth-first	No
$P2$	SLG→Rel. gr. LP	Proof-based→CNF	dsharp→sd-DNNF	Breadth-first	No
$P3$	SLG→Rel. gr. LP	Proof-based→CNF	dsharp→sd-DNNF	Depth-first	No
$P4$	SLG→Rel. gr. LP	Proof-based→BDD def.	SimpleCUDD→ROBDD	SimpleCUDD	**Yes**
$P5$	SLG→Rel. gr. LP	Rule-based→CNF	c2d→sd-DNNF	Breadth-first	No
$P6$	SLG→Rel. gr. LP	Rule-based→CNF	c2d→sd-DNNF	Depth-first	No
$P7$	SLG→Rel. gr. LP	Rule-based→CNF	dsharp→sd-DNNF	Breadth-first	No
$P8$	SLG→Rel. gr. LP	Rule-based→CNF	dsharp→sd-DNNF	Depth-first	No
$P9$	SLG→Nested tries	Proof-based→CNF	c2d→sd-DNNF	Breadth-first	**Yes**
$P10$	SLG→Nested tries	Proof-based→CNF	c2d→sd-DNNF	Depth-first	**Yes**
$P11$	SLG→Nested tries	Proof-based→CNF	dsharp→sd-DNNF	Breadth-first	**Yes**
$P12$	SLG→Nested tries	Proof-based→CNF	dsharp→sd-DNNF	Depth-first	**Yes**
$P13$	SLG→Nested tries	Proof-based→BDD def.	SimpleCUDD→ROBDD	SimpleCUDD	No

4 Evaluation

4.1 Experimental Set-Up

Our experiments aim to determine the impact of the different components on the performance of the 14 pipelines. And more specifically, the components which have a crucial impact on the overall performance.

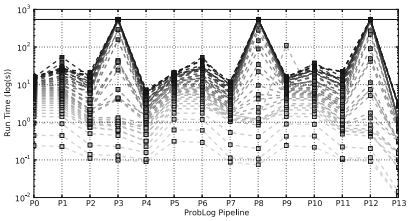
We run the 14 pipelines on 7 benchmark sets with in total 319 benchmark programs: “Alzheimer” [7], “Balls” [20], “Dictionary” [18], “Grid” [8], “Les Miserables” [18], “Smokers” [16], “WebKB” [9]. The programs from the “Alzheimer”, “Dictionary”, “Les Miserables” and “WebKB” are built from real-world data; the rest are based on artificial data.

The benchmark programs we use encode different directed probabilistic graphs. The graphs corresponding to the “Grid” benchmarks are acyclic with a hierarchical structure and maximum in/out degree of 3. The rest are cyclic; the ones in the “Les Miserables” and the “Dictionary” are sparse graphs (with density < 0.0012 and < 0.0002 respectively). Probabilistic graphs are encoded as shown in Fig. 1. The queries to these programs ask for the probability a path exists between two nodes.

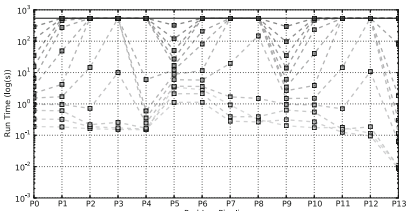
A program from the "Smokers" or "WebKB" benchmark sets contains multiple queries. The rest contain one query. The variety of these benchmarks ensures a close to realistic estimate of the general performance of ProbLog pipelines. The programs from the "Balls" benchmark set use annotated disjunctions [21] to encode random events with multiple outcomes. They are acyclic.

Our benchmarks have been used previously to evaluate different aspects of ProbLog implementations. The benchmarks from the "Alzheimer" set were used to motivate the development and test the performance of the first ProbLog system. The "Smokers" and "WebKB" benchmark sets are used for testing ProbLog2, i.e. different loop-breaking and knowledge compilation approaches. Also, the "Grid" benchmark set was developed in the context of ProbLog2 and to compare the knowledge compilation to sd-DNNFs with knowledge compilation to ROBDDs. The "Balls" benchmark set is used to test the performance of a new encoding of Annotated Disjunctions for ProbLog programs (mainly affecting the grounding). That is why we believe our experiments will allow to clearly determine the crucial components in the inference pipeline.

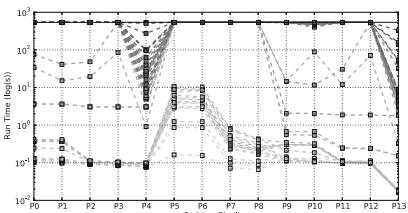
In our experiments, we measure the run times of each component while performing the MARG or the COND task for the given query(ies) and evidence. Because the sd-DNNF compilers are non-deterministic [3, 15], i.e. for the same CNF the compiled sd-DNNFs may differ, we run all tests 5 times and report the



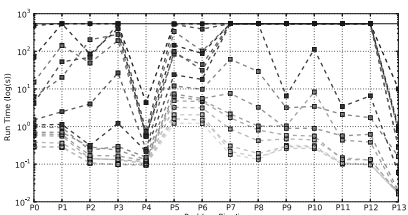
a) "Balls" benchmark set.



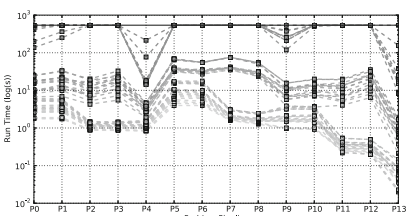
b) "Grid" benchmark set.



c) "Les Miserables" benchmark set.



d) "Smokers" benchmark set.



e) "WebKB" benchmark set.

Fig. 3. Run times for ProbLog pipelines performing MARG inference

average run time. Previous tests with these compilers within ProbLog have shown that the average time for 5 runs gives a realistic estimate on the performance. We set a time-out of 540 seconds for each run.

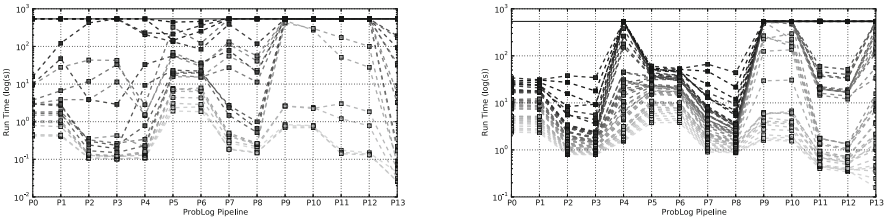
Section 4.2 presents our experimental results. A discussion follows in Section 4.3. Detailed description of our benchmarks, complete results and color diagrams can be found in [19]. Enlarged and color version of the diagrams in Fig. 3 and Fig. 4 are available in http://people.cs.kuleuven.be/~dimitar.shterionov/pipeline_diagrams.pdf. Our benchmarks can be found at http://people.cs.kuleuven.be/~dimitar.shterionov/benchmarks_pipelines.zip. In the future we would like to extend this set with new problems in order to improve generality of our conclusions.

4.2 Results

We present the total run time (the sum of the grounding, Boolean formula conversion, knowledge compilation and evaluation times) of each pipeline for a benchmark program executing MARG or COND inference. The reason to focus only on the total run time is that any change in the performance of two pipelines which share all but one component will be due to the different component. Whether the algorithm that implements the component, the compatibility with the input data or the output have an affect on the overall performance is not of importance. Rather, we are interested in how the different components' implementations influence the pipeline as a whole. To get an idea of the impact of individual components we compare the result for pipelines which differ by one component. For example, comparing pipelines $P0 - P8$ to pipelines $P9 - P13$ will determine the effect of the two different grounding approaches. Fig. 3 shows the total run time for performing MARG inference on the "Balls", "Grid", "Les Miserables", "Smokers" and "WebKB" benchmark sets. The results from the "Les Miserables" benchmarks are similar to the "Alzheimer" and the "Dictionary"; although the results from the "Smokers" benchmarks are similar to the "WebKB" we show both diagrams so that later they can be compared to the results from performing COND inference shown in Fig. 4.

In each figure a horizontal line is associated with one benchmark program and shows the total run time (thus the lower the better) of each pipeline (x-axis) executing the MARG or the COND task on that program. We use a logarithmic scale for the time axis (the y-axis). We present the lines in different shades of gray relative to the size of the dependency graph representing the program. The black line parallel to the x-axis indicates the 540^{th} second, that is, the time-out.

We also give the number of timeouts that occurred for each pipeline performing MARG and COND inference in Table 2 and Table 3 respectively. They show the total number of timeouts and the relative number of timeouts with respect to the total number of programs in a benchmark set for which at least one pipeline terminated successfully. For example, $P4$ times out for a total of 11 benchmarks when executing COND inference (see Table 3); 2 of the programs that time out are from the "Smokers" set and 9 from the "WebKB" set; in total 20 programs of the "Smokers" and 48 of the "WebKB" benchmark sets have been successfully executed; we compute the relative number of timeouts as $2/20 + 9/48 = 0.2875$.



a) "Smokers" benchmark set. b) "WebKB" benchmark set.

Fig. 4. Run times for ProbLog pipelines performing COND inference

Table 2. Number of benchmark programs for which MARG inference times out

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	46	53	62	84	8	144	145	158	177	48	47	68	89	14
Total (relative):	3.14	3.48	4.35	5.34	0.72	7.76	7.94	8.8	9.28	3.95	4.12	5.04	5.71	1.64

Table 3. Number of benchmark programs for which COND inference times out

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	3	3	3	4	11	0	0	4	4	42	42	27	27	21
Total (relative):	0.15	0.15	0.15	0.2	0.29	0.0	0.0	0.2	0.2	1.25	1.25	0.94	0.94	0.55

4.3 Discussion

We discuss the results from our experiments with the MARG task separately from the COND task. This is because computing the conditional probabilities in MetaProbLog (whose components we use to build other pipelines) differs from how conditional probabilities are computed in ProbLog2. The difference is due to the way evidence is processed.

MARG Inference. *Grounding* Comparing pipelines $P0, .., P4$ to $P9, .., P13$ in Fig. 3 shows that grounding to relevant ground LP and grounding to nested tries have similar impacts on the performance. The default MetaProbLog pipeline, $P13$ and pipeline $P4$ differ on the grounding representation. $P13$ appears to be faster than the rest in almost all of the cases. The timeouts in Table 2 though show that pipelines which use the relevant ground LP representation can solve (relatively) more problems than the ones using the nested tries. In particular, we notice that $P4$ outperforms $P13$. The effect of the one grounding representation compared to the other is though small therefore we can state that the choice of grounding representation is not crucial for the total inference performance.

Boolean Formula Conversion. When comparing pipelines $P0, .., P3$, to $P5, .., P8$ in Fig. 3 we observe that the Boolean formula conversion has a strong impact on the performance. By itself the time for conversion is not significant but it is the output Boolean formula that strongly influences the next components in the inference pipeline – knowledge compilation and evaluation. Knowledge compilation is computationally the most expensive task. The proof-based approach

generates Boolean formulae which are easier to compile, i.e. the compilation time is lower than for the rule-based approach [19]. The time out results in Table 2 show that pipelines using the proof-based conversion time out 42% to 59%[4] less than pipelines using the rule-based approach.

For the effectiveness of the conversion of great importance is the presence of cycles in the grounding. We notice (Fig. 3 a) and b)) that pipelines using the rule-based conversion handle the acyclic graphs from the "Balls" and the "Grid" benchmark sets equally well or even better than some of the pipelines using the proof-based conversion. This is because the conversion does not need to handle any cycles and the rule-based conversion which simply traverses the ground program is not only faster (see Fig. 3 a) and Fig. 3 b)) but also generates easy-to-compile Boolean formulae.

These results show that the Boolean formula conversion is crucial for the inference pipeline.

Knowledge Compilation and Evaluation Knowledge compilation has the highest impact on the inference run time. Generally, knowledge compilation to ROBDDs is preferable for MARG inference (compare $P4$ and $P13$ to the rest in Fig. 3).

In the case of knowledge compilation to sd-DNNFs a pipeline which uses c2d shows better scalability compared to one with dsharp but is slower for the less complex problems. Furthermore, the breadth-first evaluation approach is in general preferable to the depth-first approach (compare $P0$ to $P1$ or $P11$ to $P12$ in Fig. 3 c)), although for the "Balls" benchmarks this evaluation approach performs poorly (see $P3$, $P8$ and $P12$ in Fig. 3 a)). The reason is the structure of the graph associated with the relevant ground LP – low out degree, i.e. 9, long paths from the root to the nodes.

COND Inference. The conditional probability of a query q given evidence $E = e$ is computed as the ratio $P(q|E = e) = \frac{P(q \wedge E=e)}{P(E=e)}$. First both the nominator and denominator need to be computed separately. Then their division gives the final result. MetaProbLog and ProbLog2 use different approaches when it comes to computing the conditional probabilities. In particular, there are differences regarding the grounding to nested tries and compiling to ROBDDs compared to grounding to a relevant ground LP and knowledge compilation to s-DDNNFs.

Grounding We notice from Fig. 4 a) and b) and Table 3 that grounding to nested tries has a negative effect on the overall performance as compared to grounding to a relevant ground LP. The former approach is: (i) for a query q and evidence $E = e$ a new query $q^{E=e}$ (i.e., $q \wedge E = e$) is created; (ii) $q^{E=e}$ and the atoms in E are proven in order to determine the relevant grounding (stored as nested tries). In the latter case, a query q and the atoms in E are used separately and not in a conjunction to determine the relevant ground LP. Although the two approaches result in very similar groundings, the evidence atoms and their predetermined values make a difference for the performance of the next components.

[4] We use the relative number of timeouts rather than the total number of timeouts in order to determine a more general interval.

Boolean Formula Conversion The Boolean formula is built by using either the proof-based or the rule-based method. In the case of pipelines $P0$ to $P9$ the Boolean formula (either represented as a CNF or as a BDD script) is augmented with clauses to state the truth values for the evidence atoms. They often help the knowledge compilation as they may prune parts of the compiled circuit. The positive effect is obvious when comparing pipelines $P0$ to $P4$ with $P9$ to $P13$ in Fig. 4 but also from Table 3.

Knowledge Compilation and Evaluation The additional clauses for the evidence added to the Boolean formula improve the performance for pipelines $P0$ to $P3$ and $P5$ to $P9$ as compared to executing the MARG task. The two pipelines using ROBDDs ($P4$ and $P13$) do not perform well. A reason for the decreased performance of these pipelines is that for multiple queries (including evidence) it is required to build and evaluate a forest of ROBDDs. In order to compute the conditional probability of a query q given evidence $E = e$ a ROBDD for the conjunction $q \wedge E = e$ is added to the ROBDD forest even when the conjunction is false ($P(q \wedge E = e) = 0.0$ therefore $P(q|E = e) = 0.0$), thus performing unnecessary operations. Indeed, this slow down is observed for the "WebKB" benchmark programs where a lot of the queries are false given that the evidence is true. Fig. 4 shows that the ROBDD-based pipelines ($P4$ and $P13$) do not scale as well as in the case of MARG inference. Which is also confirmed by Table 3.

5 Conclusions and Future Work

In this paper we presented a detailed description of the inference pipelines of ProbLog and analyzed their performance on 7 benchmark sets. Our analysis shows that the Boolean formula conversion has a crucial impact on the performance of the inference pipeline for both MARG and COND tasks. We showed that in most of the cases pipelines which use a *proof-based conversion*, *knowledge compilation to sd-DNNF with c2d* and the *breadth-first evaluation* approach and pipelines which use *proof-based conversion* and *compilation to ROBDDs* perform better than the rest. $P4$ and $P13$ are the most efficient pipelines for our benchmarks on performing MARG inference. $P13$ is the default pipeline of MetaProbLog. $P4$ is one of the new pipelines we introduce with this paper (combining ProbLog2 with ROBDDs).

We also showed that for COND inference it is crucial how the evidence is handled. Pipelines which use *compilation to sd-DNNF* and *breadth-first evaluation* outperform the rest. The most efficient pipeline for computing the COND task is $P0$. We also determined that this difference is due to how evidence is handled.

Our analysis determines two main directions for future research: (i) to improve the Boolean formula conversion component and (ii) to investigate how to improve ROBDDs with respect to computing conditional probabilities. Furthermore, pipeline $P4$ which combines the grounding of ProbLog2 with the knowledge compilation and evaluation of MetaProbLog via a direct conversion of the (loop-free)

relevant ground LP to BDD definitions shows very promising results. To determine its actual place among the different ProbLog implementations we plan to further evaluate its performance on all inference and learning tasks supported by ProbLog.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
2. Chen, W., Swift, T., Warren, D.S.: Efficient top-down computation of queries under the well-founded semantics. J. Log. Program. **24**(3), 161–199 (1995)
3. Darwiche, A.: A compiler for deterministic, decomposable negation normal form. In: Dechter, R., Sutton, R.S. (eds). AAAI/IAAI, pp. 627–634. AAAI Press/MIT Press (2002)
4. Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: Proceedings of the 16th European Conference on Artificial Intelligence, pp. 328–332 (2004)
5. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009) (chapter 12)
6. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research **17**, 229–264 (2002)
7. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: a probabilistic prolog and its application in link discovery. In Proceedings of the 20th International Joint Conference on Artificial Intelligence, pp. 2468–2473. AAAI Press (2007)
8. Fierens, D., Van Den Broek, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., de Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory and Practice of Logic Programming, Special Issue on Probability, Logic and Learning 15(3), 358–401 (2015)
9. Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., De Raedt, L.: Inference in probabilistic logic programs using weighted CNF's. In: Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence, pp. 211–220 (2011)
10. Gutmann, B., Thon, I., De Raedt, L.: Learning the parameters of probabilistic logic programs from interpretations. In: Gunopulos, D., Hofmann, T., Malerba, D., Vazirgiannis, M. (eds.) ECML PKDD 2011, Part I. LNCS, vol. 6911, pp. 581–596. Springer, Heidelberg (2011)
11. Janhunen, T.: Representing normal programs with clauses. In: Proc. of the 16th European Conference on Artificial Intelligence, pp. 358–362. IOS Press (2004)
12. Kimmig, A., Demoen, B., De Raedt, L., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming **11**, 235–262 (2011)
13. Mantadelis, T.: Efficient Algorithms for Prolog Based Probabilistic Logic Programming. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, November 2012. Janssens, Gerda (supervisor)
14. Mantadelis, T., Janssens, G.: Dedicated tabling for a probabilistic setting. In: Hermenegildo, M.V., Schaub, T. (eds) ICLP (Technical Communications), vol. 7 of LIPIcs, pp. 124–133. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
15. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: fast d-DNNF compilation with sharpSAT. In: Kosseim, L., Inkpen, D. (eds.) Canadian AI 2012. LNCS, vol. 7310, pp. 356–361. Springer, Heidelberg (2012)

16. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S.H. (eds.) Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 1–27. Springer, Heidelberg (2008)
17. Rauzy, A., Châtelet, E., Dutuit, Y., Bérenguer, C.: A practical comparison of methods to assess sum-of-products. Rel. Eng. & Sys. Safety **79**(1), 33–42 (2003)
18. Shterionov, D., Janssens, G.: Data acquisition and modeling for learning and reasoning in probabilistic logic environment. In: Antunes, L., Pinto, H.S., Prada, R., Trigo, P. (eds) Proceedings of the 15th Portuguese Conference on Artificial Intelligence, pp. 298–312 (2011)
19. Shterionov, D., Janssens, G.: Crucial components in probabilistic inference pipelines: Data and results. Technical report, KU Leuven, 2014. Ref. number CW679. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW679.pdf
20. Shterionov, D., Renkens, J., Vlasselaer, J., Kimmig, A., Meert, W., Janssens, G.: The most probable explanation for probabilistic logic programs with annotated disjunctions. In: Proceedings of the 24th International Conference on Inductive Logic Programming
21. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 431–445. Springer, Heidelberg (2004)

A Haskell Implementation of a Rule-Based Program Transformation for C Programs

Salvador Tamarit[1(✉)], Guillermo Vigueras[2], Manuel Carro[1,2], and Julio Mariño[1]

[1] Universidad Politécnica de Madrid, Madrid, Spain
{salvador.tamarit,julio.marino}@upm.es
[2] IMDEA Software Institute, Madrid, Spain
{guillermo.vigueras,manuel.carro}@imdea.org

Abstract. Obtaining good performance when programming heterogeneous computing platforms poses significant challenges for the programmer. We present a program transformation environment, implemented in Haskell, where architecture-agnostic scientific C code is transformed into a functionally equivalent one better suited for a given platform. The transformation rules are formalized in a domain-specific language (STML) that takes care of the syntactic and semantic conditions required to apply a given transformation. STML rules are compiled into Haskell function definitions that operate at AST level. Program properties, to be matched with rule conditions, can be automatically inferred or, alternatively, stated as annotations in the source code. Early experimental results are described.

Keywords: High-performance computing · Scientific computing · Heterogeneous platforms · Rule-based program transformation · Domain-specific language · Haskell

1 Introduction

There is currently a strong trend in high-performance computing towards the integration of various types of computing elements: vector processors, GPUs being used for non-graphical purposes, FPGA modules, etc. interconnected in the same architecture. Each of these components is specially suited for some class of computations, which makes the resulting platform able to excel in performance by mapping computations to the unit best able to execute them and is proving to be a cost-effective alternative to more traditional supercomputing architectures [4]. However, this specialization comes at the price of additional hardware and, notably, software complexity. Developers must take care of very

Work partially funded by EU FP7-ICT-2013.3.4 project 610686 *POLCA*, Comunidad de Madrid project S2013/ICE-2731 *N-Greens Software* and MINECO Projects TIN2012-39391-C04-03 and TIN2012-39391-C04-04 (*StrongSoft)* and TIN2013-44742-C4-1-R (*CAVI-ROSE*).

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 105–114, 2015.
DOI:10.1007/978-3-319-19686-2_8

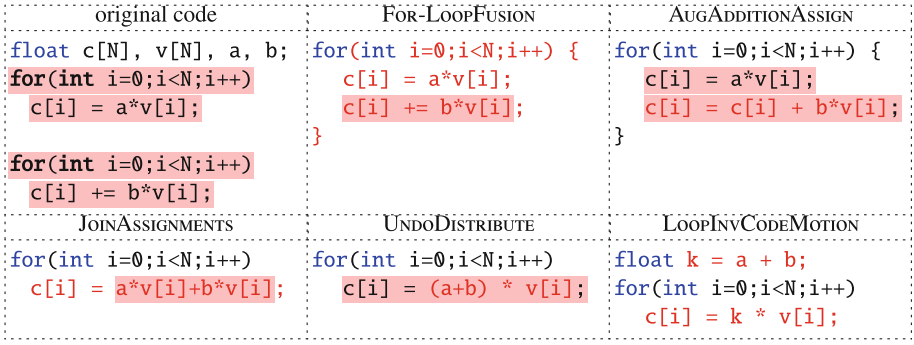


Fig. 1. A sequence of transformations of a C code that computes $\mathbf{c} = a\mathbf{v} + b\mathbf{v}$

different features to make the most of the underlying computing infrastructure. Thus, programming these systems is restricted to a few experts, which hinders its widespread adoption, increases the likelihood of bugs and greatly limits portability.

Defining programming models that ease the task of efficiently programming heterogeneous systems is the goal of the ongoing European research project POLCA.[1] The project specifically targets scientific programming on heterogeneous platforms, due to the performance attained by certain hardware components for some classes of computations – e.g., GPUs and linear algebra – and to the energy savings achieved by heterogeneous computing in scientific applications characterized by high energy consumption [4,7]. Additionally, most scientific applications rely on a large base of existing algorithms that must be ported to the new architectures in a way that gets the most out of their computational strengths, while avoiding pitfalls and bottlenecks, and preserving the meaning of the original code. Porting is carried out by transforming or replacing certain fragments of code to improve their performance in a given architecture while preserving their meaning. Unfortunately, (legacy) code often does not spell its meaning or the programmer's intentions clearly, although scientific code usually follows patterns rooted in its mathematical origin.

Our proposal is to develop a framework for semantic-based program transformation of scientific code where the validity of a given transformation is guided by high-level annotations expressing the mathematical foundation of the source code. Fig. 1 shows a sample code transformation sequence, containing the original fragment of C code along with the result of applying *loop-fusion*, reorganizing assignments, algebraic rewriting based on *distributivity* and moving *invariant* expressions out of a loop body. Some of these transformations are currently done by existing compilers. However, they are performed internally, and we need them to be applied at the source code level, since they may enable further source code-level transformations.

[1] *Programming Large Scale Heterogeneous Infrastructures*, http://polca-project.eu.

Due to space limitations, we are not showing the code annotations required to associate algebraic properties with variables and operators, etc.[2] – in this paper we will focus on the design of the tool implementing them.

The decision of whether to apply a given transformation depends on many factors. First, it is necessary to ensure that applying a rule at a certain point is sound. Several sources of information can be used here, from static analyzers (e.g., to extract data dependency and type information) to inline code annotations provided by the programmer. Second, whether the transformation may improve efficiency, which is far from trivial: *Cost models* for different target architectures are needed, as the transformation process will eventually be guided by estimations of the final performance (which may bring problems such as local optima). Finally, the transformed code may contain new derived annotations that can affect subsequent steps.

Despite the broad range of compilation and refactoring tools available [1, 8, 10], no existing tool fitted the needs of the project, so we decided to implement our own transformation framework, including a domain specific language for the definition of semantically sound code transformation rules (STML), and a transformation engine working at AST level (http://goo.gl/yuOFiE). Declarative languages are used in different ways in this project. First, the rewriting engine itself is implemented in Haskell; second, STML rules have a declarative flavor as they are rewriting rules whose application should not change the semantics of the program being rewritten; and last, the rules themselves are translated into Haskell code. Rules are written using a C-like syntax, which makes it easy for C programmers to understand their meaning and to define them, while the rules can transparently access core functionality provided by the Haskell rewriting engine, and be accessed by it.

The engine selects rules and blocks of code where these rules can be safely applied. There may be several possibilities, and the engine is able to return all of them in a list. In the final tool we plan to use heuristics to select the most promising transformation chain (Section 4) and to have available an interactive mode which can interplay with the guided search when it is not possible to automatically determine whether some rule can / should be applied or when the programmer so desires it. At the moment, only the interactive mode is implemented.[3]

2 Tool Description

The main two functionalities of our tool are: 1) to parse transformation rules written in our domain-specific language (*Semantic Transformation Meta-Language*, STML), and to translate them into Haskell; and 2) to perform source-to-source C code transformation based on these rules, possibly making use of

[2] The full example code can be found at http://goo.gl/LWRNOy.

[3] As an temporary step, useful for validation, random selection of rules and locations up to a certain number of transformations, is also available.

information provided in code annotations (*pragmas*). Occasionally, new pragmas can be injected in the transformed code.

The transformation tool is written in Haskell. The `Language.C` library [5] is used to parse the input C code and build its *abstract syntax tree* (AST), which is then manipulated using the *Scrap Your Boilerplate* (SYB) [6] Haskell library: functions like `everything` and `everywhere` allow us to easily extract information from the AST or modify it with a generic traversal of the whole structure. Both libraries are used to perform the transformation, but also for the translation of STML rules into Haskell: the STML rules themselves are expressed in a subset of C and are parsed using the same `Language.C` library. The tool is composed of four main modules:

- `Main.hs`: Implements the tool's workflow: calls the parser on the input C code which builds the AST, links the pragmas to the AST, executes the transformation sequence (interactive or automatically) and outputs the transformed code.
- `PragmaPolcaLib.hs`: Reads pragmas and links them to their corresponding AST. Restore / injects pragmas in transformed code.
- `Rul2Has.hs`: Translates STML rules (stored in an external file) into Haskell functions which actually perform the AST manipulation. Reads and loads STML rules as an AST and generates the corresponding Haskell code in a `Rules.hs` file.
- `RulesLib.hs`: Supports `Rules.hs` to identify STML rule applicability (matching, preconditions, etc.) and execution (AST traversal and mutation).

2.1 The Rule Language and Its Translation

The rule language used by our tool is inspired by CML [3], which is in turn an evolution of CTT [2]. We named it *Semantic Transformation Meta-Language* (STML), to highlight the use of information beyond the syntax of the language to transform (inferred or provided in code annotations). STML is syntactically simpler than CML and closer to C, but it features additional functionality, such as richer conditions or the ability to express only once antecedents common to several transformation rules.

```
rule_name {
   pattern: {...}
   condition: {...}
   generate: {...}
   assert:  {...}
}
```

Fig. 2. Rule template

Fig. 2 shows a rule template: whenever a piece of code matching the code in the `pattern` section is found which meets a series of conditions stated in the `condition` section, the matched code is replaced by the code in the `generate` section. The symbols in `pattern` are meta-variables which are substituted for the actual symbols in the code before performing the translation. The `conditions` can refer to both syntactic and semantic properties. The generated code can have additional (semantic) properties which can be explicitly stated in the `assert` section to make the application of other rules possible.

Table 1. Basic functions for the `condition` section of rules

Function	Description
`is_identity(`E_{op}`,E)`	`E` is an identity for E_{op}
`no_writes(`E_v`,(S|[S]|E))`	E_v is not written in `(S|[S]|E)`
`no_reads(`E_v`,(S|[S]|E))`	E_v is not read in `(S|[S]|E)`
`no_rw(`E_v`,(S|[S]|E))`	E_v is neither read nor written in `(S|[S]|E)`
`pure((S|[S]|E))`	There is not any assignment in `(S|[S]|E)`
`is_const(E)`	There is not any variable inside `E`
`is_block(S)`	`S` is a block of statements
`not(`E_{cond}`)`	E_{cond} is false

Table 2. Language constructs and functions for the `generate` section of the rules

Function/Construct	Description
`subs((S|[S]|E),`E_f`,`E_t`)`	Replace each occurrence of E_f in `(S|[S]|E)` for E_t
`if_then:{`E_{cond}`;(S|[S]|E);}`	If E_{cond} is true, then generate `(S|[S]|E)`
`if_then_else:{`E_{cond}`;` `(S|[S]|E)`$_t$`;(S|[S]|E)`$_e$`;}`	If E_{cond} is true, then generate `(S|[S]|E)`$_t$, else generate `(S|[S]|E)`$_e$
`gen_list:{[(S|[S]|E)];}`	Each statement/expression in `[(S|[S]|E)]` produces a different rule consequent.

```
undo_distributive {
    pattern: {
        (cexpr(b) * cexpr(a))
        + (cexpr(c) * cexpr(a)); }
    condition: {
        pure(cexpr(a));
        pure(cexpr(b));
        pure(cexpr(c)); }
    generate: {
        cexpr(a) *
        (cexpr(b) + cexpr(c)); }
}
```

Fig. 3. Distributive property

Fig. 3 shows a simple example: a rule which applies the distributive property to optimize code by transforming code like `((a[i] - 1) * v[i]) + (v[i] * f(b,3))` into `v[i] * ((a[i] - 1) + f(b,3))`. Meta-variables are marked to denote their role, i.e. what type of syntactic entity they can match: an expression (`cexpr(·)`), a statement (`cstmt(·)`), or a sequence of statements (`cstmts(·)`). In the example, meta-variables `a`, `b`, and `c` will only match expressions. Tables 1 and 2 briefly describe additional constructions which can be used in the `condition` and `generate` sections to check for properties of the code being matched (e.g., `is_identity(·, ·)`) and to have a more flexible and powerful code generation (e.g., `subs(·,·,·)`). In these tables, `E` represents an expression, `S` represents a statement, and [S] represents a sequence of statements. Additionally, other constructs such as `bin_oper(`E_{op}`,`E_l`,`E_r`)` can be used both to match and generate previously matched syntactic constructs (a binary operand, in this case). The tables are not meant to be exhaustive and, in fact, they can be extended to incorporate whatever property *imported* from external analysis tools.

```
aug_addition_assign {
    pattern:   {
        cexpr(a) += cexpr(b); }
    condition: {
        pure(cexpr(a)); }
    generate: {
        cexpr(a) =
          cexpr(a) + cexpr(b); }
}
```

(a) STML Rule.

```
--aug_addition_assign
rule_aug_addition_assign_462
   old@(CAssign CAddAssOp var_a_463 var_b_464 _) | True &&
         (null (allDefs [(CBlockStmt (CExpr (Just var_a_463)
         undefNode))])) =
     [("aug_addition_assign",old,(CAssign CAssignOp var_a_463
         (CBinary CAddOp var_a_463 var_b_464 undefNode)
         undefNode))]
rule_aug_addition_assign_462 _ = []
```

(b) Haskell code.

Fig. 4. Augmented addition: STML rule and Haskell code

2.2 Matching and Generating Code Through Synthesized Haskell

The actual transformation of the AST is performed by Haskell code automatically synthesized from STML rules, and contained in the file `Rules.hs`. We can classify STML rules among those which operate at expression level (easier to implement) and those which can manipulate both expressions and (sequences of) statements. The latter need to consider sequences of statements (`cstmts`) of unbound size, for which Haskell code that explicitly performs an AST traversal is generated.

When generating Haskell code, the rule sections (`pattern`, `condition`, `generate`, `assert`) generate the corresponding LHS's, guards, and RHS's of a Haskell function. If the conditions to apply a rule are met, the result is returned in a triplet `(rule_name, old_code, new_code)` where the two last components are, respectively, the matched and transformed sections of the AST. Since several rules can be applied at several locations of the AST, the result of applying the Haskell function implementing an STML rule is a list of tuples, one for each rule and location where the rule can be applied. This list will in a future have a heuristically determined benefit associated, which would make it possible to prioritize them. The transformation stops when either no more rules are applicable, or a stop condition is found – e.g., no applicable rule increase code quality above some threshold or a maximum number of rule applications is reached.

Example 1 (Augmented Addition). The rule in Fig. 4a transforms the augmented assignment `+=` to a simple assignment: `x += f(3)` is transformed into `x = x + f(3)`. Fig. 4b shows its Haskell translation. Note that `v[i++] += 1` can not be transformed because `v[i++]` is not *pure* – one of the conditions required by the rule. When conditions are present in the rule, the transformation express them in the Haskell code. In this case, the purity condition `pure(cexpr(a))` is translated to the Haskell guard `(null (allDefs [(CBlockStmt (CExpr (Just var_a_463) undefNode))]))`, which constructs an *artificial* block of statements containing only the expression `var_a_463` and checks that the list of variables assigned inside it (returned by function `allDefs` from `RulesLib.hs`) is empty. Symbols `CBlockStmt`, `CExpr`, and `undefNode` are defined in `Language.C`.

Example 2 (Undo Distributive). Consider again the STML rule in Fig. 3. This rule is translated into the code in Fig. 5, where some clauses have been omitted:

```
-- undo_distributive
rule_undo_distributive_75
   old@(CBinary CAddOp (CBinary CMulOp var_b_76 var_a_77 _) (CBinary CMulOp var_c_78 var_a_79
        _) _)
   | (exprEqual var_a_79 var_a_77) && True
         && (null (allDefs [(CBlockStmt (CExpr (Just var_a_79) undefNode))]))
         && (null (allDefs [(CBlockStmt (CExpr (Just var_b_76) undefNode))]))
         && (null (allDefs [(CBlockStmt (CExpr (Just var_c_78) undefNode))])) =
   [("undo_distributive",old,(CBinary CMulOp var_a_79 (CBinary CAddOp var_b_76 var_c_78
        undefNode) undefNode))]
...
rule_undo_distributive_75 _ = []
```

Fig. 5. Haskell code compiled from the `undo_distributive` rule

the commutative properties of addition and multiplication, known by the tool, force the generation of eight clauses. Checking the applicability of the rule (either because of the pattern or because of the `conditions`) is again implemented as clause guards. One example of the former is `(exprEqual var_a_79 var_a_77)`, that checks that both expressions matching "`a`" are indeed the same. The code for `pure(cexpr(a))` (cf. `b` and `c`) is the same as in the previous example.

```
useless_assign {
    pattern: {
        cstmts(ini);
        cexpr(lhs) = cexpr(lhs);
        cstmts(fin); }
    condition: { pure(cexpr(lhs)); }
    generate: {
        cstmts(ini);
        cstmts(fin);}
}
```

Fig. 6. Rule to remove useless assignments

Our final example shows the translation of a rule which transforms a sequence of statements. The code produced for this case is more complex than for the case of expression-transforming rules. Due to space limitations, we will just provide some insight on how the translation is done.

Example 3 (Useless Assignment Removal). The STML rule in Fig. 6 removes an assignment that does not change the expression being evaluated nor the l-value, i.e. it would remove `v[i] = v[i]`, but not `v[i++] = v[i++]` because `v[i++]` is not pure.[4] Fig. 7 shows its Haskell translation. The helper function `rule_useless_assign_503` searches for the rule pattern: the assignment `cexpr(lhs)= cexpr(lhs)` and its surrounding "holes" (cf. `ctstms(_)`). Function `rule_useless_assign_504` builds the consequent of the rule, checking the guard conditions and generating, for each occurrence of the pattern in the block, the corresponding consequent. The rule itself is implemented by function `rule_useless_assign_501` which is, essentially, a Haskell list comprehension calling `rule_useless_assign_503` in its *generator* expression and `rule_useless_assign_504` in the *construction* expression, to return the list of triples.

[4] It is debatable whether that rule can be applied to human-produced code. However, it is useful when several rules are chained (see the *Identity Matrix* example at http://goo.gl/LWRNOy).

```
-- useless_assign
rule_useless_assign_501 old@(CCompound ident bs nodeInfo) =
  (concat [rule_useless_assign_504 item ident nodeInfo old | item@(True,_,_) <-
       (rule_useless_assign_503 bs [] True )]) ++ []
rule_useless_assign_501 _ = []

rule_useless_assign_504 (True,[var_ini_507,var_fin_508], [(CBlockStmt (CExpr (Just (CAssign
     CAssignOp var_lhs_505 var_lhs_506 _)) _))]) ident nodeInfo old | (exprEqual var_lhs_506
     var_lhs_505) && (null (allDefs [(CBlockStmt (CExpr (Just var_lhs_506) undefNode))])) =
  concat ([[("useless_assign", old, (CCompound ident (var_ini_507 ++ var_fin_508)
       nodeInfo))] | True] ++ [])
rule_useless_assign_504 _ _ _ _ _ = []

rule_useless_assign_503 [] acc False = [(True,[acc],[])]
rule_useless_assign_503 [] acc _ = [(False,[acc],[])]
rule_useless_assign_503 (stat@(CBlockStmt (CExpr (Just (CAssign CAssignOp var_lhs_509
     var_lhs_510 _)) _)):tail_) accsl True =
  let
    listItems = rule_useless_assign_503 tail_ [] False
  in [(True, (accsl:(inter)),(stat:(pats))) | (True,inter,pats) <- listItems] ++
       (rule_useless_assign_503 tail_ (accsl ++ [stat]) True)
rule_useless_assign_503 (other:tail_) acc bool1 =
  rule_useless_assign_503 tail_ (acc ++ [other]) bool1
```

Fig. 7. Haskell code obtained from rule `useless_assign`

3 Experimental Evaluation

We have implemented (among others) the transformation rules mentioned in Fig. 1 and checked that the tool can successfully apply them in sequence. We have also carried out a preliminary performance test. The code in Fig. 1, which implements the C equivalent of the linear algebra $\mathbf{c} = a\mathbf{v}+b\mathbf{v}$, was compiled using `gcc -O3` in all cases and cache misses, number of floating point (FP) operations, and execution time were measured. We performed the evaluation with `gcc 4.47` in a Linux CentOS 6.5 with kernel 2.6.32 running in an Intel i7 3770, using PAPI 5.4.0 to profile the execution. The number of cache misses is of interest for CPU-based (multicore) architectures, and therefore also relevant for parallel platforms using the OpenMP and MPI programming models. The number of FP operations is interesting for CPU-based architectures and also for systems with scarce computational resources (like FPGAs and SoC platforms).

Table 3 shows, cumulatively, the effect of the transformation sequence in Fig. 1 on cache misses, FP operations, and execution time as percentages of the initial values (which is, for execution time, 46 ms.). These values are significantly reduced: ∼50% for execution time and L1 misses and ∼33% for FP operations. These results were obtained averaging values for 30 runs. For all parameters, the standard deviation was lower than 0.46% during the runs.

4 Conclusions and Future Work

We have briefly presented the goals and internal design of a tool to perform rule-based refactoring of procedural programs. The tool is written in Haskell, and the rules it executes are of a declarative nature. The use of Haskell (instead

Table 3. Impact of successive code transformations in Fig. 1 on several metrics

Metric	original	For-LoopFusion	AugAdditionAssign	JoinAssignments	UndoDistribute	LoopInvCodeMotion
TIME	100,00	50,23	50,22	49,83	49,85	49,32
FP_OPS	100,00	99,88	99,89	99,95	33,34	33,34
L1_MISS	100,00	49,62	49,62	49,62	49,62	49,62

of, for example, the infrastructure provided by LLVM) has proven to simplify and accelerate the development of the tool without compromising its speed / scalability so far. An experimental validation of a simple but relevant case, which uses algebraic properties of the code under transformation, has shown substantial improvements. The tool (http://goo.gl/yuOFiE) is being applied to a series of examples [9] elicited within the POLCA project, and to other examples (http://goo.gl/LWRNOy) where e.g., complexity reductions have been achieved for some cases.

As future work, we plan to implement metrics-based heuristics to perform an automatic (guided) search through the space of transformations. While we have already defined some metrics to determine the *adequacy* of transformations for different architectures, this is ongoing research within the consortium. We plan also to improve the interface to external analysis tools, of which we have identified those performing dependency analysis (e.g., polytope-based compilation) and reasoning over heap pointers (e.g., separation logic) as immediately applicable. A (more) formal definition of STML is now on the works.

References

1. Bagge, O.S., Kalleberg, K.T., Visser, E., Haveraaen, M.: Design of the codeboost transformation system for domain-specific optimisation of C++ programs. In: 3rd. Intl. Workshop on Source Code Analysis and Manipulation (SCAM2003), pp. 65–75. IEEE (2003)
2. Boekhold, M., Karkowski, I., Corporaal, H.: High-performance computing and networking. In: Sloot, P.M.A., Hoekstra, A.G., Bubak, M., Hertzberger, B. (eds.) HPCN-Europe 1999. LNCS, vol. 1593, pp. 673–682. Springer, Heidelberg (1999)
3. Brown, A., Luk, W., Kelly, P.: Optimising Transformations for Hardware Compilation. Tech. rep, Imperial College London (2005)
4. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Proc. of the 3rd Workshop on General-Purpose Computation on GPUs, pp. 63–74. ACM (2010)
5. Huber, B.: The `language-c` Package (2014). https://hackage.haskell.org/package/language-c
6. Lammel, R., Jones, S.P., Magalhaes, J.P.: The `syb` Package (2009). https://hackage.haskell.org/package/syb
7. Lindtjorn, O., Clapp, R.G., Pell, O., Fu, H., Flynn, M.J., Mencer, O.: Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications. IEEE Micro **31**(2), 41–49 (2011)

8. Schupp, S., Gregor, D., Musser, D., Liu, S.M.: Semantic and Behavioral Library Transformations. Information and Software Technology **44**(13), 797–810 (2002)
9. The POLCA Consortium: Specific Use Case Requirements. POLCA Project (#610686) Deliverable D-5.1, February 2014
10. Visser, E.: Program transformation with Stratego/XT. In: Lengauer, C., Batory, D., Blum, A., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)

On Logic Programming Representations of Lambda Terms: de Bruijn Indices, Compression, Type Inference, Combinatorial Generation, Normalization

Paul Tarau(✉)

Department of Computer Science and Engineering,
University of North Texas, Denton, USA
tarau@cse.unt.edu

Abstract. We introduce a compressed de Bruijn representation of lambda terms and define its bijections to standard representations. Compact combinatorial generation algorithms are given for several families of lambda terms, including open, closed, simply typed and linear terms as well as type inference and normal order reduction algorithms. We specify our algorithms as a literate Prolog program. In the process, we rely in creative ways on unification of logic variables, cyclic terms, backtracking and definite clause grammars.

Keywords: Lambda calculus · de Bruijn indices · Lambda term compression · Type inference · Normalization · Combinatorics of lambda terms

1 Introduction

Lambda terms [1] provide a foundation to modern functional languages, type theory and proof assistants and have been lately incorporated into mainstream programming languages including Java 8, C# and Apple's Swift. Generation of lambda terms has practical applications to testing compilers that rely on lambda calculus as an intermediate language, as well as in generation of random tests for user-level programs and data types. At the same time, several instances of lambda calculus are of significant theoretical interest given their correspondence with logic and proofs.

Prolog's underlying backtracking and unification make it an ideal tool for defining compact combinatorial generation algorithms for various families of lambda terms. Of particular interest are representations that are canonical up to alpha-conversion (variable renamings) among which the most well-known ones are de Bruijn's indices [2], representing bound variables as the number of binders to traverse to the lambda abstraction binding them.

However, a sequence of binders in de Bruijn notation, can be seen as a natural number expressed in unary notation. This suggests introducing a compressed

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 115–131, 2015.
DOI:10.1007/978-3-319-19686-2_9

representation of the binders that puts in a new light the underlying combinatorial structure of lambda terms and highlights their connection to the *Catalan family* of combinatorial objects [3], among which binary trees are the most well known. The proposed compressed de Bruijn notation also simplifies generation of some families of lambda terms.

At the same time, the use of Prolog's unification of logic variables is instrumental in designing compact algorithms for inferring simple types or for generating linear, linear affine or lambda terms with bounded unary height as well as in implementing normalization algorithms.

To be able to use the most natural representation for each of the proposed algorithms, we implement bijective transformations between lambda terms in standard as well as de Bruijn and compressed de Bruijn representation.

The paper is organized as follows. Section 2 introduces the compressed de Bruijn terms and bijective transformations from them to standard lambda terms. Section 3 describes generation of binary trees and mappings from lambda terms to binary trees representing their inferred types and and their applicative skeletons. Section 4 describes generators for several classes of lambda terms, including closed, simply typed, linear, affine as well as terms with bounded unary height and terms in the binary lambda calculus encoding. Section 5 describes a normal order reduction algorithm for lambda terms relaying on their de Bruijn representation. Section 6 discusses related work and section 7 concludes the paper.

The paper is structured as a literate Prolog program. The code has been tested with SWI-Prolog 6.6.6 and YAP 6.3.4. It is also available as a separate file at http://www.cse.unt.edu/~tarau/research/2015/dbx.pro.

2 A Compressed de Bruijn Representation of Lambda Terms

We represent standard lambda terms [1] in Prolog using the constructors `a/2` for applications and `l/2` for lambda abstractions. Variables bound by the lambdas as well as their occurrences are represented as *logic variables*. As an example, the lambda term $\lambda x0.(\lambda x1.(x0\ (x1\ x1))\ \lambda x2.(x0\ (x2\ x2)))$ will be represented as `l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))`.

2.1 De Bruijn Indices

De Bruijn indices [2] provide a *name-free* representation of lambda terms. All terms that can be transformed by a renaming of variables (α-conversion) will share a unique representation. Variables following lambda abstractions are omitted and their occurrences are marked with positive integers *counting the number of lambdas until the one binding them* is found on the way up to the root of the term. We represent them using the constructor `a/2` for application, `l/1` for lambda abstractions (that we will call shortly *binders*) and `v/1` for marking the integers corresponding to the de Bruijn indices.

For instance, the term `l(A,a(l(B,a(A,a(B,B))),l(C,a(A,a(C,C)))))` is represented as `l(a(l(a(v(1),a(v(0),v(0)))),l(a(v(1),a(v(0),v(0))))))`, corresponding to the fact that `v(1)` is bound by the outermost lambda (two steps away, counting from `0`) and the occurrences of `v(0)` are bound each by the closest lambda, represented by the constructor `l/1`.

From de Bruijn to Lambda Terms with Canonical Names. The predicate `b2l` converts from the de Bruijn representation to lambda terms whose canonical names are provided by logic variables. We will call them terms in *standard notation*.

```
b2l(DeBruijnTerm,LambdaTerm):-b2l(DeBruijnTerm,LambdaTerm,_Vs).

b2l(v(I),V,Vs):-nth0(I,Vs,V).
b2l(a(A,B),a(X,Y),Vs):-b2l(A,X,Vs),b2l(B,Y,Vs).
b2l(l(A),l(V,Y),Vs):-b2l(A,Y,[V|Vs]).
```

Note the use of the built-in `nth0/3` that associates to an index `I` a variable `V` on the list `Vs`. As we initialize in `b2l/2` the list of logic variables as a free variable `_Vs`, free variables in open terms, represented with indices larger than the number of available binders will also be consistently mapped to logic variables. By replacing `_Vs` with `[]` in the definition of `b2l/2`, one could enforce that only closed terms (having no free variables) are accepted.

From Lambda Terms with Canonical Names to de Bruijn Terms. Logic variables provide canonical names for lambda variables. An easy way to manipulate them at meta-language level is to turn them into special "$VAR/1" terms - a mechanism provided by Prolog's built-in `numbervars/3` predicate. Given that "$VAR/1" is distinct from the constructors lambda terms are built from (`l/2` and `a/2`), this is a safe (and invertible) transformation. To avoid any side effect on the original term, in the predicate `l2b/2` that inverts `b2l/2`, we will uniformly rename its variables to fresh ones with Prolog's `copy_term/2` built-in. We will adopt this technique through the paper each time our operations would mutate an input argument otherwise.

```
l2b(StandardTerm,DeBruijnTerm):-
  copy_term(StandardTerm,Copy),
  numbervars(Copy,0,_),
  l2b(Copy,DeBruijnTerm,_Vs).

l2b('$VAR'(V),v(I),Vs):-once(nth0(I,Vs,'$VAR'(V))).
l2b(a(X,Y),a(A,B),Vs):-l2b(X,A,Vs),l2b(Y,B,Vs).
l2b(l(V,Y),l(A),Vs):-l2b(Y,A,[V|Vs]).
```

Note the use of `nth0/3`, this time to locate the index `I` on the (open) list of variables `_Vs`. By replacing `_Vs` with `[]` in the call to `l2b/3`, one can enforce that only closed terms are accepted.

Example 1. *Illustrates the bijection defined by predicates* `l2b` *and* `b2l`.

```
?- LT=l(A,l(B,l(C,a(a(A,C),a(B,C))))),l2b(LT,BT),b2l(BT,LT1),LT=LT1.
LT = LT1, LT1 = l(A, l(B, l(C, a(a(A, C), a(B, C))))),
BT = l(l(l(a(a(v(2), v(0)), a(v(1), v(0)))))).
```

2.2 Going One Step Further: Compressing the Blocks of Lambdas

Iterated lambdas (represented as a block of constructors `l/1` in the de Bruijn notation) can be seen as a successor arithmetic representation of a number that counts them. So it makes sense to represent that number more efficiently in the usual binary notation. Note that in de Bruijn notation blocks of lambdas can wrap either applications or variable occurrences represented as indices. This suggests using just two constructors: `v/2` indicating in a term `v(K,N)` that we have `K` lambdas wrapped around variable `v(N)` and `a/3`, indicating in a term `a(K,X,Y)` that `K` lambdas are wrapped around the application `a(X,Y)`.

We call the terms built this way with the constructors `v/2` and `a/3` *compressed de Bruijn terms*.

2.3 Converting Between Representations

We can make precise the definition of compressed deBruijn terms by providing a bijective transformation between them and the usual de Bruijn terms.

From de Bruijn to Compressed. The predicate `b2c` converts from the usual de Bruijn representation to the compressed one. It proceeds by case analysis on `v/1, a/2, l/1` and counts the binders `l/1` as it descends toward the leaves of the tree. Its steps are controlled by the predicate `up/2` that increments the counts when crossing a binder.

```
b2c(v(X),v(0,X)).
b2c(a(X,Y),a(0,A,B)):-b2c(X,A),b2c(Y,B).
b2c(l(X),R):-b2c1(0,X,R).

b2c1(K,a(X,Y),a(K1,A,B)):-up(K,K1),b2c(X,A),b2c(Y,B).
b2c1(K, v(X),v(K1,X)):-up(K,K1).
b2c1(K,l(X),R):-up(K,K1),b2c1(K1,X,R).

up(From,To):-From>=0,To is From+1.
```

From Compressed to de Bruijn. The predicate `c2b` converts from the compressed to the usual de Bruijn representation. It reverses the effect of `b2c` by expanding the `K` in `v(K,N)` and `a(K,X,Y)` into `K l/1` binders (no binders when `K=0`). The predicate `iterLam/3` performs this operation in both cases, and the predicate `down/2` computes the decrements at each step. We will reuse the predicates `up/2` and `down/2` that can be seen as abstracting away the successor/predecessor operation.

```
c2b(v(K,X),R):-X>=0,iterLam(K,v(X),R).
c2b(a(K,X,Y),R):-c2b(X,A),c2b(Y,B),iterLam(K,a(A,B),R).

iterLam(0,X,X).
iterLam(K,X,l(R)):-down(K,K1),iterLam(K1,X,R).

down(From,To):-From>0,To is From-1.
```

Example 2. *Illustrates the bijection defined by the predicates* `b2c` *and* `c2b`.

```
?- BT=l(l(l(a(a(v(2), v(0)), a(v(1), v(0))))),b2c(BT,CT),c2b(CT,BT1).
BT = BT1, BT1 = l(l(l(a(a(v(2), v(0)), a(v(1), v(0))))),
CT = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))) .
```

A convenient way to simplify defining chains of such conversions is by using Prolog's DCG transformation. For instance, the predicate `c2l/2` (which expands to something like `c2l(X,Z):-c2b(X,Y),b2l(Y,Z)`), converts from compressed de Bruijn terms and standard lambda terms using de Bruijn terms as an intermediate step, while `l2c/2` works the other way around.

```
c2l --> c2b,b2l.
l2c --> l2b,b2c.
```

2.4 Open and Closed Terms

Lambda terms might contain *free variables* not associated to any binders. Such terms are called *open*. A *closed* term is such that each variable occurrence is associated to a binder.

Closed terms can be easily identified by ensuring that the lambda binders on a given path from the root outnumber the de Bruijn index of a variable occurrence ending the path. The predicate `isClosed` does that for compressed de Bruijn terms.

```
isClosed(T):-isClosed(T,0).

isClosed(v(K,N),S):-N<S+K.
isClosed(a(K,X,Y),S1):-S2 is S1+K,isClosed(X,S2),isClosed(Y,S2).
```

3 Binary Trees, Lambda Terms and and Types

We can see our compressed de Bruijn terms as binary trees decorated with integer labels. The binary trees provide a skeleton that describes the applicative structure of the underlying lambda terms. At the same time, types in the *simple typed lambda calculus* [4] share a similar binary tree structure.

Binary trees are among the most well-known members of the Catalan family of combinatorial objects [3], that has at least 58 structurally distinct members, covering several data structures, geometric objects and formal languages.

Generating Binary Trees. We will build binary trees with the constructor `->/2` for branches and the constant `o` for its leaves. This will match the usual notation for simple types [4] of lambda terms that can be represented as binary trees.

A generator / recognizer of binary trees of a fixed size (seen as the number of internal nodes, counted by entry `A000108` in [5]) is defined by the predicate `scat/2`.

```
scat(N,T):-scat(T,N,0).

scat(o)-->[].
scat((X->Y))-->down,scat(X),scat(Y).
```

Note the creative use of Prolog's DCG-grammar transformation. After the DCG expansion, the code for `scat/3` becomes something like:

```
scat(o,K,K).
scat((X->Y),K1,K3):-down(K1,K2),scat(X,K2,K3),scat(K3,K4).
```

Given that down(K1,K2) unfolds to `K1>0,K2 is K1-1` it is clear that this code ensures that the total number of nodes `N` passed by `scat/2` to `scat/3` controls the size of the generated trees. We will reuse this pattern through the paper, as it simplifies the writing of generators for various combinatorial objects. It is also convenient to standardize on the number of *internal nodes* as defining the *size* of our terms.

Example 3. *Illustrates trees with 3 internal nodes (built with the constructor "`->/2`") generated by* `scat/2`.

```
?- scat(3,BT).
BT = (o->o->o->o) ;
BT = (o-> (o->o)->o) ;
BT = ((o->o)->o->o) ;
BT = ((o->o->o)->o) ;
BT = (((o->o)->o)->o) .
```

Note the right associative constructor "`->`" reducing the use of parentheses.

3.1 Type Inference with Logic Variables

Simple types, represented as binary trees built with the constructor "`->/2`" with empty leaves representing the unique primitive type "`o`", can be seen as a "Catalan approximation" of lambda terms, centered around ensuring their safe and terminating evaluation (strong normalization).

While in a functional language inferring types requires implementing unification with occur check, as shown for instance in [6], this operation is available in Prolog as a built-in. Also a "post-mortem" verification that unification has not introduced any cycles is provided by the built-in `acyclic_term/1`.

The predicate `extractType/2` works by turning each logical variable `X` into a pair `_:TX`, where `TX` is a fresh variable denoting its type. As logic variable

bindings propagate between binders and occurrences, this ensures that types are consistently inferred.

```
extractType(_:TX,TX):-!. % this matches all variables
extractType(l(_:TX,A),(TX->TA)):-extractType(A,TA).
extractType(a(A,B),TY):-extractType(A,(TX->TY)),extractType(B,TX).
```

Instead of (inefficiently) using unification with occur-check at each step, we ensure that at the end, our term is still *acyclic*, by using the built-in ISO-Prolog predicate `acyclic_term/1`.

```
hasType(CTerm,Type):-
  c2l(CTerm,LTerm),
  extractType(LTerm,Type),
  acyclic_term(LTerm),
  bindType(Type).
```

At this point, most general types are inferred by `extractType` as fresh variables, somewhat similar to multi-parameter polymorphic types in functional languages, if one interprets logic variables as universally quantified. However, as we are only interested in simple types, we will bind uniformly the leaves of our type tree to the constant "o" representing our only primitive type, by using the predicate `bindType/1`.

```
bindType(o):-!.
bindType((A->B)):-bindType(A),bindType(B).
```

We can also define the predicate `typeable/1` that checks if a lambda term is well typed, by trying to infer and then ignoring its inferred type.

```
typeable(Term):-hasType(Term,_Type).
```

Example 4. *Illustrates typability of the term corresponding to the* `S` *combinator* $\lambda x0.\ \lambda x1.\ \lambda x2.((x0\ x2)\ (x1\ x2))$ *and untypabilty of the term corresponding to the* `Y` *combinator* $\lambda x0.(\ \lambda x1.(x0\ (x1\ x1))\ \ \lambda x2.(x0\ (x2\ x2)))$*, in de Bruijn form.*

```
?- hasType(a(3,a(0,v(0,2),v(0,0)),a(0,v(0,1),v(0,0))),T).
T = ((o->o->o)-> (o->o)->o->o).
?- hasType(
   a(1,a(1,v(0,1),a(0,v(0,0),v(0,0))),a(1,v(0,1),a(0,v(0,0),v(0,0)))),T).
false.
```

4 Generating Special Classes of Lambda Terms

To generate lambda terms of a given size, we can write generators similar to the ones for binary trees in section 3. Moreover, we have the choice to use generators for standard, de Bruijn or compressed de Bruijn terms and then bijectively morph the resulting terms in the desired representation, as outlined is section 2.

Generating Motzkin Trees. Motzkin-trees (also called binary-unary trees) have internal nodes of arities 1 or 2. Thus they can be seen as an abstraction of lambda terms that ignores de Bruijn indices at the leaves. The predicate `motzkinTree/2` generates Motzkin trees with L internal and leaf nodes.

```
motzkinTree(L,T):-motzkinTree(T,L,0).

motzkinTree(u)-->down.
motzkinTree(l(A))-->down,motzkinTree(A).
motzkinTree(a(A,B))-->down,motzkinTree(A),motzkinTree(B).
```

Motzkin-trees are counted by the sequence `A001006` in [5]. If we replace the first clause of `motzkinTree/2` with `motzkinTree(u)-->[]`, we obtain binary-unary trees with `L` internal nodes, counted by the entry `A006318` (Large Schröder Numbers) of [5].

4.1 Generation of de Bruijn Terms

We can derive a generator for closed lambda terms in de Bruijn form by extending the Motzkin-tree generator to keep track of the lambda binders. When reaching a leaf `v/1`, one of the available binders (expressed as a de Bruijn index) will be assigned to it nondeterministically.

The predicate `genDB/4` generates closed de Bruijn terms with a fixed number of internal (non-index) nodes, as counted by entry `A220894` in [5].

```
genDB(v(X),V)-->{down(V,V0),between(0,V0,X)}.
genDB(l(A),V)-->down,{up(V,NewV)},genDB(A,NewV).
genDB(a(A,B),V)-->down,genDB(A,V),genDB(B,V).
```

The range of possible indices is provided by Prolog's built-in integer range generator `between/3` that provides values from `0` to `V0`.

Our generator of deBruijn terms is exposed through two interfaces: `genDB/2` that generates closed de Bruijn terms with exactly `L` non-index nodes and `genDBs/2` that generates terms with up to `L` non-index nodes, by not enforcing that exactly `L` internal nodes must be used.

```
genDB(L,T):-genDB(T,0,L,0).
genDBs(L,T):-genDB(T,0,L,_).
```

Like in the case of Motzkin trees, a slight modification of the first clause of `genDB/4` will enumerate terms counted by sequence `A135501` in [5].

Example 5. *Illustrates the generation of terms with up to* `2` *internal nodes.*

```
?- genDBs(2,T).
T = l(v(0)) ;
T = l(l(v(0))) ;
T = l(l(v(1))) ;
T = l(a(v(0), v(0))) ;
```

4.2 Generators for Closed Terms in Compressed de Bruijn Form

A generator for compressed de Bruijn terms can be derived by using `DCG` syntax to compose a generator for closed de Bruijn terms `genDB` and `genDBs` and a transformer to compressed terms `b2c/2`.

```
genCompressed --> genDB,b2c.
genCompresseds--> genDBs,b2c.
```

4.3 Generators for Closed Terms in Standard Notation

```
genStandard-->genDB,b2l.
genStandards-->genDBs,b2l.
```

Example 6. *Illustrates generators for closed terms in compressed de Bruijn and standard notation with logic variables providing lambda variable names.*

```
?- genCompressed(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)).

?- genStandard(2,T).
T = l(_G3434, l(_G3440, _G3440)) ;
T = l(_G3434, l(_G3440, _G3434)) ;
T = l(_G3437, a(_G3437, _G3437)).
```

4.4 Generating Closed Lambda Terms in Standard Notation

With logic variables representing binders and their occurrences, one can also generate lambda terms in standard notation directly. The predicate `genLambda/2` equivalent to `genStandard/2`, builds a list of logic variables as it generates binders. When generating a leaf, it picks nondeterministically one of the binders among the list of binders available, `Vs`. As usual, the predicate `down/2` controls the number of internal nodes.

```
genLambda(L,T):-genLambda(T,[],L,0).

genLambda(X,Vs)-->{member(X,Vs)}.
genLambda(l(X,A),Vs)-->down,genLambda(A,[X|Vs]).
genLambda(a(A,B),Vs)-->down,genLambda(A,Vs),genLambda(B,Vs).
```

4.5 Generating Typeable Terms

The predicate `genTypeable/2` generates closed typeable terms of size `L`. These are counted by entry `A220471` in [5].

```
genTypeable(L,T):-genCompressed(L,T),typeable(T).
genTypeables(L,T):-genCompresseds(L,T),typeable(T).
```

Example 7. *Illustrates a generator for closed typeable terms.*

```
?- genCompressed(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)).
```

4.6 Generating Normal Forms

Normal forms are lambda terms that cannot be further reduced. A normal form should not be an application with a lambda as its left branch and, recursively, its subterms should also be normal forms. The predicate `nf/4` defines this inductively and generates all normal forms with `L` internal nodes in de Bruijn form.

```
nf(v(X),V)-->{down(V,V0),between(0,V0,X)}.
nf(l(A),V)-->down,{up(V,NewV)},nf(A,NewV).
nf(a(v(X),B),V)-->down,nf(v(X),V),nf(B,V).
nf(a(a(X,Y),B),V)-->down,nf(a(X,Y),V),nf(B,V).
```

As we standardize our generators to produce compressed de Bruijn terms, we combine `nf/4` and the converter `b2c/2` to produce normal forms of size exactly `L` (predicate `nf/2`) and with size up to `L` (predicate `nfs/2`).

```
nf(L,T):-nf(B,0,L,0),b2c(B,T).
nfs(L,T):-nf(B,0,L,_),b2c(B,T).
```

Example 8. *Illustrates normal forms with exactly 2 non-index nodes.*

```
?- nf(2,T).
T = v(2, 0) ;
T = v(2, 1) ;
T = a(1, v(0, 0), v(0, 0)) .
```

The number of solutions of our generator replicates entry `A224345` in [5] that counts closed normal forms of various sizes.

4.7 Generation of Linear Lambda Terms

Linear lambda terms [7] restrict binders to *exactly one* occurrence.

The predicate `linLamb/4` uses logic variables both as leaves and as lambda binders and generates terms in standard form. In the process, binders accumulated on the way down from the root, must be split between the two branches of an application node. The predicate `subset_and_complement_of/3` achieves this by generating all such possible splits of the set of binders.

```
linLamb(X,[X])-->[].
linLamb(l(X,A),Vs)-->down,linLamb(A,[X|Vs]).
linLamb(a(A,B),Vs)-->down,
  {subset_and_complement_of(Vs,As,Bs)},
  linLamb(A,As),linLamb(B,Bs).
```

At each step of `subset_and_complement_of/3`, `place_element/5` is called to distribute each element of a set to exactly one of two disjoint subsets.

```
subset_and_complement_of([],[],[]).
subset_and_complement_of([X|Xs],NewYs,NewZs):-
  subset_and_complement_of(Xs,Ys,Zs),
  place_element(X,Ys,Zs,NewYs,NewZs).
```

```
place_element(X,Ys,Zs,[X|Ys],Zs).
place_element(X,Ys,Zs,Ys,[X|Zs]).
```

As usual, we standardize the generated terms by converting them with `l2c` to compressed de Bruijn terms.

```
linLamb(L,CT):-linLamb(T,[],L,0),l2c(T,CT).
```

Example 9. *Illustrates linear lambda terms for* `L=3`.

```
?- linLamb(3,T).
T = a(2, v(0, 1), v(0, 0)) ;
T = a(2, v(0, 0), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(1, 0), v(0, 0)) ;
T = a(0, v(1, 0), v(1, 0)) .
```

4.8 Generation of Affine Linear Lambda Terms

Linear affine lambda terms [7] restrict binders to *at most one* occurrence.

```
afLinLamb(L,CT):-afLinLamb(T,[],L,0),l2c(T,CT).

afLinLamb(X,[X|_])-->[].
afLinLamb(l(X,A),Vs)-->down,afLinLamb(A,[X|Vs]).
afLinLamb(a(A,B),Vs)-->down,
  {subset_and_complement_of(Vs,As,Bs)},
  afLinLamb(A,As),afLinLamb(B,Bs).
```

Example 10. *Illustrates generation of affine linear lambda terms in compressed de Bruijn form.*

```
?- afLinLamb(3,T).
T = v(3, 0) ;
T = a(2, v(0, 1), v(0, 0)) ;
T = a(2, v(0, 0), v(0, 1)) ;
T = a(1, v(0, 0), v(1, 0)) ;
T = a(1, v(1, 0), v(0, 0)) ;
T = a(0, v(1, 0), v(1, 0)) ;
```

Clearly all linear terms are affine. It is also known that all affine terms are typeable.

4.9 Generating Lambda Terms of Bounded Unary Height

Lambda terms of bounded unary height are introduced in [8] where it is argued that such terms are naturally occurring in programs and it is shown that their asymptotic behavior is easier to study.

They are specified by giving a bound on the number of lambda binders from a de Bruijn index to the root of the term.

```
boundedUnary(v(X),V,_D)-->{down(V,V0),between(0,V0,X)}.
boundedUnary(l(A),V,D1)-->down,
  {down(D1,D2),up(V,NewV)},
  boundedUnary(A,NewV,D2).
boundedUnary(a(A,B),V,D)-->down,
  boundedUnary(A,V,D),boundedUnary(B,V,D).
```

The predicate `boundedUnary/5` generates lambda terms of size `L` in compressed de Bruijn form with unary hight `D`.

```
boundedUnary(D,L,T):-boundedUnary(B,0,D,L,0),b2c(B,T).
boundedUnarys(D,L,T):-boundedUnary(B,0,D,L,_),b2c(B,T).
```

Example 11. *Illustrates terms of unary height 1 with size up to 3.*

```
?- boundedUnarys(1,3,R).
R = v(1, 0) ;
R = a(1, v(0, 0), v(0, 0)) ;
R = a(1, v(0, 0), a(0, v(0, 0), v(0, 0))) ;
R = a(1, a(0, v(0, 0), v(0, 0)), v(0, 0)) ;
R = a(0, v(1, 0), v(1, 0)) .
```

4.10 Generating Terms in Binary Lambda Calculus Encoding

Generating de Bruijn terms based on the size of their binary lambda calculus encoding [9] works by using a DCG mechanism to build the actual code as a list `Cs` of `0` and `1` digits and specifying the size of the code in advance.

```
blc(L,T,Cs):-length(Cs,L),blc(B,0,Cs,[]),b2c(B,T).

blc(v(X),V)-->{between(1,V,X)},encvar(X).
blc(l(A),V)-->[0,0],{NewV is V+1},blc(A,NewV).
blc(a(A,B),V)-->[0,1],blc(A,V),blc(B,V).
```

Note that de Bruijn binders are encoded as `00`, applications as `01` and de Bruijn indices in unary notation are encoded as `00...01`. This operation is preformed by the predicate `encvar/3`, that, in DCG notation, uses `down/2` at each step to generate the sequence of `1` terminated `0` digits.

```
encvar(0)-->[0].
encvar(N)-->{down(N,N1)},[1],encvar(N1).
```

Example 12. *Illustrates generation of 8-bit binary lambda terms (*`Cs`*) together with their compressed de Bruijn form (*`T`*).*

```
?- blc(8,T,Cs).
T = v(3, 1),
Cs = [0, 0, 0, 0, 0, 0, 1, 0] ;
T = a(1, v(0, 1), v(0, 1)),
Cs = [0, 0, 0, 1, 1, 0, 1, 0] .
```

Note that while not bijective, the binary encoding has the advantage of being a self-delimiting code. This facilitates its use in an unusually compact interpreter.

5 Normalization of Lambda Terms

Evaluation of lambda terms involves *β-reduction*, a transformation of a term like `a(l(X,A),B)` by replacing every occurrence of `X` in `A` by `B`, under the assumption that `X` does not occur in `B` and *η-conversion*, the transformation of an application term `a(l(X,A),X)` into `A`, under the assumption that X does not occur in `A`.

The first tool we need to implement normalization of lambda terms is a safe substitution operation. In lambda-calculus based functional languages this can be achieved through a HOAS (Higher-Order Abstract Syntax) mechanism, that borrows the substitution operation from the underlying "meta-language". To this end, lambdas are implemented as functions which get executed (usually lazily) when substitutions occur. We refer to [10] for the original description of this mechanism, widely used these days for implementing embedded domain specific languages and proof assistants in languages like Haskell or ML.

While logic variables offer a fast and easy way to perform *substitutions*, they do not offer any elegant mechanism to ensure that substitutions are *capture-free*. Moreover, no HOAS-like mechanism exists in Prolog for borrowing anything close to *normal order reduction* from the underlying system, as Prolog would provide, through meta-programming, only a *call-by-value* model.

We will devise here a simple and safe interpreter for lambda terms supporting normal order β-reduction by using de Bruijn terms, which also ensures that terms are unique up to α-equivalence. As usual, we will omit η-conversion, known to interfere with things like type inference, as the redundant argument(s) that it removes might carry useful type information.

The predicate `beta/3` implements the β-conversion operation corresponding to the binder `l(A)`. It calls `subst/4` that replaces in `A` occurrences corresponding the the binder `l/1`.

```
beta(l(A),B,R):-subst(A,0,B,R).
```

The predicate `subst/4` counts, starting from `0` the lambda binders down to an occurrence `v(N)`. Replacement occurs at at level `I` when `I=N`.

```
subst(a(A1,A2),I,B,a(R1,R2)):-I>=0,
  subst(A1,I,B,R1),
  subst(A2,I,B,R2).
```

```
subst(l(A),I,B,l(R)):-I>=0,I1 is I+1,subst(A,I1,B,R).
subst(v(N),I,_B,v(N1)):-I>=0,N>I,N1 is N-1.
subst(v(N),I,_B,v(N)):-I>=0,N<I.
subst(v(N),I,B,R):-I>=0,N=:=I,shift_var(I,0,B,R).
```

When the right occurrence `v(N)` is reached, the term substituted for it is shifted such that its variables are marked with the new, incremented distance to their binders. The predicate `shift_var/4` implements this operation.

```
shift_var(I,K,a(A,B),a(RA,RB)):-K>=0,I>=0,
  shift_var(I,K,A,RA),
  shift_var(I,K,B,RB).
shift_var(I,K,l(A),l(R)):-K>=0,I>=0,K1 is K+1,shift_var(I,K1,A,R).
shift_var(I,K,v(N),v(M)):-K>=0,I>=0,N>=K,M is N+I.
shift_var(I,K,v(N),v(N)):-K>=0,I>=0,N<K.
```

Normal order evaluation of a lambda term, if it terminates, leads to a unique normal form, as a consequence of the Church-Rosser theorem, elegantly proven in [2] using de Bruijn terms. Termination holds, for instance, in the case of simply typed lambda terms. Its implementation is well known; we will follow here the algorithm described in [11]. We first compute the *weak head normal form* using `wh_nf/2`.

```
wh_nf(v(X),v(X)).
wh_nf(l(E),l(E)).
wh_nf(a(X,Y),Z):-wh_nf(X,X1),wh_nf1(X1,Y,Z).
```

The predicate `wh_nf1/3` does the case analysis of application terms `a/2`. The key step is the β-reduction in its second clause, when it detects an "eliminator" lambda expression as its left argument, in which case it performs the substitution of its binder, with its right argument.

```
wh_nf1(v(X),Y,a(v(X),Y)).
wh_nf1(l(E),Y,Z):-beta(l(E),Y,NewE),wh_nf(NewE,Z).
wh_nf1(a(X1,X2),Y,a(a(X1,X2),Y)).
```

The predicate `to_nf` implements normal order reduction. It follows the same skeleton as `wh_nf`, which is called in the third clause to perform reduction to weak head normal form, starting from the outermost lambda binder.

```
to_nf(v(X),v(X)).
to_nf(l(E),l(NE)):-to_nf(E,NE).
to_nf(a(E1,E2),R):-wh_nf(E1,NE),to_nf1(NE,E2,R).
```

Case analysis of application terms for possible β-reduction is performed by `to_nf1/3`, where the second clause calls `beta/3` and recurses on its result.

```
to_nf1(v(E1),E2,a(v(E1),NE2)):-to_nf(E2,NE2).
to_nf1(l(E),E2,R):-beta(l(E),E2,NewE),to_nf(NewE,R).
to_nf1(a(A,B),E2,a(NE1,NE2)):-to_nf(a(A,B),NE1),to_nf(E2,NE2).
```

The predicates `to_nf` provides a Turing-complete lambda calculus interpreter working on de Bruijn terms. It is guaranteed to compute a normal form, if it

exists. The predicate `evalStandard/2` works on standard lambda terms, that in converts to de Bruijn terms and then back after evaluation. The predicate `evalCompressed/2` works in a similar way on compressed de Bruijn terms. We express them as a composition of functions (first argument in, second out) using Prolog's DCG notation.

```
evalStandard-->l2b,to_nf,b2l.
evalCompressed-->c2b,to_nf,b2c.
```

Example 13. *Illustrates evaluation of the lambda term* $SKK = ((\ \lambda x0.\ \lambda x1.\ \lambda x2.((x0\ x2)\ (x1\ x2))\ \ \lambda x3.\ \lambda x4.x3)\ \ \lambda x5.\ \lambda x6.x5)$ *in compressed de Brijn form, resulting in the definition of the identity combinator* $I = \lambda x0.x0$.

```
?- S=a(3,a(0,v(0,2),v(0,0)),a(0,v(0,1),v(0,0))),K=v(2,1),
     evalCompressed(a(0,a(0,S,K),K),R).
S = a(3, a(0, v(0, 2), v(0, 0)), a(0, v(0, 1), v(0, 0))),
K = v(2, 1),
R = v(1, 0).
```

6 Related Work

The classic reference for lambda calculus is [1]. Various instances of typed lambda calculi are overviewed in [4]. De Bruijn's notation for lambda terms is introduced in [2]. The compressed de Bruijn representation of lambda terms proposed in this paper is novel, to our best knowledge.

The combinatorics and asymptotic behavior of various classes of lambda terms are extensively studied in [6,7,12]. Distribution and density properties of random lambda terms are described in [13].

Lambda terms of bounded unary height are introduced in [8]. John Tromp's binary lambda calculus is only described through online code and the Wikipedia entry at [9].

Generators for closed and well-typed lambda terms, as well as their normal forms, expressed as functional programming algorithms, are given in [6], derived from combinatorial recurrences. However, they are significantly more complex than the ones described here in Prolog. On the other hand, we have not found in the literature generators for linear, linear affine terms and lambda terms of bounded unary height. Normalization of lambda terms and its confluence properties are described in [1] and [14] with functional programming algorithms given in [11] and HOAS-based evaluation first described in [10].

In a logic programming context, unification of simply typed lambda terms has been used in as the foundation of the programming language λProlog [15,16] and applied to higher order logic programming [17].

7 Conclusion

We have described compact (and arguably elegant) combinatorial generation algorithms for several important families of lambda terms. Besides the newly

introduced a compressed form of de Bruijn terms we have used ordinary de Bruijn terms as well as a canonical representation of lambda terms relying on Prolog's logic variables. In each case, we have selected the representation that was more appropriate for tasks like combinatorial generation, type inference or normalization. We have switched representation as needed, though bijective transformers working in time proportional to the size of the terms. Our combinatorial generation algorithms match the corresponding sequence of counts by size, given in [5] as an empirical validation of their correctness. Our techniques, combining unification of logic variables with Prolog's backtracking mechanism and DCG grammar notation, recommend logic programming as an unusually convenient meta-language for the manipulation of various families of lambda terms and the study of their combinatorial and computational properties.

Acknowledgments. This research has been supported by NSF grant 1423324.

References

1. Barendregt, H.P.: The Lambda Calculus Its Syntax and Semantics, Revised edn., vol. 103. North Holland (1984)
2. Bruijn, N.G.D.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. Indagationes Mathematicae **34**, 381–392 (1972)
3. Stanley, R.P.: Enumerative Combinatorics. Wadsworth Publ. Co., Belmont (1986)
4. Barendregt, H.P.: Lambda calculi with types. In: Handbook of Logic in Computer Science, vol. 2. Oxford University Press (1991)
5. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences (2014). Published electronically at https://oeis.org/
6. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. J. Funct. Program. **23**(5), 594–628 (2013)
7. Grygiel, K., Idziak, P.M., Zaionc, M.: How big is BCI fragment of BCK logic. J. Log. Comput. **23**(3), 673–691 (2013)
8. Bodini, O., Gardy, D., Gittenberger, B.: Lambda-terms of bounded unary height. In: ANALCO, pp. 23–32. SIAM (2011)
9. Wikipedia: Binary lambda calculus – wikipedia, the free encyclopedia (2015) [Online; accessed 20-February-2015]
10. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. PLDI 1988, New York, pp. 199–208. ACM (1988)
11. Sestoft, P.: Demonstrating lambda calculus reduction. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 420–435. Springer, Heidelberg (2002)
12. David, R., Grygiel, K., Kozik, J., Raffalli, C., Theyssier, G., Zaionc, M.: Asymptotically almost all λ-terms are strongly normalizing. Preprint: arXiv: math. LO/0903.5505 v3 (2010)

13. David, R., Raffalli, C., Theyssier, G., Grygiel, K., Kozik, J., Zaionc, M.: Some properties of random lambda terms. Logical Methods in Computer Science **9**(1) (2009)
14. Kamareddine, F.: Reviewing the Classical and the de Bruijn Notation for calculus and Pure Type Systems. Journal of Logic and Computation **11**(3), 363–394 (2001)
15. Miller, D.: Unification of simply typed lambda-terms as logic programming. In: Proc. Int. Conference on Logic Programming (Paris), pp. 255–269. MIT Press (1991)
16. Nadathur, G., Mitchell, D.J.: System description: teyjus - a compiler and abstract machine based implementation of λprolog. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 287–291. Springer, Heidelberg (1999)
17. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press, New York (2012)

Programming Microcontrollers in OCaml: The OCaPIC Project

Benoît Vaugon[1(✉)], Philippe Wang[2], and Emmanuel Chailloux[2]

[1] U2IS, ENSTA ParisTech, Palaiseau, France
benoit.vaugon@ensta.fr
[2] LIP6, CNRS UMR 7606, UPMC, Paris, France
philippe.wang@gmail.com,
emmanuel.chailloux@lip6.fr

Abstract. PIC microcontrollers are low-cost programmable integrated circuits,consume very little energy, but are hard to program due to very little available resources. They are traditionally programmed using low-level languages (e.g., assembler or subsets of C), which provide very few safeguards if any. This paper presents the issues we had to solve to successfully port a modern multi-paradigm general-purpose programming language, which notably provides automatic memory management and strong static type checking, to this rather peculiar hardware.

Keywords: Microcontroller · Virtual machine · Programming language implementation · OCaml · Applications

1 Introduction

Microcontrollers are programmable integrated circuits. Within a single chip, they contain a processing unit, various volatile and nonvolatile memory units, and a set of "internal interfaces" facilitating communications with the outside world. They are designed to be programmed, and then placed in an electronic circuit in which they perform more or less complex work.

The PIC microcontrollers are commercialized by the Microchip company. There are hundreds of PIC microcontrollers models, classified in different series (PIC16, PIC18, etc.) according to their characteristics. Machine word size depends on the series: 8 bits for the most available series (PIC10, PIC12, PIC16 and PIC18), 16 bits (PIC24, PIC30, PIC33) or 32 bits for the most recent and powerful series (PIC32). Such capacities may be compared to 1980's microprocessors (e.g., Intel 8080, Zilog Z80).

We focus on the PIC18 series, which provides the minimum performance required for us to work with. Without the use of any external component, this series provides in a chip at most 4 KiB of RAM, 128 KiB of program memory and the maximum CPU speed is 64 MHz. They are traditionally programmed using

Philippe Wang—This work started while the author was at LIP6-UPMC.

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 132–148, 2015.
DOI:10.1007/978-3-319-19686-2_10

low-level languages (e.g., assembler and subsets of C or Basic). PIC instruction sets are rather peculiar, making code generation challenging. For instance, there is no hardware implementation of the division operation on the PIC18 series and multiplication is absent for anterior series. Many compilers provide only minimal support for language constructions and standard libraries.

In this paper, we propose to use a modern high-level functional-based multi-paradigm general-purpose programming language, which notably provides automatic memory management, freeing the developer so they may focus on other tasks and is quite useful against segmentation faults, and strong static type checking, which intends to help catch errors of a certain class at compile-time instead of runtime. We chose OCaml, which is a particularly rich programming language, developed and distributed by Inria since 1990s. Of course, other languages could have been used instead, and the principles presented in this paper may be adapted or applied to many other languages.

To this end, we first developed a virtual machine (VM), largely based on the standard OCaml VM, and a relevant standard library. Then, to address the size of the generated binaries, we developed several tools to reduce their size, using different techniques: first we modified the bytecode format to reduce the size of binaries by about 75%; and then we developed a tool to remove unnecessary computations. In order to facilitate testing and debugging processes, we developed two simulators, one that simply uses the standard OCaml compiler and links to alternative libraries, and another that emulates the PIC18 environment. All those development tools have been tested by several implementations on actual hand-crafted hardware.

The rest of this paper is organized as follows. In the next section we briefly describe the two opposite worlds: the PIC microcontrollers families and the OCaml language. In section 3, we describe our implementation of a modified OCaml VM, a specific runtime library, and a modified standard library. Section 4 offers several ways to reduce the size of programs, in order to allow more programs to work on PIC. Then, section 5 presents two OCaPIC simulation tools that have been very useful during development. Section 6 presents some implementations that use OCaPIC, including a two-player board game and a programmable calculator. Finally, in section 7, we discuss various experiments on PIC programming with high-level languages and compare them to ours, on the execution speed and memory occupation, and conclude on our experience in section 8.

2 Two Worlds: PIC Microcontrollers and the OCaml Language

2.1 PIC Microcontrollers

The PIC microcontrollers (PICs) have small instruction sets (RISC) that are composed of arithmetic and logical instructions, branching instructions, and some special instructions, notably to access the flash memory containing the program.

PICs usually have four separate memory units. One is a nonvolatile memory using flash technology, called "program memory". Its size varies from hundreds of bytes to hundreds of KiB depending on the model. Our testing model, the PIC18F4620 contains 64 KiB of program memory (see Fig. 1). As its name suggests, it stores the program to execute and some constant data. Programming a PIC consists in writing the program into this memory. It is generally rewritable from 1,000 to 100,000 times depending on the model.

Model	Architecture	Flash memory	Registers	EEPROM	Speed	I/O pins
18F4620	8 bits	64 KiB	3968 B	1024 B	10 MIPS	36

Fig. 1. PIC18F4620 features

To dynamically store information, PICs only use tens to hundreds of thousands of General Purpose Registers (GPRs). They are accessible for reading and writing by the computing unit in two different ways: either directly by storing register indexes into instructions as usually, or indirectly to access registers of indexes that are computed dynamically with a complex indirection mechanism.

To configure a PIC (change the clock speed, lock/unlock flash accesses, etc.), communicate with internal hardware modules (timers, PWM modules, serial port encoders/decoders, etc.) and external modules (digitally or analogically) via the pins, a PIC uses about a hundred of Special Function Registers (SFRs). From the program, SFRs are mapped to reading and/or writting registers. Internally, the SFRs memory is electrically connected to the different hardware interfaces. Moreover some PICs have a small readable and writable EEPROM. (from some B to some KiB). These memory units are generally rewritable between 1,000,000 and 10,000,000 times.

In our implementation, the assembler code was written for the PIC18 series, and therefore concerns a bit more than 210 models of PIC currently in production. This series has the distinction of having an "extended instruction set", originally created to facilitate compilation from the C language. These instructions ease OCaml's stack management and significantly improve performance. Specifically, our tests were made on a PIC 18F4620. It was chosen primarily for its relatively large number of registers. The amount of dynamic memory is the factor which gives the greatest constraints in adapting OCaml programs for PIC.

2.2 The OCaml Language and Its Virtual Machine

The OCaml language is a high-level multi-paradigm programming language of the ML family. It implements functional, imperative, modular and object-oriented paradigms. Its main characteristics come from its type system providing a strong static type checking with type inference. The standard distribution comes with two compilers. The first one emits bytecode for its VM (also known as the ZAM)[6]. The second one emits native code. Both compilers share a runtime library implemented in C.

The OCaml VM, precisely described in [4], is a stack-based VM for a functional-based programming language with exceptions and objects. It uses

7 virtual registers: an accumulator to store one value (usable, for instance, as an operand of an arithmetic operation), a code pointer (containing the address of the next instruction to interpret), a stack pointer, a pointer for the closest exception handler, an extra-arguments counter (used for special function calls), an environment (pointing to the closure corresponding to the function that is currently executed) and a global data array (used to store static values such as static strings, and to make communications possible between modules). It is rather high-level, *e.g.*, it has a subset of instructions for a general apply mechanism (APPLY, APPTERM, RETURN, GRAB, RESTART). The instruction set contains 148 different instructions, but about 60% of which are shortcuts for several instructions combinations. No type verifications are made at runtime by the interpreter because the compiler guaranties that type checking is unnecessary at runtime [15].

3 The OCaml Virtual Machine on a PIC

The standard OCaml VM is implemented in C code. An obvious approach to consider is compiling this VM for the PIC18 architecture. However, the complete OCaml VM – that is the interpreter with the runtime library plus the bytecode loader – is about 22 000 lines of C code. The compiled runtime library alone is more than 250 KiB, which is more than four times the size of the available flash memory on most of PIC18 microcontrollers.

3.1 Bytecode Interpreter

The original interpreter component is about 1,000 lines of C code. One approach could be to provide a much simplified alternative runtime library (perhaps based on the work of Pagano et al.[13]) and an alternative bytecode loader for the VM to fit. Indeed, on a PIC, we do not need such a complex generational and incremental garbage collector (Stop&Copy+Mark&Sweep+Compact[3, Chapter9]), neither do we need to check the bytecode integrity. However, most C compilers provide poor performance when compiling for PICs, especially for VM implementations. PICOBIT uses a VM-specialized C compiler that was developed specifically for this use[16].

Instead of using C, we chose to develop an OCaml VM directly in PIC18 assembler. Our VM is consequently usable for all models of the PIC18 series (about 210 models), the portability of this solution is discussed in section 7.4. This choice allowed to save a lot of space. This constraint of space also drove us to choose to implement a 16-bit VM rather than a 32-bit one. On a microcontroller that has at most 4 KiB of RAM, a factor of 2 is quite important and having 32-bit words is not quite relevant as most of the time we would not need that much. Moreover, it remains possible to use larger integers if need be, as we do provide the Int32 and Int64 modules. Just like standard OCaml implementations, we use the least-significant bit to distinguish a pointer from an immediate value. Therefore, we provide 15-bit (signed) integers by default.

Another issue is floating-point numbers. Standard OCaml uses IEEE 754 double precision (64-bit) floats. On a PIC18, we probably do not want to have such precision because the performance would be poor as there is no hardware implementation for floating point numbers. As software floating point operations have a non negligible cost, we have decided to provide 32-bit floats with basic arithmetic operations. Going down to 16 bits does not feel relevant because of the very poor precision. All in all, we use runtime representations that are very similar to those of standard OCaml, except that we have to use less space.

	Assembler	OCaml	Binary Size
Bytecode interpreter	2,500 LoC	-	5,000 B
Runtime library (w/o GC)	200 LoC	-	400 B
Stop & Copy GC	150 LoC	-	250 B
Mark & Compact GC	550 LoC	-	1,000 B
Standard library	7,000 LoC	12,000 LoC	[variable]

Fig. 2. Sizes of implementations

3.2 OCaml Runtime Library Implementation

Several aspects of implementing a runtime library for an ML language are well known to be difficult. One of them is the garbage collector. Standard OCaml's garbage collector uses two heaps that we may call the "young" and the "old" heaps, and three collection algorithms [3, Chapter 9]. The minor collection implements a Stop&Copy algorithm to copy surviving values from the young heap to the old heap. The major collection implements a Mark&Sweep algorithm and a Compact algorithm.

On a microcontroller, we can ill afford such a complex garbage collector because it would use a large part of the available flash memory. Instead, we implemented two garbage collectors (also in PIC18 assembler) selectable at link time: a small and simple Stop&Copy, and a more complex and heavier Mark&Compact. Of course, the disadvantage of the Stop&Copy algorithm on a microcontroller is very clear: since we have so little memory, "wasting" half of it for a Stop&Copy algorithm seems unacceptable. However, Stop&Copy uses constant memory (just some bytes) for collection and has additional crucial advantages. Indeed, it is particularly easier both to implement and to debug than Mark&Sweep or Mark&Compact, and appears usable in practice. Also, the execution time for a collection is very low because of the size of the heap. On a PIC18 with 4 KiB of RAM, a collection never takes more than 7ms, and this is easy to determine because there is no issue related to the memory cache since there is none. Conversely, the Mark&Sweep implementation allow to use the entire available memory. However, the Mark&Sweep implementation is clearly slower (a bit more than 5 times when the heap is full) since it uses three passes on blocks to update pointers while the Stop&Copy needs only one pass.

Using a garbage collector raises another issue regarding hardware interruption management from OCaml. Indeed, an OCaml program can be interrupted

to call an arbitrary OCaml function only when the heap is in a consistent state. As for Unix systems, we define in assembler an interruption handler that raises a flag and stores the interruption metadata in a neutral space of the PIC memory, and when the VM knows that the heap is consistent, it checks this flag and calls the OCaml interruption handler if needed.

3.3 OCaml Standard Library Implementation

Most of the OCaml standard library is implemented in OCaml. The rest is implemented in C code and may be divided in two parts: the one that is relevant to embed on a microcontroller (*e.g.*, value comparison), and the one that is not (*e.g.*, Unix socket operations).

For the part that is tightly bound to the runtime library, such as value comparison, the external functions are implemented in PIC18 assembler. Such functions are invoked using the various CCALL instructions of the OCaml VM. Let us note that in the standard OCaml VM, the CCALL instructions are used to call C functions but in our case they are used to invoke assembler.

For the part that is not relevant on a microcontroller, it is simply removed. However, we do provide instead a microcontroller-specific library for input/output operations. Otherwise, we would never observe anything coming from the microcontroller. Therefore, just as the OCaml Unix library provides a typed interface to Unix operations, we have implemented a typed interface to microcontroller-specific input/output operations, which prevents many unsound operations thanks to ML's type system. And this may be particularly appreciated when debugging a program, as loading even a tiny program onto a PIC18 microcontroller may take tens of seconds. Also, let us note that there is in general no operating system on a PIC to prevent a wrong program from damaging the hardware. Using an ML-based type system prevents many wrong programs to make it to the hardware.

As such, a typed library is provided to manipulate PIC18 *special function registers*. Writing and reading those "registers" is the normal way to configure the hardware, to manipulate PIC's interfaces (*e.g.*, the serial interface), to catch interruptions, and to program unsafe I/O operations on the PIC microcontrollers. Instead of asking the programmers to provide integers to change the bits of those registers, we use sum algebraic data types to provide a statically well-typed interface. The standard library implementation (see Fig. 2) includes 490 specific lines to the PICs, and only 4 modules out of 35 have been modified in order to fit the PICs better.

4 Tools for Reducing Code Size

4.1 Elimination of Non-useful Bytecode: `ocamlclean`

Before having non-useful code elimination, our project could already run several small OCaml programs, including the board game described later, but could easily exhaust available memory, both RAM (because of unnecessary closures)

and flash. For instance, the object-oriented paradigm was unusable because it uses the module `CamlinternalOO`, which provides "run-time support for objects and classes"[7], and this module creates a lot of closures at runtime that could not be collected in time even when they were completely useless. Actually, any libraries with too numerous or too large modules were unusable as well because using one function of a module makes the whole module and its dependencies be loaded at runtime. For instance, using Core or Batteries libraries was out of question and even the standard OCaml library could easily make the bytecode too large. Those issues were solved by designing and implementing a tool to remove all, or almost all, unnecessary bytecode. We called this tool `ocamlclean`, and it can be applied to any standard OCaml bytecode binary (regardless of its relation to OCaPIC since it outputs standard OCaml bytecode binaries). Figure 3 details the workflow from an OCaml program to a PIC microcontroller and shows where `ocamlclean`'s action takes place.

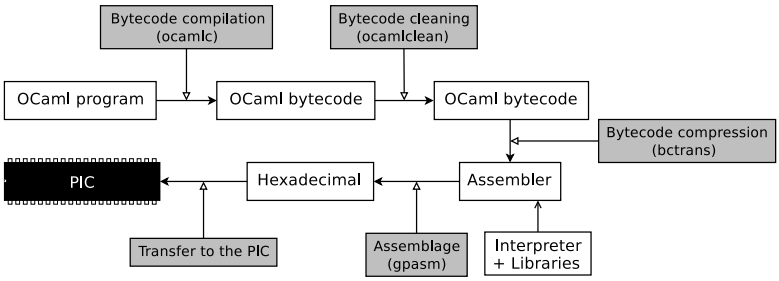


Fig. 3. From OCaml to PIC

The `ocamlclean` algorithm is not a simple reachability analysis because, in an OCaml bytecode program produced by `ocamlc`, all base blocks are reachable. The reason is that for all function codes, a closure creation is performed at the initialization step of the container module or inside an other function. To eliminate dead code in a functional language like OCaml, it is important to observe the dynamic representation of data that may point (directly or indirectly) to code sections. In practice, OCaml modules are implemented at runtime by blocks dynamically allocated in the heap. Modules usually contain functions stored in the heap as closures containing code pointers. A bytecode cleaner needs to compute which closures may be used at runtime and which may not. Code elimination is then performed in three steps, looping on them until a fixpoint is reached:

1. The first step consists in detecting blocks (corresponding to modules, in particular) in which we may eliminate some fields without changing the program behavior. To do this, `ocamlclean` starts with a data flow analysis based on an abstract interpretation of the program, and selects blocks on which it controls all accesses. It then cleans these blocks by removing unused cells allocations/initializations, and remapping usages.
2. By a second data flow analysis, `ocamlclean` computes for each code section the stack cells of the current stack frame that are written and never read.

It then selects closures that are no longer stored thanks to step 1 and removes their allocations.

3. At last, `ocamlclean` performs standard dead code elimination and removes code sections that are mentioned nowhere. Such code sections are left behind by the removal of some closure allocations at step 2.

Since code elimination potentially generates new block fields as candidates for step 1, `ocamlclean` loops on steps 1, 2 and 3 until no more dead code is eliminated. In practice, on our test programs, we need between 2 and 13 passes to reach the fixpoint. To emphasize the usability and effectiveness of `ocamlclean`, let us note that this tool has been shown useful in a large project where OCaml is used to program small and secure operating systems destined to the cloud[8].

4.2 Bytecode Format

The standard OCaml bytecode format is optimized for classical computers where memory alignment and simpler code clearly justify the loss of a few bytes. For instance, while the bytecode instructions can easily be represented by a single byte, since there are exactly 148 of them, each uses four bytes without counting their arguments. This means that each uses three non-significant zero-bytes. On a microcontroller, we can hardly afford to have so many non-significant zeros.

Our bytecode format takes back the non-significant zeros and an instruction is then represented with a single byte, plus zero, one or two bytes for its possible arguments. For instance, the instruction CONSTINT places its argument (which is an integer literal) in the accumulator register. Instead of taking eight bytes, in our bytecode format it takes three bytes, since we use one byte for the instruction code and two for a 16-bit integer.

Since arithmetic operations may have a significant cost on microcontrollers, integers are directly encoded with their memory representation, *i.e.*, integer n is directly encoded as $2n + 1$ so that this conversion does not have to occur at runtime. Indeed, we use a runtime representation very similar to the one used by standard OCaml.

Our bytecode format reduces the binary size by about 3.5 times, which has the additional advantage of speeding up reading of the bytecode on the microcontroller by the same factor. Another format modification is the pre-computation of the initial state of the VM. The standard OCaml VM deserializes the table of global data when it loads a program. Instead of doing such work, we pre-compute the states in which the stack and the heap would be in after global data loading, and we put it into the program memory. The VM would copy it verbatim into the RAM before starting interpreting the bytecode. This improves the performance such that the initialization may never take more than 2.5 milliseconds on a 4 KiB-RAM PIC18 microcontroller running at 10 MIPS.

5 Simulators

In order to make debugging a lot easier, we developed two simulators to run the programs on a basic personal computer. This is largely similar to what is done

with smartphone applications development where the software may be tested in a simulator prior to loading it on an actual phone.

Debugging activities involving a PIC microcontroller are difficult because of the limited interactions. There is not always an LCD, almost never a keyboard, and even if there were, when a program crashes, the PIC stops communicating. In such a case, it is not possible to investigate the state of the memory nor to stop the wrong program without powering the device off.

5.1 Runtime Simulator

When a program is compiled by OCaPIC, it is first compiled with the standard `ocamlc` compiler (see Fig. 3). The first simulation technique consists in linking the resulting binary with a simulator-specific runtime library, in which specific C functions replace the PIC18 assembler functions. This technique is particularly simple and the simulation runs very fast. This is useful mainly when searching for functional-related or algorithmic-related bugs.

However, this technique has a significant drawback. Some errors cannot be detected because it uses a standard VM, which works with 31-bit or 63-bit integers, rather than 15-bit integers. Also, there is a lot of memory on any modern personal computer compared to any PIC, that is why stack overflow and out of memory are not detected by this simulation technique.

```
./hw 'ocapic_lcd_simulator \
   16x2   \
   e=RD0  \
   rs=RD2 \
   rw=RD1 bus=PORTC'
```

Fig. 4. Screenshot of the simulated LCD display (2×16 chars)

5.2 Microcontroller Simulator

To address those issues, we implemented another simulator that directly interpretes the assembly code (which is normally transferred onto the PIC) according to the PIC specifications. This technique allows to test an OCaml program as if it were running on a PIC. This has been very useful in particular for the development of OCaPIC's version of the OCaml VM and its libraries, as they are implemented (partly or entirely) in PIC18 assembler.

5.3 Configuration Description

PIC simulators are numerous, they are generally used to analyze the state of a PIC's memory or to test a PIC on some external components (*e.g.*, LEDs, LCD, buttons). However, the code that is executed on a PIC is not the OCaml code of the programmer but the code of the specific OCaml VM. Few people are used to OCaml bytecode and to OCaml runtime data representation. Therefore, it

is not very useful to show them the state of the memory even if it is possible with our second simulator. However, this simulator can work with any simulated electronic circuit. To do that, the simulator interacts with a set of programs that are in charge of simulating the electronic circuit, and communicates with them via a textual protocol using standard file descriptors. OCaPIC provides a few simulated components, such as LEDs, push-buttons, switches, a keyboard, an LCD, etc. To illustrate the use of our simulators, we shall note that we wrote 108 lines of OCaml to transform an existing board game implementation (presented in the next section) into a game running on a modern personal computer with a graphical user interface, and this works for both simulators.

6 Hand Crafted Applications Using OCaPIC

This section relates our experience with the hand-crafted applications we have built over the past four years.

6.1 A Board Game

For testing and education purpose, we took an existing implementation in OCaml of a commercialized two-player board game (the standard version of *Gobblet*, edited by Gigamic and authored by T. Denoual). The initial version of the OCaml implementation was developed as a student project and used the Graphics module from the standard OCaml library. Our implementation is designed for a human player to play against the computer, and implements a classical min-max[12] algorithm. Porting this game for the PIC18 microcontroller was as simple as it could have been. The graphical user interface was replaced by a text-based interface displayed on a small LCD monitor (2 inches^2, for 10×100 pixels) to escort a board with a network of push-buttons (one for each square of the board). The min-max algorithm depth was adjusted so that it could run with the small amount of memory available on a PIC18F4620, *i.e.*, 4 KiB of RAM, which is actually the maximum for the PIC18 series. We did not need to adjust the code of the algorithm at all although it was initially designed to work on a modern personal computer.

The PIC18 implementation wins against beginner human players with only a few steps, and often (most of the time) beats experienced players when we inject a little randomization of the computer's decisions. Indeed, without randomization, we may detect a pattern to win against the computer since it would be completely deterministic and would play the same moves again and again. All in all, it results in a highly playable game, where the microcontroller computes a move in a few milliseconds at the beginning of a game, up to one or two seconds near the end of a long lasting game. The implementation is about 700 lines of OCaml (see Fig. 6). Implementing this game in another language (amongst those that are available for PIC18s) would have been a lot more difficult, notably because manually managing a PIC's memory is a challenge.

6.2 A Programmable-in-OCaml Calculator

As a two-student project, a board with a few buttons and a small 4×20-character LCD wired to a PIC18 microcontroller has been designed and implemented using OCaPIC. This board can be easily programmed in OCaml and can be used as a classical arithmetic calculator, but also as a lambda-term evaluator, or any application that may use the same hardware.

An interesting fact about this implementation is that the LCD was discovered to be soldered upside down to the board by the students. Using OCaml as the programming language allowed them to easily modify the OCaml code in order to software-rotate the display. We believe this small and complete implementation (see Fig. 6) could be used as a pedagogical base for teaching programming, or basic electronics, for instance in high schools.

Figure 5 describes main part of the electronic circuit of the calculator. Basically, the PIC receives inputs from the 20 push-buttons (on the left of the diagram) and displays expressions and results on a 4×20-character LCD display (on the right). The connection to the LCD display is very standard: data are transferred via the 8-wire bus in both directions between the port C of the PIC and the display, and connections E-D0, RS-D1 and RW-D2 are used for control. The E-D0 connection is the clock used to synchronize transfers on the bus, the PIC use the RS-D1 connection to tell if it transfer *data* (*i.e.*, characters to be displayed) or *instructions* (*e.g.*, SHIFT-LEFT, CLEAR-SCREEN, MOVE-CURSOR), and the RW-D2 connection is used to specify the direction of the transfer (from the PIC to the display or the reverse).

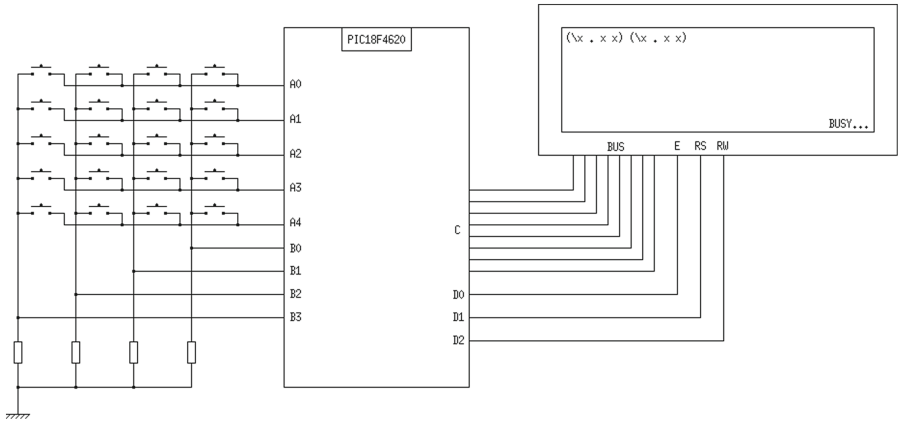


Fig. 5. Circuit of the programmable-in-OCaml calculator

The input is more interesting. It is important to know that, when using a microcontroller, we have no "high-level" protocol to connect a keyboard and receive data via an "abstract pipe". In practice, we have to explicitly send impulses and measure voltages on the pins to know which button is pushed

and which is not. That's the role of the small circuit described on the left of the schema. To test which button is pushed, the PIC repeatedly sends impulses on its outputs A0..A4, and measures voltages on its inputs B0..B3. For example, when it sends 5V on A3, it measures 5V on B0, it knows that the corresponding button is pushed. This kind of input circuit is interesting to reduce the number of pins used on the microcontroller, since to connect n×p buttons (here, $n = 4$ and $p = 5$), only $n + p$ pins are needed.

Obviously, the software management of this kind of input is complicated and, if not properly managed, tends to pollute the code. Indeed, during all the calculus performed by the PIC, the program has to regularly shift impulses sent on A0-A4, and slightly after each shift, measure voltages on B0-B3, compare them with the preceding measure, and in the case where the state of a button has changed, stop or trigger a corresponding action if necessary. Hardware interruptions are of course very useful to shift impulses and perform measures in parallel with the program execution, but interactions between the code of the interruptions and the rest of the algorithm remain a problem.

6.3 A Precise Heater

The propose of this heater is to precisely heat a recipient or a liquid to a temperature set by the user. For instance, such a device is very helpful to melt some chocolate at 45°C and keep it at this temperature for hours, which may be required in the pre-crystallization of chocolate, which is a process to obtain stable cocoa butter (*i.e.*, when the crystals of the cocoa butter are all in the stable beta form; in unstable forms, the cocoa butter will separate from the rest of the chocolate in matters of hours or less). Another example is that perfect soft-boild eggs may be obtained by leaving fresh eggs in some water maintained at 63°C for 90 minutes, as the yolk will not coagulate at this temperature but the white will; and as a bonus, maintaining eggs at such a temperature for such a long time eliminates the potential risks of salmonella.

This device consists in a circuit with a temperature probe, a heater and an LCD monitor with two push-buttons. The analog-digital converter of the microcontroller is used to measure the voltage given by the temperature probe. This implementation has the particularity of making intensive use of the floating-point library and of using the EEPROM to memorize the last measured temperature with a cache mechanism that protects the EEPROM from being written to often (otherwise the EEPROM could "fry"). The temperature stabilization algorithm is based on a software simulation of a PID (proportional-integral-derivative) controller. The source code of this application is about 250 lines of OCaml (see Fig. 6).

7 Discussion on Efficiency, Debugging and Portability

7.1 Execution Speed and Memory Occupation

On a PIC18F4620, the maximum execution speed of programs is 10,000,000 machine instructions per second, with an average close to 280,000 bytecode

instructions per second on standard examples. However, the number of bytecode instructions per second depends on the program, and is not significant when external routines (like `sleep`) are called. Anyway, we deduce that on our code examples, it takes an average of 36 machine instructions to execute a bytecode instruction. Note that a bytecode instruction makes much more work than a machine instruction, this factor is close to the comparison between the bytecode size and the size of the equivalent native code. We may also observe from our VM implementation that an average of 7.5 machine cycles are spent to handle a bytecode instruction, which represent only 21% of the run time.

There are 3,968 B of RAM on a PIC18F4620, shared by the execution stack, the heap, the VM registers (39 B), and the special function registers (128 B). The proportion between the size of the stack and the heap size is set at compile time. By default, the stack size was arbitrarily set to 172 levels, and if the Stop&Copy memory management algorithm is selected, the resulting heap size is 3,584/2=1,792 B, while with the Mark&Compact one, the heap size is 3,584 B.

The size of a PIC18F4620's program memory is 64 KiB. The size of the binary associated with the interpreter and runtime only depends on the GC algorithm used. It is about 5.5 KiB when the Stop&Copy is used and about 6.2 KiB when the Mark&Compact is used. The size of the binary associated with the assembly part depends on which routines are used. In practice, it is between 1 KiB and 12 KiB. Therefore, the maximum size of the bytecode that can be put in this PIC in the worst case is 48.5 KiB, which represents 76% of the total flash memory.

Project	Source	Interp. + Runtime	Native Lib.	Bytecode	Total
Gobblet	714 LoC	5.5 KiB	2.6 KiB	5.5 KiB	13.6 KiB
Calculator	1,704 LoC	5.5 KiB	3.7 KiB	6.2 KiB	15.4 KiB
Heater	238 LoC	5.5 KiB	9.0 KiB	12.9 KiB	27.4 KiB

Fig. 6. Program memory usage for some hand crafted applications

The figure 6 shows usages of the program memory for the different applications described in section 6. Due to the dead-code elimination of bytecode (thanks to ocamlclean), elimination of unused assembler routines (thanks to the assembler preprocessor), and heterogeneous usages of the standard library, no direct relations exist between the size of source codes and the size of the different parts of the produced binaries.

7.2 The VM Approach: Programming and Debugging

The main problem in running complex algorithms on microcontrollers is the deficiency of resources, and especially of volatile memory. The use of a garbage collector is then very useful to both facilitate code writing and memory occupation optimization. In that way, two algorithms have been implemented: a Stop&Copy and a Mark&Compact, both written in assembler code.

The OCaml VM is simple enough to be encoded directly in assembly, but provides high-level instructions that directly implement partial call, method access for object programming, exception management, etc. In addition to the OCaml

language constructions, it becomes possible to add other programming models like *constraints programming* (thanks to the Facile library), concurrent programming with a preemptive model (using VM-threads implemented using the hardware timers and interruptions offered by the PIC) or cooperative models (using reactive-ML or LWT-like approaches). The statically typed context help implement polymorphic data structures and generic libraries. Finally, let us note that programming environments like hardware and VM simulators allow to perform lots of verifications before transferring the code on the hardware. It is of course essential, before any test in real situation, to trace, profile and debug code in a comfortable environment. This VM approach is a good way to do that, it had also been tested (in other projects) to analyze code coverage by modifying the runtime environment without touching application codes [18].

7.3 Other Languages and Virtual Machines

The MPLAB development environment shipped with PIC microcontrollers offers an assembler (MPASM), a linker (MPLINK) and a simulator to test and debug an executable. It is possible to program using high-level languages, but we mainly found subsets of C and Basic (interfaced or not with MPLAB).

The other languages have three techniques to target PICs: native compilation (like C), interpretation (like Basic), compilation to a VM for bytecode interpretation as presented in this article.

For the native compilers, there are variations on the Pascal language such as Pic Micro Pascal[14] which is integrated with MPASM/MPLINK tools (from MPLAB) and offers its own IDE. Another language, between Basic and Pascal, Jal[1] proposed in 2004 a free compiler. A new version, Jal2, is supported by a still active community.

For interpreters, several implementations of the Forth language exist. Forth has the advantage of being small, and therefore can easily fit on a microcontroller. FlashForth[10] is a standalone Forth system implemented on PIC18F that provides an interpreter and a compiler/loader. There are also Forth compilers allowing to load their generated code as Rforth1[17]. The management of ports and interrupts is usually easy to manipulate in Forth.

We are not the first to use the VM approach to target the PIC architecture. Indeed, the design of a Scheme VM for PIC has already been undertaken within projects PICBIT[5] and PICOBIT[16] for a subset of R4RS. PICBIT adapted for the PIC18 series a very compact Scheme environment. The VM is implemented in C. PICOBIT extends this approach and attaches a C compiler, called SIXPIC specific to the PIC to obtain better code especially for the VM. As indicated during performance analysis, implementing an OCaml VM directly in PIC18 assembler, combined with code compaction techniques, gives very good performance. On this point, PICOBIT and OCaPIC approaches diverge, as the first focuses on the portability to other architectures while the second bets on efficiency.

We also find this VM approach for the Java language with Java Card Platform[11]. However, even by severely limiting the Java language

(*e.g.*, restrictions on automatic memory management, removal of 64-bit integers and threads) and by writing in an imperative style, we still get a too large code.

The Darjeeling project[2] is also an interesting port of the Java VM on AVR128 and MSP430 (a Texas Instruments family of microcontrollers). Like us, they compress the bytecode before its transfer to the microcontroller. An interest of this project is that they are able to compare performances of their VM approach to native code compilation. However, their number of AVR cycle per JVM instruction is close to 115, while with OCaPIC, the number of PIC cycle per OCaml VM instruction is close to 36. This difference may be explained by fundamental differences between architectures and bytecodes, and in particular complex JVM instructions like Java method calls.

7.4 Other Microcontrollers

Microcontroller families are numerous. We chose the PIC18 family because we had some experience with PIC16 before, and we did not chose the latter since it seemed too difficult to make OCaml work with even fewer resources. Notably, on series prior to the PIC18 series, the return stack for function calls is a specific stack, separated from the registers, and limited to 8 stages, at most. On PIC18, a set of specific instructions and registers were added to implement a call stack inside the PIC registers, and it is one of its most important "new" features because it makes compiling for PIC18 much easier than for any other previous series.

Porting our work from PIC18 to any higher PIC series would be relatively easy as the PIC24 and PIC32 assembly languages are basically supersets of the PIC18's. The main difficulty would be to benefit from the new instructions and hardware mechanisms to improve the VM and the runtime implementations. OCaPIC could inspire a similar implementation for architectures other than PICs, *e.g.*, ARMs (used on *STM-Nucleo* boards, for example), Atmel-AVR (used on the well known *Arduino* boards), etc. However, in a way, ARMs were fundamentally less in need of improvements of their programming environment than PICs, mainly because their instruction sets are more friendly for people writing compilers. Actually a native port is already supported by the Inria's distribution.

The Arduino platform is widely used by hobbyists, in particular in the robotics field. In this world where a kind of artificial intelligence is omnipresent, a higher level language than the usual C/C++ might be appreciated. The resources of Arduino are very similar to the PIC18s, both in memory and speed, and we presume that a similar approach would be relevant.

8 Conclusion

We have presented our port of the full OCaml language to program PIC18 microcontrollers, which have less than 4 KiB of RAM. To our knowledge, OCaml has become the richest language available to program this class of microcontrollers.

OCaPIC does not embed the richest library (yet), since there have been large libraries developed in C code over years and interfacing OCaml with C is not trivial (even in standard OCaml). For instance, Microchip provides a TCP/IP implementation for PIC18 in C code, which allows to host an HTTP server on a PIC18. However, we provide the possibility to interface OCaPIC to use external code just as we can interface standard OCaml with other languages using C code interface, except that we directly interface with assembler. As future work, we plan to provide more interfaced libraries, including USB features, trigonometry operations, external memory management, etc.

We started experimenting with Reactive ML[9] because it would be nice to program microcontrollers using this OCaml extension implementing the synchronous reactive model of Boussinot[1]. We managed to make it work on a PIC18 microcontroller after having defunctorized most functors used by Reactive ML. This was necessary because `ocamlclean` is currently not very efficient on functors, which consume too much memory.

Our project, OCaPIC, has been pretty stable for a while and is distributed as free open source software at http://www.algo-prog.info/ocapic.

References

1. van Ooijen et al, W.: Jal (not ?) Just Another Language, May 2004. http://jal.sourceforge.net/manual
2. Brouwers, N., Corke, P., Langendoen, K.: Darjeeling, a Java compatible virtual machine for wireless sensor networks. In: Proceedings of the ACM/IFIP/USENIX Middleware 2008 Conference Companion (2008)
3. Chailloux, E., Manoury, P., Pagano, B.: Developing Applications with Objective Caml. O'Reilly (2000). http://caml.inria.fr/pub/docs/oreilly-book/
4. Clerc, X.: Cadmium, February 2010. http://cadmium.x9c.fr/distrib/cadmium.pdf
5. Feeley, M., Dubé, D.: Picbit: a scheme system for the PIC microcontroller. In: Scheme and Functional Programming Workshop (SFPW 2003), pp. 7–15, November 2003
6. Leroy, X.: The ZINC experiment : an economical implementation of the ML language. Tech. Rep. RT-0117, INRIA, February 1990
7. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system (release 4.02): Documentation and user's manual. Inria, September 2014. http://caml.inria.fr/pub/docs/manual-ocaml/
8. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J.: Unikernels: library operating systems for the cloud. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS), pp. 461–472. ACM (2013)
9. Mandel, L., Pouzet, M.: ReactiveML, a reactive extension to ML. In: Proceedings of 7th International conference on Principles and Practice of Declarative Programming (PPDP 2005), Lisbon, Portugal, July 2005
10. Nordman, M.: Flashforth (2013). http://flashforth.sourceforge.net/

[1] Reactive Programming: www-sop.inria.fr/mimosa/rp/generalPresentation

11. Oracle: Java Card 3.0.4 Platform Specification. Oracle, September 2011. http://www.oracle.com/technetwork/java/javacard/specs-jsp-136430.html
12. Osborne, M., Rubinstein, A.: Course in Game Theory. MIT Press (1994)
13. Pagano, B., Andrieu, O., Moniot, T., Canou, B., Chailloux, E., Wang, P., Manoury, P., Colaço, J.L.: Experience report: using objective caml to develop safety-critical embedded tools in a certification framework. In: ICFP 2009: Proceedings of the 14th International Conference on Functional Programming, pp. 215–220. ACM (2009)
14. Paternotte, P.: Pic Micro Pascal V1.4: User Manual, July 2010. http://www.pmpcomp.fr
15. Pottier, F., Rémy, D.: Advanced Topics in Types and Programming Languages, chap. The Essence of ML Type Inference. MIT Press (2005)
16. St-Amour, V., Feeley, M.: PICOBIT: a compact scheme system for microcontrollers. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 1–17. Springer, Heidelberg (2010)
17. Tardieu, S.: A forth compiler for microchip pic 18fxxx (2011). http://www.rfc1149.net/devel/rforth1.html
18. Wang, P., Jonquet, A., Chailloux, E.: Non-intrusive structural coverage for objective caml. In: 5th Workshop on Bytecode Semantics, Verification, Analysis and Transformation, vol. 264 4 Electronic Notes in Theoretical Computer Science, pp. 59–73. Elsevier (2011). http://hal.archives-ouvertes.fr/hal-00497131/en/

Author Index