22nd Workshop on Logic-based Programming Environments (WLPE 2012)

Wim Vanhoof and Alicia Villanueva (editors)

Preface

These informal proceedings contain the papers presented at the 22nd Workshop on Logic-based Methods in Programming Environments (WLPE 2012), which was held in Budapest, Hungary, on September 8, 2012 as a satellite event of the 28th International Conference on Logic Programming, ICLP 2012.

WLPE is a series of workshops on practical logic-based software development methods and tools. Software plays a crucial role in modern society. While software keeps on growing in size and complexity, it is more than ever required to be delivered on time, free of error and meeting the most stringent efficiency requirements. Thus, more demands are placed on the software developer, and consequently, the need for methods and tools that support the programmer in every aspect of the software development process is widely recognized. Logic plays a fundamental role in analysis, verification and optimization in all programming languages, not only in those based directly on logic. The use of logic-based techniques in software development is a very active area in computing; emerging programming paradigms and growing complexity of the properties to be verified pose new challenges for the community, while emerging reasoning techniques can be exploited.

The 22nd Workshop on Logic-based methods in Programming Environments continues the series of successful workshops held in Ohio, USA (1989), Eilat, Israel (1990), Paris, France (1991), Washington D.C., USA (1992), Vancouver, Canada (1993), Santa Margherita Ligure, Italy (1994), Portland, USA (1995), Leuven, Belgium (1997), Las Cruces, USA (1999), Paphos, Cyprus (2001), Copenhagen, Denmark (2002), Mumbai, India (2003), Saint Malo, France (2004), Sitges (Barcelona), Spain (2005), Seattle, USA (2006), Porto, Portugal (2007), Udine, Italy (2008), Pasadena, USA (2009), Edinburgh, UK (2010), and Odense, Denmark (2011).

This year, there were 5 submissions, each of which was reviewed by at least 2 program committee members and the committee decided to accept 4 papers. On behalf of the Program Committee, we would like to thank the invited speaker, Manuel Carro, for kindly accepting our invitation to give a talk at the workshop. We would also like to thank the authors of submitted papers, the members of the Program Committee, and the external reviewers for their invaluable contribution. We are also grateful to Andrei Voronkov for making EasyChair available to us. Finally, WLPE 2012 is co-located with the International Conference on Logic Programming. We are grateful to the ICLP 2012 General Chair, Péter Szeredi, and the Workshop Chair, Mats Carlsson, for their invaluable support throughout the preparation and organization of the workshop.

July 2012,

Wim Vanhoof
Alicia Villanueva

Table of Contents

Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations 1
Dragan Ivanovic, Manuel Carro and Manuel Hermenegildo

Exploring file based databases via an Sqlite interface 2
Sander Canisius and Nicos Angelopoulos

A Framework for Verifying the Application-Level Race-Freeness of Concurrent Programs 10
Romain Demeyer and Wim Vanhoof

A Gradual Polymorphic Type System with Subtyping for Prolog 25
Spyros Hadjichristodoulou and David Warren

Syntactic integration of external languages in Prolog 40
Jan Wielemaker and Nicos Angelopoulos

Program Committee

Salvador Abreu	Universidade de Évora and CENTRIA
Petra Hofstedt	Brandenburg University of Technology Cottbus
Jacob Howe	City University
Yoshitaka Kameya	Tokyo Institute of Technology
Roland Kaminski	University of Potsdam
Lunjin Lu	Oakland University
Peter Schneider-Kamp	University of Southern Denmark
Alexander Serebrenik	Technische Universiteit Eindhoven
Zoltan Somogyi	The University of Melbourne
Wim Vanhoof	University of Namur
Alicia Villanueva	DSIC, Universidad Politecnica de Valencia
Damiano Zanardini	Universidad Politécnica de Madrid

Author Index

Angelopoulos, Nicos 2, 40

Canisius, Sander 2
Carro, Manuel 1

Demeyer, Romain 10

Hadjichristodoulou, Spyros 25
Hermenegildo, Manuel 1

Ivanovic, Dragan 1

Vanhoof, Wim 10

Warren, David 25
Wielemaker, Jan 40

Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations*

Dragan Ivanović,[1] Manuel Carro,[1,2] and Manuel Hermenegildo [1,2]

[1] School of Computer Science, Technical University of Madrid (UPM), Spain
[2] IMDEA Software Institute, Spain
idragan@clip.dia.fi.upm.es
{manuel.carro, manuel.hermenegildo}@{upm.es, imdea.org}

Service-Oriented Computing is an approach to creating applications where different loosely-coupled software services are composed in order to accomplish a goal that is more complex than what the constituent services can do. The composition is performed in a platform-independent manner (component services are executed in heterogeneous, usually non-local environments and accessed through a standardized interface) and is usually a long-running process which spans across organizations and administrative boundaries. In turn, these combinations are usually exposed as services themselves.

Service combinations are usually divided into orchestrations and choreographies. In the former case there is a single agent which controls the individual services and routes the data between them. In the latter, data movements and control are not centralized. In this talk we will focus on service orchestrations.

A critical point for the usability of service compositions is the Quality-of-Service (QoS) they offer. Execution time, availability, or monetary cost are some usual metrics. The acceptable values for QoS attributes in a business relation are usually defined in Service Level Agreements (SLAs), along with the penalties in case they are violated.

We present and evaluate a method whereby, using techniques from constraint logic programming, we derive, at a given point of execution of a service composition, a set of constraints that predict SLA conformance and violation scenarios over a certain time horizon. This is done on the basis of the structure of the composition and known or empirically measured properties of the component services. SLA failure and conformance constraints are expressed symbolically and may be used by other components for, e.g., development of data-mining models, optimized service matching, or triggering preventive adaptation or healing. Additional precision can be obtained within the same analysis framework by inspecting the state of the composition at the point of prediction.

References

1. D. Ivanović, M. Carro, and M. Hermenegildo. *Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations*. In Gerti Kappel, Hamid Motahari, and Zakaria Maamar, editors, *Service-Oriented Computing – ICSOC 2011*, number 7084 in LNCS, pages 62–76. Springer Verlag, December 2011. Best paper award.

* The authors were partially supported by the EC's 7th Framework Prog. Network of Excellence *S-Cube* (GA# 215483), Spanish MINECO project 2008-05624/TIN *DOVES*, and Community of Madrid project P2009/TIC/1465 *PROMETIDOS*.

Exploring file based databases via an SQLite interface

Sander Canisius, Nicos Angelopoulos, and Lodewyk Wessels

Netherlands Cancer Institute, Amsterdam, Netherlands
{s.canisius,n.angelopoulos}@nki.nl

Abstract. We present a succinct yet powerful interface library to the *SQLite* database system. The single file, serverless approach of *SQLite* along with the natural integration of relational data within Prolog, render the library a useful addition to the existing database libraries in modern open-source engines. We detail the architecture and predicates of the library and provide example deployment scenarios. Finally, this paper discusses the strengths of the system and highlights possible extensions.

1 Introduction

SQLite[1] [Allen and Owens, 2010] is a powerful, open source serverless database management system that requires no configuration as its databases are stored in a single file. Ran from a lightweight operating system (OS) library executable, it can be deployed in a number of scenarios where a traditional server-client database management system (DBMS) is not possible, advisable or necessary. In the context of Prolog, *SQLite* can provide a flexible transaction-based extension to the in-memory predicate storage. In this paper we present the implementation of a Prolog library that uses the C-interface to communicate with the *SQLite* OS library.

The relational nature of Prolog makes its co-habitation with relational database systems an attractive proposition. Not only databases can be viewed and used as external persistent storage devices that store large predicates that do not fit in memory, but it is also the case that Prolog is a natural choice when it comes to selecting an inference engine for database systems. As a result, there is an extensive body of literature on a variety of aspects of such integrations. We do not attempt a general overview of the area, but we present some of the research that is most clearly connected to our work. The *ODBC* library in *SWI-Prolog* [Wielemaker et al., 2012] is closely related to our work since we have used the library as a blue print both for the C-interface code and for the library's predicates naming and argument conventions.

The field of integrating relational databases has a long tradition going back to the early years of Prolog. For instance the pioneering work of Draxler [1991], although based on writing out *SQL* rather than directly interrogating the database, provided extensive support for translating combinations of arbitrary Prolog and table-associated predicates to optimised *SQL* queries. The code has reportedly, [Mungall, 2009], been ported to a number of Prolog systems, while it has also be extended and adopted for the *ODBC* library in the *Blipkit* software suite [Mungall, 2009]. Another approach which

[1] http://www.sqlite.org/

targeted machine learning and tabling as well as importing tables as predicates is MYDDAS, [Costa and Rocha, 2008].

Our current contribution does not deal with the theoretical or extensional aspects of integrating a database system to Prolog, in contrast, we concentrate on describing an open-source modern library that can be used out-of-the-box with a zero configuration, community supported database system. In doing so, we hope that the library will be a useful tool for the logic programming community and provide a solid basis in which researchers can contribute rather than having to reinvent the basic aspects of such integration.

The integration of flexible external database storage is crucial to the uptake of Prolog to new application and research areas which can assist with the up-keep of its development and its user penetration. Two such application areas are bioinformatics and web-servers. In the former, the ability to tap on to large data-sets is a central aspect due to the deluge of data publicly available resulting from the ever-growing number of high throughput technologies available to experimental biologists [for instance see Szklarczyk et al., 2011]. Our effort, along with other recent libraries, [Angelopoulos et al., 2012. In preparation.], takes a piece-meal approach into providing tools that can increase the penetration of Prolog into bioinformatics. These act complementary to more holistic approaches that provide a suite of programs within a single framework [Mungall, 2009]. In web-server applications, single file databases can facilitate scalable and effective interthread and back-end communication. Our library can enhance the excellent web facilities in modern open-source Prolog systems [Cabeza and Hermenegildo, 2001, Wielemaker et al., 2008].

The remainder of the paper is organised as follows. Section 2 describes the workings of the library. Section 3 presents some tests and possible applications, while the concluding remarks are in Section 4.

2 Library specifics

Here we present the overall architecture of the system along with the specific details of the three component architecture. Our library was developed on *SWI 6.1.4* under a *Linux* operating system. It is also expected to be working on the *Yap 6.3.2* by means of the *C*-interface emulation that has been also used in the porting other low-level libraries [Angelopoulos et al., 2012. In preparation.]. We publish [2] the library as open source and we encourage the porting to other Prolog engines as well as contributions from the logic programming community to its further development. Installation and running are extremely simple and only depend on the location of the OS *SQLite* library.

2.1 Overall architecture

Our library is composed of three main components. At the lower level, written in *C*, the part that handles opening, closing and communicating with the *SQLite* OS library. The *C* code is modelled after, and borrows crucial parts from the *ODBC* library of *SWI*.

[2] `http://bioinformatics.nki.nl/~nicos/sware`

predicate name and arity	moded arguments
sqlite_connect/2	+File, ?Conn
sqlite_connect/3	+File, ?Conn, +Opts
sqlite_disconnect/1	+Conn
sqlite_current_connection/1	-Conn
sqlite_default_connection/1	-Conn
sqlite_query/2	+SQL, -Row
sqlite_query/3	+Conn, +SQL, -Row
sqlite_format_query/3	+Conn, +SQL, -Row
sqlite_current_table/2	+Conn, -Row
sqlite_table_column/3	+Conn, ?Table, -Column
sqlite_table_count/3	+Conn, +Table, -Count

Table 1. Predicates table for Sqlite library. Top part: core functionality of the library, pertaining connection management and basic SQL query communication. Bottom section: auxiliary predicates on formatted querries and database introspection. There are no explicit predicates for importing tables, this is done wholesale for a database via a predicate option when a connection is initiated.

On top of the low-level interface, sit two layers that ease the communication with the database. On the one hand, a set of predicates allow the interrogation of the database dictionary, while the third layer allows the integration and interaction with tables as Prolog predicates.

2.2 Low-level interface

The heart of the library is its interface to *SQLite*. This is implemented in *C* and has strong affinity to the *ODBC* layer in *SWI*. The top part of Table 1 lists the interface predicates to the core system. The basic opening and closing operations are implemented as :

```
sqlite_connect(+File, ?Conn)

sqlite_disconnect(+Conn)
```

The *C* code creates a unique, opaque term to keep track of open connections. However, this is not particularly informative to the users/programmers. To circumvent this, the library allows for aliases to connections that can act as mnemonic handles through which connections are known to the user. To establish such an alias it is sufficient to give a non variable, usually atomic, value to the connection variable of *sqlite_connect/2*.

As a running example we will use the connection to a large but simple protein database from *Uniprot*. It has two tables referenced on a single key and each having $286,525$ entries. Table 2 summarises the basic parameters of the database. Except if otherwise mentioned, the examples in this papers assume that the database detailed in Table 2 is open by :

```
sqlite_connect('uniprot.sqlite', uniprot)
```

table name	population	columns
secondary_accessions	286525	secondary_accession, primary_accession
identifier_mapping	286525	uniprot_accession, identifier_type, target_identifier

Table 2. Structure of the database used as an example in the text.

To fine-tune details on the type of connection we wish to establish to the database file, we introduce an options argument in *sqlite_connect/3* . As is common practice, this is the last argument of the predicate. In addition to allowing a list of options from the set of possible options we also allowed single term options as a matter of convenience. The members of the recognised set are :

`alias(Alias)` An alternative way to register a connection alias. Note that in the case where `Conn` is also non-variable `Alias` and `Conn` should be unifiable.

`as_predicates(AsPred)` AsPred should be a Boolean value which when true instructs the library to create hook predicates that map each sqlite table to a Prolog predicate. These are created in the `AtMod` module, and it is the user's responsibility to make sure each predicate is unique in this module. See Section 2.4.

`at_module(AtMod)` Defines the module in which predicates should be defined when `AsPred` is `true`. By default, when this option is not present, predicates are defined in the `user` module.

`exists(Exists)` `Exists` should be a Boolean value. When this value is `false` the library does not throw an error if the provided file does not exist. The default value is `true` in which case an error is thrown if the *SQLite* file does not exist.

The user can interrogate all open connections and the existence of a specific connection via *sqlite_current_connection/1*. This predicate backtracks over all open connection if it is queried with a variable as its argument. The library also maintains the notion of a default connection which can be identified with *sqlite_default_connection/1*. This is usually set to the last connection opened and it is particularly useful in the common deployment scenario where a single connection is maintained.

The bulk of the traffic with *SQLite* is directed via *sqlite_query/2,3*. The version with arity 2 is simply a shortcut for applying the query on the default connection. The `SQL` argument of the predicates is an atom of valid *SQL* syntax, which when applied to the opened connection identified by `Conn`, results in rows that are returned one at the time in the form of a *row/n* term structure.

2.3 SQL wrapping predicates

We include in the library a small number of predicates that assist user interaction with databases. These include parametrised query strings, interrogating the database dictionary and simple aggregate operations.

sqlite_format_query(Conn,SqlFormat,Row) Post a format style *SQL* query to *SQLite* `Connection` and get row result in `Row`.

sqlite_current_table(Conn,Table) Enumerate all tables in the database or query whether a specific table is part of a database.

sqlite_table_column(Conn,Table,Column) This predicate holds for each triplet of `Conn`, `Table` and `Column` such that the named table has the respective field (`Column`) in the context of the given database.

sqlite_table_count(Conn,Table,Count) Count is the number of data rows in `Table`; which is a table present in the database identified by `Conn`.

2.4 Tables as predicates

With the *as_predicates/1* option of *sqlite_connect/3* we can direct the library to create linking predicates for each table in the database. That is a predicate is created for each table in the underlying database. The predicates are created at module identified by option *at_module/1*. It is the responsibility of the user to ensure there are no name clashes. Once thus declared, the table predicates behave as normal Prolog predicates. The system makes simple transformations when filling the predicates with results from the database. Currently, this is by mean of creating an *SQL* `SELECT` statements in which the `WHERE` subclause is formed from the ground arguments of the corresponding goal. For a table with name `Name` and columns that have a one-to-one correspondence with the list of variables in `Args`, the library constructs and asserts a clause in the Prolog database by using the following code :

```
Head =..[Name|Args],
Body = sqlite:sqlite_holds(Conn,Name,Arity,Columns,Args),
user:assert((Head :- Body)),
```

At runtime *sqlite_holds/5* ensures that the appropriate transformations take place and the results are fed back to Prolog as expected. By using predicated tables the number of rows for table *secondary_accessions/2* can be found with the following query :

```
?-
   Opt = as_predicates(true),
   sqlite_connect('uniprot.sqlite',uniprot,Opt),
   findall( A, secondary_accessions(A,_), As ),
   length( As, Len ).

As = ['A0A111', 'A0A112', 'A0A113', 'A0A131'|...]
Len = 286525.
```

Predicated tables only depend on *SQL* transformations and as such are not specific to *SQLite* but can be easily ported to other interface libraries such as *ODBC*.

3 Applications

3.1 Bioinformatics

The last decades have witnessed a phenomenal increase in the amount of biological knowledge that has been published and codified. This acceleration can be directly at-

tributed to the evolution of high throughput technologies such as genome wide expression assays, microscopy and deep sequencing.

One important way in which biological knowledge is codified is in the form of databases and ontologies. These include protein-protein interaction (PPI) databases such as STRING [Szklarczyk et al., 2011] and HPRD [Keshava Prasad et al., 2009] and protein information databases such as the universal protein resource [*Uniprot*, The UniProt Consortium, 2012]. STRING collates information about protein interactions from a variety of sources. It currently contains information on $5,214,234$ proteins from $1,133$ organisms and holds $224,346,017$ interactions. HPRD holds human proteins and interactions between them. Currently there are $39,194$ interactions in HPRD. The *Uniprot* database includes a wide-ranging array of information on proteins. In this paper we concentrated on the mapping abilities between naming conventions for proteins. In the *SQLite* file we created from a recently downloaded dataset from *Uniprot*there are $286,525$ gene identifier mappings.

Prolog is a powerful platform for bioinformatics research and analysis. Its ability to query relational datasets and express recursive searches succinctly are of particular interest to ontologies and databases tabulating millions of relations. One of the main roadblocks hindering the use of Prolog in this research area is the lack of effective tools that give access to the resources available. Databases form the basic layer of biological knowledge available. The use of effective tools to connect databases in efficient, resilient and integrative manners to the logic engine can assist in narrowing this gap.

3.2 Web-services

Engaging Prolog with the world wide web (WWW) in the role of a web-server has been well advocated and served by supporting libraries [Cabeza and Hermenegildo, 2001, Wielemaker et al., 2008]. Furthermore, there has been previous motivating work on systems that realise Prolog servers that mediate the web-publication of material stored in relational databases [Angelopoulos and Taylor, 2010].

The library presented here further facilitates the role of Prolog in this area. The main benefits of *SQLite* in this context are :

persistence - Prolog based servers need persistent storage of data. It is conceivable that such data can be realised as external files managed privately. However, the facilities offered by *DBMS* along with the presented predicated tables lead naturally to steadfast and scalable solutions.

threading - Web-servers are inherently multi-threaded and the ability to communicate through a shared file-based database provides further plurality to interthread communication methods.

ease of deployment - *SQLite* is arguably one of the easiest database back-ends to install and maintain. It depends on a single OS library and created databases are OS agnostic. In terms of the database it is thus trivial to migrate, move or upgrade the server.

filestore - One of the main success stories for *SQLite* has been in providing application specific filestore solutions. This fits well within a web-server setting where the server is provided with a clean back-end for all interactions with the filestore.

4 Conclusions

We presented a stable and efficient library for integrating a file-based DBMS to modern open-source Prolog engines. We have argued that Prolog is a powerful platform for data analysis and computational research in bioinformatics and for the realisation of agile web-servers that require minimal programming effort. Biological knowledge captured in the growing list of databases can be efficiently reasoned with, within logic programming. The library presented here facilitates such integrations in a straight forward, efficient and zero-configuration fashion that can facilitate rapid prototyping, collaborative coding and the development of reusable programs.

There are a number of possible extensions that can be envisaged on top of the presented library. These are not necessarily specific to this library but can also be of relevance to similar approaches such as the *ODBC* library of *SWI*. One possibility would be to have a more refined model for imported predicates which will not necessarily import all tables from a database. With regard to predicated tables, an integration with the work of Draxler [1991] would improve performance on queries that can be mapped to *SQL* join operations. Ideally, one would like this to be done behind the scenes, away from any user intervention. Finally, it might be possible to integrate the *ODBC* and *SQLite* libraries in one package within *SWI*, creating a powerful, common interface in which user programs are largely agnostic to the back-end database system used.

The underlying development principles of our approach follows other recent developments [Angelopoulos et al., 2012. In preparation.]. The common aspect is that although inspired from a specific application area, in both cases bioinformatics, the approaches produced stand-alone libraries that can be of use to a wider audience. This approach complements projects that take a more integrative approach to program development. In the case of bioinformatics such a suite is the *Blipkit* [Mungall, 2009].

5 Acknowledgements

We would like to thank Jan Wielemaker for discussions, feedback and for making the *ODBC* library of the *SWI-Prolog* open source, on which our library is based. Also, we are thankful to the anonymous reviewers who made valuable suggestions.

Bibliography

Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. Apress, 2nd edition, 2010. ISBN 1-4302-3225-0.

Nicos Angelopoulos and Paul Taylor. An extensible web interface for databases and its application to storing biochemical data. In *WLPE '10*, Edinburgh, Scotland, July 2010. URL `http://arxiv.org/pdf/1009.3771v1`.

Nicos Angelopoulos, Vitor Santos Costa, Joao Azevedo, Rui Camacho, and Lodewyk Wessels. Integrative statistics for logical reasoning, 2012. In preparation. URL `http://bioinformatics.nki.nl/~nicos/sware/real/`.

D. Cabeza and M. Hermenegildo. Distributed www programming using (ciao) prolog and the pillow library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.

J. Costa and R. Rocha. Global Storing Mechanisms for Tabled Evaluation. In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the 24th International Conference on Logic Programming, ICLP'2008*, number 5366 in LNCS, pages 708–712, Udine, Italy, December 2008. Springer-Verlag.

C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates*. PhD thesis, Zurich University, 1991.

T. S. Keshava Prasad, Renu Goel, Kumaran Kandasamy, Shivakumar Keerthikumar, Sameer Kumar, Suresh Mathivanan, Deepthi Telikicherla, Rajesh Raju, Beema Shafreen, Abhilash Venugopal, Lavanya Balakrishnan, Arivusudar Marimuthu, Sutopa Banerjee, Devi S. Somanathan, Aimy Sebastian, Sandhya Rani, Somak Ray, C. J. Harrys Kishore, Sashi Kanth, Mukhtar Ahmed, Manoj K. Kashyap, Riaz Mohmood, Y. L. Ramachandra, V. Krishna, B. Abdul Rahiman, Sujatha Mohan, Prathibha Ranganathan, Subhashri Ramabadran, Raghothama Chaerkady, and Akhilesh Pandey. Human protein reference database2009 update. *Nucleic Acids Research*, 37(suppl 1):D767–D772, 2009.

Chris Mungall. Experiences using logic programming in bioinformatics. In Patricia Hill and David Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009*, volume 5649 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2009.

Damian Szklarczyk, Andrea Franceschini, Michael Kuhn, Milan Simonovic, Alexander Roth, Pablo Minguez, Tobias Doerks, Manuel Stark, Jean Muller, Peer Bork, Lars J. Jensen, and Christian von Mering. The string database in 2011: functional interaction networks of proteins, globally integrated and scored. *Nucleic Acids Research*, 39(suppl 1):D561–D568, 2011. doi: 10.1093/nar/gkq973. URL `http://nar.oxfordjournals.org/content/39/suppl_1/D561.abstract`.

The UniProt Consortium. Reorganizing the protein space at the universal protein resource (uniprot). *Nucleic Acids Res*, 40:D71–D75, 2012.

Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. Swi-prolog and the web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. ISSN 1471-0684.

A Framework for Verifying the Application-Level Race-Freeness of Concurrent Programs

Romain Demeyer and Wim Vanhoof

University of Namur, Belgium
{rde,wva}@info.fundp.ac.be

Abstract. Even if concurrent programming becomes more prevalent in software development, it still faces the harsh reality that writing correct and efficient programs is hard and error-prone. While existing static and dynamic analyses are capable of detecting low-level race conditions, i.e. potential concurrent updates of some shared variable, programs can still exhibit errors due to the presence of so-called *application-level race conditions*, when concurrent execution may violate the design-level consistency of the shared variables. In this paper, we present a framework for statically detecting application-level race conditions. Based on an easy-to-write specification, we use a combination of symbolic execution and equational reasoning to verify that each atomic section of the program preserves the consistency of the shared variables with respect to the given specification, as such guaranteeing that no race conditions exist that could result in violations of this specification.

Keywords: concurrent programming, static analysis, race detection

1 Introduction

In recent years, hardware manufacturers have started to focus on further developing parallel architectures due to the problems encountered with increasing clock speeds and straight-line instruction throughput [24]. This shift in hardware evolution is provoking a fundamental and necessary turn towards concurrency in software development [24]. Unfortunately, developing correct and efficient concurrent programs is hard, as the underlying programming model is more complex than it is for sequential programs and errors are often introduced due to unexpected interference between threads [21]. To protect parts of the code from the interference of other threads, programmers define so-called *atomic sections*. These are code fragments whose execution and the data they handle must not be affected by the operations executed by other threads. To allow the programmer to define such atomic sections, concurrent languages offer synchronization primitives such as locks [17] or, more recently, software transactional memories (STM) [11, 20].

The ability to define atomic sections, by locks or some other mechanism, is in itself not sufficient to develop correct concurrent programs, as the programmer still has to decide what parts of the code must be protected, which is far from

trivial in practice [3]. Having too fine-grained sections increases the risk of errors while having too coarse-grained sections is inefficient as the parallelism is not exploited optimally.

Let us first illustrate what kind of behavior can happen when atomic sections are not correctly placed in a program and, thus, when race conditions can occur. In this paper, we focus on a Haskell-like language [26], recognized to be particularly well-suited for concurrent programming, notably because, while the way it allows to define concurrent programs is very similar to standard imperative languages, it proposes a more explicit distinction between pure computations and those that may have side effects [7, 22]. Although different definitions exist of what is a race condition, it is most frequently regarded as occurring "when two threads can access a data variable simultaneously, and at least one of two accesses is a write"[9]. We believe that this definition is limited and we denote this kind of race condition by *low-level races*. Consider for example the source code of Fig. 1, where the `do` notation refers to the classic syntactic sugar to express monadic computations [8, 10] and `readTVar`, resp. `writeTVar`, allows to read a value from, resp. write a value in, a shared variable. The function `add` inserts a new value `n` into a list of integers (the shared variable `shTab`), and adds this very value to the value of another shared variable `shSum`. Each update of a shared variable is protected by an atomic section – i.e. the operation `atomically` – and, consequently, the program is free of low-level race conditions. This is both sufficient and efficient, as long as the values of the shared variables are independent. However, if there is an implicit link between the values of the shared variables, the story is more subtle. Suppose that in the example, `shSum` is meant to represent the sum of the integers in the list `shTab`. In this case, an inconsistent state (in which the value of `shTab` has been updated while the value of `shSum` has not) is exposed *between* the two atomic sections, which is problematic in a concurrent program. Suppose for example that at this precise point, another thread doubles the value of `shSum` and of each integer of `shTab` (the whole operation being protected inside an atomic section). At the end of the execution of both threads, the sum of the integers in the list `shTab` and the value of `shSum` will be different, breaking the programmer's intention and thus showing an unacceptable error in the program. This situation, which can of course be corrected by using a single atomic section in the `add` function for both updates of the shared variables, is what is sometimes called an *application-level race condition* [3], as it represents an inconsistency with respect to the logic of the application that cannot be observed from the source code alone. Like any

```
add shTab shSum n =
 do atomically ( do tab <- readTVar shTab
                    writeTVar shTab (n:tab) )
    atomically ( do s <- readTVar shSum
                    writeTVar shSum (s+n) )
```

Fig. 1. Example of a program with atomic sections

race, application-level race conditions are hard to prevent with testing, as their occurrence depends on the interleaving of threads. While application-level race-conditions are common in large programs [13], classic race detectors or race-free type systems are unable to detect them [5, 17] as these analyzes do not have or do not use information about the intended application logic.

In this paper, we present a novel and generic approach for static analysis targeting the detection of application-level race conditions. Our specific contributions are the following:

- We propose a simple but powerful method to formalize what we call the *consistency* of the shared variables, i.e. the application-level properties and dependencies that the programmer wants to see preserved during the program's execution, in the form of a set of *invariants*.
- We propose to statically compute a safe characterization of how the shared variables change during the execution of an atomic section. For this purpose, we formalize a *symbolic evaluation* for a core subset of STM Haskell [7]. Note that the process we describe could easily be adapted for a standard imperative language.
- We show how a theorem prover can be used to prove, from the derived characterizations and some annotations, that an atomic section preserves the consistency of the shared variables, thereby allowing to prove that the program is application-level race-free with respect to the given specification.

The remainder of this paper is structured as follows: In Section 2, we present an intuitive overview of our analysis by using a running example. In Section 3, we formalize the static analysis for a subset of STM Haskell. Finally, we discuss in Section 4 the related work and the future prospects.

2 Defining and Verifying the Consistency of Shared Variables: an Overview

The key idea behind our analysis consists in verifying, for each atomic section in the program under consideration, that *if* the shared variables are in a consistent state before entering an atomic section, they will again be in a consistent state upon exit of the atomic section. In this section, we introduce the details of this approach using the example of Section 1 as running example.

2.1 Consistency Definition

An *application-level race condition*, which we simply call *race* for the remainder of the text, occurs when shared variables are exposed – i.e. unprotected by an atomic section – in a state that is considered by the programmer as inconsistent with respect to the design of his or her application. As this is related to the application logic, our analysis requires to provide information about what one considers as a consistent state. Back to our example of Section 1, we need to express the programmer's intention with respect to the shared variables, namely

```
add shTab shSum n = atomically ( do                  (S0)
                      tab <- readTVar shTab          (S1)
                      writeTVar shTab (n:tab)        (S2)
                      s <- readTVar shSum            (S3)
                      writeTVar shSum (s+n) )        (S)
```

Fig. 2. The corrected version of the running example

that `shSum` is supposed to store the sum of the integers in the list `shTab`. We can formalize this with the following equation:

$$sum(val(\texttt{shTab})) = val(\texttt{shSum}) \tag{1}$$

The identifier *sum* is just a name that we invent to represent the sum of the elements in a list; it is *not* a function defined in the program, only a name. We will use this kind of names to represent what we will call *characteristics*, i.e. those properties of the values stored in the shared variables that are important in defining their consistency. Intuitively, the equation above denotes the fact that the sum of the elements of the list stored in the shared variable `shTab` must equal the value stored in the shared variable `shSum`. The identifier *val* is a predefined identifier representing the value of a shared variable. An equation such as the one above is what we call an *invariant*. The programmer can define as many characteristics and invariants as he or she needs for formalizing the consistency of the shared variables. For example, if the list stored in `shSum` was to stay sorted, a second equation of the form $sorted(val(\texttt{shTab})) = \texttt{True}$ could be added to the invariants.

2.2 Analysis

In order to illustrate the key ideas of our analysis, let us consider the corrected version of our running example, depicted in Fig. 2. The correction consists, of course, in integrating both atomic sections into a single one such that *both* shared variables are updated in a single transaction.

The main idea behind the analysis is simple: what can we derive about the characteristics of the shared variables that hold at the end of the atomic section, when assuming that the invariants hold at the start of the atomic section. The information we derive is represented by a set of equations. When analyzing the single atomic section of the example, we proceed thus as follows:

- At the beginning of the atomic section, we suppose (**assumption**) that the invariant (equation (1)) holds, so we start with a set of equations `S0` containing only the invariant:

$$\texttt{S0} = \{sum(val(\texttt{shTab})) = val(\texttt{shSum})\}$$

- Subsequently, we analyze each action within the atomic section in turn, and compute how the action changes the set of equations that were valid before the action. In our example, the first action links the value stored in `shTab` with the lambda variable `tab`, so the analysis adds the equation $val(\texttt{shTab}) = \texttt{tab}$ to `S0`. Moreover, it replaces $val(\texttt{shTab})$ by `tab` in the other equation(s). The reason is that the value stored in `shTab` can change, while the lambda variable `tab` is by definition immutable. This gives us the set `S1` representing the equations that hold after the first action.

$$\texttt{S1} = \{sum(\texttt{tab}) = val(\texttt{shSum}), val(\texttt{shTab}) = \texttt{tab}\}$$

- The second action consists by writing a new value into `shTab`. So, our analysis replaces the second equation of `S1` by $val(\texttt{shTab}) = \texttt{n:tab}$. Note that we can keep the first equation in the set as it does not contain any reference to `shTab` any-more, and we obtain

$$\texttt{S2} = \{sum(\texttt{tab}) = val(\texttt{shSum}), val(\texttt{shTab}) = \texttt{n:tab}\}$$

- Then, the value stored in `shSum` is linked with `s`. We add the equation $val(\texttt{shSum}) = \texttt{s}$ and replace $val(\texttt{shSum})$ by `s` in the other equations, obtaining as such

$$\texttt{S3} = \{sum(\texttt{tab}) = \texttt{s}, val(\texttt{shTab}) = \texttt{n:tab}, val(\texttt{shSum}) = \texttt{s}\}$$

- Finally, a new value is stored in `shSum`, so the analysis replaces the last equation by $val(\texttt{shSum}) = \texttt{s+n}$ and we obtain

$$\texttt{S} = \{sum(\texttt{tab}) = \texttt{s}, val(\texttt{shTab}) = \texttt{n:tab}, val(\texttt{shSum}) = \texttt{s+n}\}$$

Note that this set `S` represent the equations characterizing the shared variables that hold at the end of the atomic section, i.e. it represents the state of the shared variables as it will be exposed to other threads. Consequently, in order to prove that the analyzed piece of code does not contain an application-level race condition with respect to the provided specification, it must be proved that the invariant(s) hold in this final state `S`. How this proof can be made is the subject of the following subsection.

2.3 Proof

In order to prove that the set of equations `S` that hold at the end of the atomic section implies the invariant (equation (1)), we will have to define a number of *axioms* that relate how the characteristics that are used in the invariants evolve by the functions used within the atomic section. In case of our example, the following axiom will be sufficient to complete the proof:

$$\forall \texttt{x,xs} : sum(\texttt{x:xs}) = sum(\texttt{xs})\texttt{+x} \tag{2}$$

This axiom states that the sum of the list obtained by inserting any integer `x` into any list `xs` is equal to the sum of the list `xs` and `x`. In our analysis, axioms can be seen, intuitively, as a way to define the characteristics *with respect to the functions of the program.* Axioms, as well as invariants, are easy to write for

a programmer who is aware of the design requirements and the specifications of its program. But with these apparently lightweight annotations, a simple **equational reasoning** is sufficient to prove that the atomic section is race-free. Returning to our example, using the equations in `S` and the axiom (2) we can rewrite the left-hand side of the invariant (1) as follows:

$$\begin{aligned} sum(val(\texttt{shTab})) &\overset{\texttt{S}}{=} sum(\texttt{n:tab}) \\ &\overset{(2)}{=} sum(\texttt{tab})\texttt{+n} \\ &\overset{\texttt{S}}{=} \texttt{s+n} \\ &\overset{\texttt{S}}{=} val(\texttt{shSum}) \end{aligned}$$

obtaining as such the right-hand side of the invariant, thereby proving that the invariant is implied by the set `S` and the axiom and, thus, that the invariant is preserved by the atomic section. The proof of the invariants is automated in an efficient way by using a theorem prover for first-order logic such as Vampire [18], the E Theorem Prover [19] or iProver-Eq [12]. For this purpose, the abstract states generated by our analysis and the axioms provided by the user must be translated in the TPTP syntax [23].

In conclusion of this section sketching the workings of our analysis, let us note the following:

- The specification required for the proof is lightweight. In case of the example, a single invariant and a single axiom are sufficient to prove that the code is application-level race-free.
- The type system of STM Haskell enforces that each access to a shared variable occurs inside such an atomic section, hence, to conclude race-freeness of a complete program it is sufficient to repeat our procedure for each atomic section present in the program.
- Returning to the initial erroneous code of Section 1, it can be seen that the analysis would be unable to prove that the invariant holds at the end of the first atomic section (same equations as `S2`) and hence would conclude that that code is not guaranteed to be application-level race-free.

3 Static Analysis

In this section, we define formally a static analysis for a STM functional language $\mathcal{H}$ that can be seen as a subset of STM Haskell. We first present the syntax and the semantics of the language $\mathcal{H}$, which are quite similar to the languages used in [7, 4].

3.1 The language $\mathcal{H}$

The language $\mathcal{H}$ is a subset of STM Haskell [7] and other similar languages [4, 14]. A program is a set of function definitions of the form $f \hookrightarrow M$ where f is the (unique) name of a function and M is an expression. The syntax of expressions is

Expression		Meaning	Class
x	$(x \in \mathbf{X})$	variable	value
a	$(a \in \mathbf{N})$	address	value
$\lambda x.M$		lambda abstraction	value
$c\ \overline{M}$	$(c \in \mathbf{C})$	construction	value
$M\,N$		application	pure expression
f		function call	pure expression
`case` M `of` $\overline{c\overline{x} \to N}$		case expression	pure expression
`readTVar` M		STM read variable	STM operation
`writeTVar` M N		STM write variable	STM operation
M`>>=`N		bind	STM/IO operation
`return` M		return	STM/IO operation
`atomically` M		IO execute transaction	IO operation
$M \mid N$		thread composition	IO operation

Fig. 3. Syntax of $\mathcal{H}$ expressions

given in Fig. 3. We assume denumerable distinct sets **X** and **N** of, respectively, *pure* variables that are local to lambda expressions, and *transactional* variables, i.e. shared variables, that are global to the program and that can be referenced by any expression. The set **C** represents the set of data type constructors used by the program. In the definition we use M and N to refer to expressions, $\overline{M}$ to refer to possibly empty sequences of expressions $\langle M_1, \ldots, M_n \rangle$ and $\overline{x}$ to refer to possibly empty sequences of different variables $\langle x_1, \ldots, x_n \rangle$. In the same way, we use $\overline{c\overline{x} \to N}$ to represent a non-empty sequence of case expressions $c_1\overline{x_1} \to N_1, \ldots, c_n\overline{x_n} \to N_n$, where the constructors c_i are all different. The set of all expressions is denoted by **M**.

For the remainder of the text, we suppose that the expressions are well-formed. In particular, STM operations `readTVar` and `writeTVar` can only be used inside atomic sections, which are delimited by the IO operation `atomically`, and thread composition only admits expressions which are composed of IO operations. The operations `return` and `>>=` are overloaded as they can be used to return a value, respectively to compose sequentially operations, as well in the IO class as in the STM class. Moreover, we suppose that every program has a main function, which is composed of IO operations. A complete type system for $\mathcal{H}$ can easily be defined [4].

We define a reduction relation [15] of the form $(M, \sigma) \to (M', \sigma')$ to specify the behavior of $\mathcal{H}$ programs. The environment σ comprises a mapping from transactional variables to values. The reduction rules CALL and APP, where we denote the capture-avoiding substitution of N for each free occurrence of x in M by $M\{x/N\}$, as well as CASE and BIND are standard rules for functional languages. Note that these reductions have no side-effect in the sense that σ is not modified. The rule READ, respectively WRITE, defines the reading from, respectively writing into, a transactional variable. The rule ATOM shows that a (transactional) expression M is reduced in one step using the operation

$$\text{CALL}\ \frac{f \hookrightarrow M \in P}{(f,\sigma) \to (M,\sigma)} \qquad \text{APP}\ \frac{}{((\lambda x.M)\ N,\sigma) \to (M\{x/N\},\sigma)}$$

$$\text{CASE}\ \frac{}{(\texttt{case}\ c_j\,\overline{M}\ \texttt{of}\ \overline{c\overline{x} \to N}\ ,\sigma) \to (N_j\{\overline{x_j}/\overline{M}\},\sigma)} \qquad \text{BIND}\ \frac{}{(\texttt{return}\ M\texttt{>>=}N,\sigma) \to (N\ M,\sigma)}$$

$$\text{READ}\ \frac{}{(\texttt{readTVar}\ a,\sigma[a \mapsto N]) \to (\texttt{return}\ N,\sigma[a \mapsto N])}$$

$$\text{WRITE}\ \frac{}{(\texttt{writeTVar}\ a\ M,\sigma[a \mapsto M']) \to (\texttt{return}\ (),\sigma[a \mapsto M])}$$

$$\text{ATOM}\ \frac{(M,\sigma) \to^* (\texttt{return}\ M',\sigma')}{(\texttt{atomically}\ M,\sigma) \to (\texttt{return}\ M',\sigma')} \qquad \text{CTX}\ \frac{(M,\sigma) \to (M',\sigma')}{(\mathbb{C}[M],\sigma) \to (\mathbb{C}[M'],\sigma')}$$

Fig. 4. Semantics of $\mathcal{H}$

`atomically`. Finally, the rule CTX shows that a reduction can be proceeded in an evaluation context [15] $\mathbb{C}$ which identifies the active site [.] for the reduction [7]. Formally, we define an evaluation context as follows :

$\mathbb{C} ::=$
$[.] \mid (\mathbb{C} \mid M) \mid (M \mid \mathbb{C}) \mid (\mathbb{C}\ \texttt{>>=}\ M) \mid \mathbb{C}\ N \mid \texttt{case}\ \mathbb{C}\ \texttt{of}\ \overline{f\overline{x} \to N}$
$\mid \texttt{readTVar}\ \mathbb{C} \mid \texttt{writeTVar}\ \mathbb{C}\ M\}$

and use $\mathbb{C}[M]$ to denote the selection or substitution of the evaluation site [.] in a context $\mathbb{C}$ by an expression M. Intuitively, the fact that, in case of thread composition, an expression can be seen as an instance of two contexts depicts the fact that an arbitrary thread is selected for the next transition.

As we consider well-formed, well-typed programs, these reduction rules allow different threads to interleave IO operations (as does the execution of complete atomic sections) but does not allow the *content* of two atomic sections to be interleaved.

3.2 Static Generation of Abstract States

Our analysis supposes that we have a set of characteristics **P**. As said before, a characteristic is an identifier defined by the programmer. We introduce the notion of *terms* with respect to **P**, which is a global parameter for the analysis of a program, in Definition 1.

Definition 1. *The set of* terms *is defined as follows:*

$T ::= M$		*expression*
$\mid val(a)$	*reference to the value (content) of a shared variable*	
$\mid p(T)$	$(p \in \boldsymbol{P})$	*characteristic of a term*

Basically, our analysis manipulates an abstract state $\mathbb{S}$, which is defined as follows:

Definition 2. *An abstract state* $\mathbb{S}$ *is a set of couples of the form* $(E, \overline{T})$ *such that*

- E *is a set of* equations *of terms, where an equation between two terms* T_1 *and* T_2 *is denoted by* $T_1 = T_2$,

– $\overline{T}$ *is a possibly empty sequence of terms.*

Intuitively, at a program point, the abstract state $\mathbb{S}$ contains relevant information about characteristics and shared variables that can hold at that point; each couple $(E, \overline{T})$ in $\mathbb{S}$ contains the information collected over a possible path in the source code up to that program point. In such a couple $(E, \overline{T})$, the sequence of terms $\overline{T}$ must be seen as a stack where the terms wait for an another term to form an equation to be added in E.

The analysis of an expression (and of a program) is defined with respect to a particular set of equations, denoted by I and that represents the set of *invariants*. Intuitively, this set contains the equations, defined by the programmer, such that he or she expects that they are preserved by every atomic section.

Moreover, we are interested in collecting the abstract state at particular programs points such as at the end of atomic sections, in order to check whether the invariants hold at that particular point. To that end, we suppose that all occurrences of `atomically` and `>>=` in the program are labeled by unique labels from, respectively, $\mathcal{L}_a$ and $\mathcal{L}_b$, where $\mathcal{L}_a$ and $\mathcal{L}_b$ are two sets of distinct labels, i.e. $\mathcal{L}_a \cap \mathcal{L}_b = \emptyset$. To keep a trace of the abstract states at these specific points of the program, we use a mapping ρ from labels to abstract states. We denote by $dom(\rho)$ the domain of the mapping (or *trace*) ρ. Intuitively, for such a trace ρ and a label $l \in dom(\rho) \cap \mathcal{L}_a$, the abstract state $\rho(l)$ represents the abstract state at the end of the atomic section labeled l and, likewise, for a label $l \in dom(\rho) \cap \mathcal{L}_b$, $\rho(l)$ represents the abstract state between the two operations bound at l. We denote by $\rho[l \mapsto \mathbb{S}]$ the trace ρ' such that $\rho'(l) = \mathbb{S}$ and $\forall l' : l' \in dom(\rho) \wedge l' \neq l : \rho'(l') = \rho(l')$.

Using our analysis, we will compute an environment ϕ which is defined below

Definition 3. *An environment ϕ for a program P is a set of tuples of the form $(f, \mathbb{S}_1, \mathbb{S}_2, \rho)$ where f is a function of the program P, $\mathbb{S}_1$ and $\mathbb{S}_2$ are abstract states and ρ a trace (also called* trace*) with $dom(\rho)$ the labels from f.*

Intuitively, The meaning of a tuple of the form $(f, \mathbb{S}_1, \mathbb{S}_2, \rho)$ is that the abstract state $\mathbb{S}_2$ and ρ represents the results of analyzing the code of the function f with respect to an abstract state $\mathbb{S}_1$.

The analysis of an expression M with respect to an abstract state $\mathbb{S}$, an environment ϕ, a trace ρ, both representing the result of analysis so far, and a set of invariants I is denoted by $\mathcal{A}[[M]]^{\mathbb{S}}_{\phi,\rho}$ (we omit I as it can be seen as a global parameter of the analysis of a program) and defined in Figure 5. The effect of analyzing an expression consists basically in computing an update of the abstract state, environment, and trace and, as such, the result of analysis is represented by a triple, i.e. $\mathcal{A}[[M]]^{\mathbb{S}}_{\phi,\rho} = (\mathbb{S}', \phi', \rho')$. In the definition of $\mathcal{A}$, we use the following notations and concepts. If $\overline{T}$ is the sequence $\langle T_1, ..., T_n \rangle$, we denote by $T : \overline{T}$ the new sequence of terms $\langle T, T_1, ..., T_n \rangle$. We say that T' is a subterm of T if $T = T'$ or if $T = p(T'')$ and T' is a subterm of T'' for some characteristic $p \in \mathbf{P}$. We denote by E_{T_1/T_2} the set of equations obtained by replacing every occurrence of T_1 by T_2 in the subterms of every equation of E and we denote by $remove(E, T)$ the set of equations obtained by removing

$\mathcal{A}[[\texttt{atomically}_{l_a}\ M]]^{\mathbb{S}}_{\phi,\rho} = (\mathbb{S}, \phi', \rho'[l_a \mapsto \mathbb{S}''])$
where

- $\mathbb{S}' = \{(I \cup E, \overline{T}) \mid (E, \overline{T}) \in \mathbb{S}\}$
- $\mathcal{A}[[M]]^{\mathbb{S}'}_{\phi,\rho} = (\mathbb{S}'', \phi', \rho')$

$\mathcal{A}[[M \texttt{>>=}_{l_b} N]]^{\mathbb{S}}_{\phi,\rho} = \mathcal{A}[[N]]^{\mathbb{S}'}_{\phi', \rho'[l_b \mapsto \mathbb{S}']\}}$ where $\mathcal{A}[[M]]^{\mathbb{S}}_{\phi} = (\mathbb{S}', \phi', \rho')$

$\mathcal{A}[[M \mid N]]^{\mathbb{S}}_{\phi,\rho} = \mathcal{A}[[N]]^{\mathbb{S}}_{\phi',\rho'}$ where $\mathcal{A}[[M]]^{\mathbb{S}}_{\phi,\rho} = (\mathbb{S}', \phi', \rho')$

$\mathcal{A}[[\texttt{readTVar}\ M]]^{\mathbb{S}}_{\phi,\rho} = (\mathbb{S}', \phi, \rho)$ where $\mathbb{S}' = \{(E, val(M) : \overline{T}) \mid (E, \overline{T}) \in \mathbb{S}\}$

$\mathcal{A}[[\texttt{return}\ M]]^{\mathbb{S}}_{\phi,\rho} = (\mathbb{S}', \phi, \rho)$ where $\mathbb{S}' = \{(E, M : \overline{T}) \mid (E, \overline{T}) \in \mathbb{S}\}$

$\mathcal{A}[[\texttt{M}\ \ \texttt{N}]]^{\mathbb{S}}_{\phi,\rho} = \mathcal{A}[[M]]^{\mathbb{S}'}_{\phi,\rho}$ where $\mathbb{S}' = \{(E, N : \overline{T}) \mid (E, \overline{T}) \in \mathbb{S}\}$

$\mathcal{A}[[\lambda x.M]]^{\mathbb{S}}_{\phi} = \mathcal{A}[[M]]^{\mathbb{S}'}_{\phi,\rho}$
where

- $\mathbb{S}' = \{(\{x = T\} \cup E, \overline{T}) \mid (E, T : \overline{T}) \in \mathbb{S}\} \cup \{(E, \langle\rangle) \mid (E, \langle\rangle) \in \mathbb{S}\}$

$\mathcal{A}[[\texttt{writeTVar}\ M\ N]]^{\mathbb{S}}_{\phi,\rho} = (\mathbb{S}', \phi, \rho)$
where $\mathbb{S}' = \{(\{val(M) = N\} \cup clean(E, val(M)), () : \overline{T}) \mid (E, \overline{T}) \in \mathbb{S}\}$

$\mathcal{A}[[\texttt{case}\ M\ \texttt{of}\ \overline{c\ \overline{x} \to N}\ ]]^{\mathbb{S}}_{\phi,\rho} = (\bigcup \mathbb{S}_k, \bigcup \phi_k, \bigcup \rho_k)$
where $\mathcal{A}[[N_k]]^{\mathbb{S}'_k}_{\phi,\rho} = (\mathbb{S}_k, \phi_k, \rho_k)$ and $\mathbb{S}'_k = \{(\{M = c_k\ \overline{x_k}\} \cup E, \overline{T}) \mid (E, \overline{T}) \in \mathbb{S}\}$

$$\begin{aligned}\mathcal{A}[[f]]^{\mathbb{S}}_{\phi,\rho} &= (\mathbb{S}', \phi, \rho) && \text{if } \mathbb{S}' \neq \emptyset \text{ and } \mathbb{S} \neq \emptyset.\\ &= (\emptyset, \phi \cup \{(f, \mathbb{S}, \emptyset, \rho^{\emptyset})\}, \rho) && \text{if } \mathbb{S}' = \emptyset \text{ and } \mathbb{S} \neq \emptyset.\\ &= (\emptyset, \phi, \rho) && \text{otherwise.}\end{aligned}$$

where $\mathbb{S}' = \bigcap_{\mathbb{S}^* \in \phi(f,\mathbb{S})} \mathbb{S}^*$ and $\phi(f, \mathbb{S}) = \{\mathbb{S}^* | \exists \rho : (f, \mathbb{S}, \mathbb{S}^*, \rho) \in \phi\}$

Fig. 5. Definition of the core analysis function $\mathcal{A}$

every equation from E where T is a subterm of at least one of both terms in the equation. Finally, we denote $clean(E, T)$ as the set of equations $E_{T/T'}$ if there is an element T' such that the equation $T = T' \in E$; if there is no such element, $clean(E, T) = remove(E, T)$.

One of the most important steps in the analysis is the treatment of an atomic section. Before analyzing the content of the section, we add the set of invariants I to the sets of equations in the abstract state. This represents the fact that we suppose that the invariants hold at the beginning of the atomic section. The abstract state $\mathbb{S}'$ obtained when reaching the end of the atomic section is recorded in the trace, but the abstract state resulting from analyzing the atomic section is again the initial one, $\mathbb{S}$. This guarantees that the analysis will be continued with an abstract state that does not include the invariants, as these may no longer hold outside an atomic section due to the interference with other threads. When analyzing the bind operator `>>=`, we analyze the left part of the expression, saving the abstract state for this point in the trace, and we analyze the right part based on the updated abstract values, i.e. the

updated abstract state, environment and trace. For the thread composition, we arbitrarily choose to analyze the left part, and then the right part, based on the same abstract state but with successively updated environment and trace. Reading a shared variable M is handled by adding $val(M)$ to the list of terms in each element of $\mathbb{S}$. Note that, as other STM operations, reading a shared variable will typically be bound to another operation, formally represented by a lambda expression whose lambda variable will be bound to the first term of the list of term ($val(M)$ in this case) upon analysis of the latter operations. A `return M` is handled in the same way, adding the returned expression M to the lists of terms. An application is handled by analyzing the left part of the expression M (representing the function or abstraction that is called) with respect to an abstract state in which the argument expression N is added to the lists of terms. When analyzing a lambda expression, we use the first element of the list of terms in the current abstract state to compose an equation with the lambda variable that will subsequently be added to each element of $\mathbb{S}$. The analysis of a `writeTVar` expression is somewhat more involved. Before erasing the current content of M, we check in all sets of equations if there is an equation of the form $val(M) = T$, which will then be removed, and we replace every occurrence of $val(M)$ by T in the other equations, before adding an equation defining the new value of M. This operation is important as it allows to keep information about the characteristics that hold for the previous value of the shared variable into the equations. Case expressions represent different paths through the program, so we combine the abstract states, the environments and the traces (which have distinct domains as labels are unique) of each path in the case expression. Finally, we deal with function call by looking in the environment what abstract state is associated with the *call pattern*, i.e. the function and the abstract state at the moment of the call. If no such call pattern is recorded in the environment, we add it to the environment so that it can be explored later.

Given the definition of $\mathcal{A}$, we can now define a function $\mathcal{F} : \phi \to \phi'$ that, when applied successively and combined with a suitable abstraction operator, can be made to converge to a fixed point which corresponds to the final environment. This environment contains, for each function, a number of abstract states that represent all call patterns encountered during the analysis and the corresponding resulting abstract states and traces. The definition of $\mathcal{F}$ is given in Definition 4 below.

Definition 4. *Let P be a program where the main function is f_{main}, ϕ an environment for P, and α a suitable widening operator, we define*

$$\mathcal{F}\phi = \bigcup_{(f,\mathbb{S},\mathbb{S}',\rho)\in\phi} \{(f,\mathbb{S},\mathbb{S}'',\rho')\} \cup (\phi'/\phi)$$

where $f \hookrightarrow M \in P$ and $\mathcal{A}[[M]]^{\mathbb{S}}_{\phi,\emptyset} = (\mathbb{S}'', \phi', \rho')$.

The analysis of P is then defined as computing the fixed point, denoted by ϕ^P, of

$$\phi_{n+1} = \alpha(\mathcal{F}\phi_n)$$

where $\phi_0 = \{(f_{main}, \{(\emptyset, \langle\rangle)\}, \emptyset, \emptyset)\}$.

Intuitively, we analyze using $\mathcal{A}$ every call pattern present in ϕ, thereby producing an updated environment. To be sure that a fixed point is reached in a finite number of iterations, we suppose a widening operator α [6]. Even though the definition of a well-performing widening operator is out of the scope of the current paper, we provide a simple operator in Example 1 below.

Example 1. Let $\cap_n$ be an operator for the intersection of tuples of sets S_k and S'_k such as $(S_1, ..., S_n) \cap_n (S'_1, ..., S'_n) = (S_1 \cap S'_1, ..., S_n \cap S'_n)$, a naive operator α for our analysis can be defined as follows: $\alpha(\phi) = \bigcup_{f \in P}(f, \mathbb{S}, \mathbb{S}', \rho')$ where $(\mathbb{S}, \mathbb{S}', \rho') = \bigcap_3 \{(E_1, E_2, \rho) | (f, E_1, E_2, \rho) \in \phi\}$.

3.3 Proof of Race-Freeness

In the previous section, we have defined an analysis that, for a program P and a set of invariants I, produces an environment ϕ^P that contains traces. We denote by ρ^P the set of all couples $(l, \mathbb{S})$ computed by the analysis ϕ^P, which is formally defined as:

$$\rho^P = \bigcup \{(l, \mathbb{S}) \mid (f, \mathbb{S}', \mathbb{S}'', \rho) \in \phi^P \wedge \rho(l) = \mathbb{S}\}.$$

The key idea of the race detection using the results of our analysis is to check, for each atomic section, that the abstract state associated to the exit point of the atomic section fulfills the invariants. This proof can be expressed using the *first-order logic*. Indeed, the equations in the abstract states, as well as the invariants, can be expressed in first-order logic in a straightforward way. Proving an invariant will generally require a number of *axioms*. As said before, an axiom allows to link a characteristic to the program code, thereby specifying under what condition a characteristic holds during program execution. In our framework, an axiom is a first-order formula that typically involves a quantifier. Let $FO(\mathbb{S})$ the first-order formula that is equivalent to the information of the abstract state $\mathbb{S}$, we will say that the set of invariants I *can be derived from* $\mathbb{S}$ *using the set of axioms* $\boldsymbol{A} = \{A_1, ..., A_n\}$ if every invariant in I is a logical consequence of $FO(\mathbb{S}) \wedge A_1 \wedge ... \wedge A_n$ in the first-order logic. It follows that a program P is *race free with respect to the set of invariants* I *and a set of axioms* $\boldsymbol{A}$ if :

$$\forall\, \mathbb{S} \in \{\mathbb{S}' \mid (l_a, \mathbb{S}') \in \rho^P \wedge l_a \in \mathcal{L}_a\} : I \text{ can be derived from } \mathbb{S} \text{ using } \mathbf{A}.$$

In practice, we translate the results of our analysis to the TPTP syntax [23] and we use an efficient theorem prover for the first-order logic with equality [19] to check the derivation. To illustrate, we show in Fig 6 the TPTP-translation which is generated by our analysis for the running exemple of Section 2, and which is used by the prover [19] to successfully check the invariant.

4 Conclusion

Related Work Different approaches exist to deal with the detection of higher-level anomalies in concurrent programs. These approaches differ in the distinct

```
fof(eq1,axiom,(s = sum(tab))).
fof(eq2,axiom,(val(shTab) = cons(n,tab))).
fof(eq3,axiom,(val(shSum) = plus(s,n))).
fof(eq4,axiom,( ! [X,XS] : (sum(cons(X,XS)) = plus(sum(XS),X) ))).
fof(invariantsatisfactory,question,( sum(val(shTab)) = val(shSum))).
```

Fig. 6. TPTP file produced by our analysis

range of bugs they can handle. That suggests the existence of different compromises ranging from sometimes too strict to sometimes too weak correctness criteria. The concept of *high-level data race* was introduced by [1]. Such a race occurs when a set of shared memory locations is meant to be accessed atomically, but the memory locations are accessed separately somewhere in the program [25]. Note that the definition does not include *what* is the link between two shared variables. As a consequence, the static and dynamic detection of high-level data races [1, 16, 25] can trigger false positives (when two shared variables are accessed together but in a bad way) and false negatives (when two shared variables are accessed separately but it does not violate the link between them). However, these detection tools are very useful in practice because they can find a lot of races and typically do not require additional annotations. Moreover, they can be extended to also detect *stale-value errors*, i.e. variable replica that are outdated [16, 2]. An alternative to this approach is to deal with the subtly variant concept of *application-level race condition*, which was introduced by [3]. This notion not only captures the existence of a link between shared variables, but *explicitly* involves the *nature* of that link. By defining under what condition another thread can violate a program invariant, detection of application-level race conditions is more accurate, avoiding a lot of false positives and negatives. On the other hand, analyzes are typically heavier to use as the programmer has to provide additional annotations. Our work is inspired on the static analysis developed for verifying atomic blocks in an object-oriented language [3]. However, specific issues must be addressed, such as the particular control-flow inherent to higher-order functional programming while [3] mainly deals with access permissions and unpacking methodologies related to object-oriented programming. Moreover, our approach involves a very different formalism, with different expressive power, to specify and to verify the consistency of the shared variables. We think that our more declarative formalism works well with high-level properties that must be maintained throughout the program, while the *typestate* used in [3] are adapted to describe, for each function, in which conditions the function is allowed to be called while keeping the program race-free. We feel that our annotations are well-separated from the source code, easy to write and easy to incorporate in an application in a clean and straightforward way. Note that the invariants that we use are different from those that can be checked dynamically in STM Haskell [8], since they involve self-defined characteristics. Being dynamic, this checking cannot guarantee validity for every execution but can be very complementary to our analysis in order to get more details about a consistency violation.

Future Work In order to reduce the likelihood of race conditions, the programmer could be tempted to use too coarse-grained atomic sections and overprotect some (inoffensive) computations, thereby badly reducing the potential parallelism in the program. We advocate that the information provided by our analysis may be used in order to find the optimal size of the atomic sections within a program. This is why our analysis also stores the *intermediate* abstract states, i.e. the labels of $\mathcal{L}_b$. We can easily imagine different strategies that use this information to reduce the size of too coarse-grained atomic sections while keeping the program race-free. A simple strategy is to split an atomic section at a point where the intermediate abstract state fulfills the invariants and to check that the new program is still race-free. More evolved and powerful strategies imply the reordering of the statements based on a data-flow analysis. Also as a future work is the question of a suitable and effective widening operator to deal with recursion inside atomic sections. Finally, a key element for the practical usefulness of our framework is the research of a higher-level formalism (which fits with our analysis) for the specifications that the programmer must provide.

Acknowledgments. We thank the anonymous reviewers for their constructive comments on a previous version of this paper.

References

1. C. Artho, K. Havelund, and A. Biere. High-level data races. In *NDDL/VVEIS*, pages 82–93, 2003.
2. C. Artho, K. Havelund, and A. Biere. Using block-local atomicity to detect stale-value concurrency errors. In F. Wang, editor, *Automated Technology for Verification and Analysis: Second International Conference, ATVA 2004, Taipei, Taiwan, ROC, October 31-November 3, 2004. Proceedings*, volume 3299 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2004.
3. N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In G. E. Harris, editor, *OOPSLA*, pages 227–244. ACM, 2008.
4. J. Borgström, K. Bhargavan, and A. D. Gordon. A compositional theory for stm haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 69–80, New York, NY, USA, 2009. ACM.
5. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing races and deadlocks. *ACM SIGPLAN Notices*, 37(11):211–230, Nov. 2002.
6. A. Cortesi. Widening operators for abstract interpretation. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40, Washington, DC, USA, 2008. IEEE Computer Society.
7. T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.
8. T. Harris and S. Peyton Jones. Transactional memory with data invariants. In *TRANSACT '06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, jun 2006.

9. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. *SIGPLAN Not.*, 39:1–13, June 2004.
10. P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992.
11. S. P. Jones. Beautiful concurrency. In A. Oram and G. Wilson, editors, *Beautiful Code*, pages 385–406. O'Reilly & Associates, Inc., Sebastopol, CA 95472, 2007. ch. 24.
12. K. Korovin and C. Sticksel. iprover-eq: An instantiation-based theorem prover with equality. In *5th International Joint Conference on Automated Reasoning, IJCAR 2010*, Lecture Notes in Computer Science, pages 196–202, Berlin / Heidelberg, 2010. Springer.
13. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In S. J. Eggers and J. R. Larus, editors, *ASPLOS*, pages 329–339. ACM, 2008.
14. K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. *SIGPLAN Not.*, 43:51–62, January 2008.
15. F. Nielson, H. R. Nielson, and C. Hanking. *Principles of Program Analysis*. Springer, 2005.
16. V. Pessanha, R. J. Dias, J. a. M. Lourenço, E. Farchi, and D. Sousa. Practical verification of high-level dataraces in transactional memory programs. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '11, pages 26–34, New York, NY, USA, 2011. ACM.
17. P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.
18. A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, Aug. 2002.
19. S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15:111–126, August 2002.
20. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
21. M. Sottile, T. G. Mattson, and C. E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman & Hall/CRC, 1st edition, 2009.
22. S. Stork, P. Marques, and J. Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 933–940, New York, NY, USA, 2009. ACM.
23. G. Sutcliffe, S. Schulz, K. Claessen, and A. V. Gelder. Using the tptp language for writing derivations and finite interpretations. In *In Ulrich Fuhrbach and Natarajan Shankar, editors, Proc. of the 3rd IJCAR, Seattle, volume 4130 of LNAI*, pages 67–81. Springer, 2006.
24. H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.
25. B. Teixeira, J. a. Lourenço, E. Farchi, R. Dias, and D. Sousa. Detection of transactional memory anomalies using static analysis. In *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '10, pages 26–36, New York, NY, USA, 2010. ACM.
26. S. Thompson. *The Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

A Gradual Polymorphic Type System with Subtyping for Prolog

Spyros Hadjichristodoulou[†⋆] and David S. Warren[†⋆⋆]

[†]Department of Computer Science, Stony Brook University, Stony Brook, NY

Abstract. Although Prolog was designed and developed as an untyped language, there have been numerous attempts at proposing type systems suitable for it. The goal of research in this area has been to make Prolog programming easier and less error-prone not only for novice users, but for the experienced programmer as well. Despite the fact that many of the proposed systems have deep theoretical foundations that add types to Prolog, most Prolog vendors are still unwilling to include any of them in their compiler's releases. Hence standard Prolog remains an untyped language. Our work can be understood as a step towards typed Prolog. We propose an extension to one of the most extensively studied type systems proposed for Prolog, the Mycroft-O'Keefe type system [1], and present an implementation in XSB-Prolog. The resulting type system can be characterized as a *Gradual* type system [2], where the user begins with a completely untyped version of his program, and incrementally obtains information about the possible types of the predicates he defines from the system itself, until type signatures are found for all the predicates.

1 Introduction

Since the seminal work of Mycroft and O'Keefe [1] in introducing a polymorphic type system for Prolog, there has been vast research on the area. Some approaches extend or reconstruct their type system [3], while others take different paths for introducing types in logic programs [4,5,6].

The common denominator, however, is that they are about *theoretically* defining and constructing a type system for Logic Programming[1]. Although some early Prolog implementations contained type checking mechanisms based on the Mycroft-O'Keefe type system (e.g. the DEC-10 compiler), most modern systems keep Prolog as an untyped language [2]. Mercury[3], and Visual-Prolog[4] are the exceptions which have become famous for the type-checking abilities they offer

⋆ shadjichrist@cs.stonybrook.edu

⋆⋆ warren@cs.stonybrook.edu

[1] With the exception of [3], where an actual implementation is described

[2] As we will discuss later, an implementation for the Mycroft-O'Keefe type system was developed with the intention of being distributed with SWI-Prolog and YAP

[3] http://www.mercury.csse.unimelb.edu.au/

[4] http://www.visual-prolog.com/

to programmers; however, this comes with the price of strong typing and fewer primitive types than Prolog actually assumes in programs in these languages.

As discussed in [1], the main purpose of developing a polymorphic type system for Prolog is to provide the programmer with another tool to make programming easier and less error-prone. We consider this work as a step in that direction; our goal is to build a working type system which will enable programmers to write correct Prolog programs more easily than before. The challenge is to combine various aspects of the approaches introduced in the literature to implement a robust type inference system for XSB Prolog [7], using its tabled engine's capabilities. Our type checking and inference mechanism will offer users two modes of operation; firstly, they will be able to type-check their program, if they provide a type signature for every predicate they define. Secondly, if some (or all) predicate type signatures are missing, the type inference engine will automatically infer them and present them to the user. If the user indicates that he is satisfied with the type inference engine's suggestions, then these types will be considered by the system as if they were declared. This process will be conducted incrementally, *gradually*; the user will start with a completely untyped (or partially typed) version of his program, and type inference will gradually give more information about what the types of the defined predicates may be. The user can stop the process at any point; even if some of the predicates remain with no types declared, the system guarantees that the typed predicates will never be used with arguments of the wrong type. This kind of type system, called *Gradual Type System* was introduced in [2].

2 Background and related work

The first work introducing some kind of type checking and inference in Prolog was Mycroft and O'Keefe's Polymorphic type system, [1]. It is based on the seminal work by Robin Milner, [8], who created a polymorphic type system for the ML family of functional programming languages, and first introduced the notion of "Well-typedness". In the Mycroft-O'Keefe type system, type signatures are provided by the user for each defined predicate, and the type checker's task is to verify that each use of a predicate respects the signature declaration. The only notion of *inference* in this type system is with **Variables**; when a predicate `p(X,Y)` is type-checked against its signature, a type is inferred for both `X` and `Y`. Also, it allows for *polymorphism*; arguments of predicates and terms can be of *any* type, denoted by a type variable. Finally, it allows users to define their own type constructors.

Example 1. The type constructor for lists is

```
:- type list(A) ---> [] ; [A|list(A)].
```

The declaration `:- type` is built-in, and it tells the system that a type constructor declaration for the type `list(A)` follows after `--->`. `A` is a *type variable*, so any possible type can be used in its place. In this case, the recursively defined datatype `list(A)` has two constructors: either `[]` for the empty list, or `[A|list(A)]` for

non-empty lists. As we can see from the second constructor, this definition ensures that a list may contain only elements of the same type, since the only way to produce the type `list(A)` from a non-empty list is if the list's first argument is of type `A`, and the rest of the list is of type `list(A)`.

Mycroft and O'Keefe introduce the notion of "Well-typedness", which has the same meaning as in the Hindley-Milner type system: "Well-typed programs can't go wrong". In the context of Prolog, this means that in any execution of the program, no predicate will be called with arguments that don't respect the type signature declared by the user. The following example illustrates the operations described above:

Example 2. The user declares a type for the well-known `append/3` predicate, as

```
:- pred append(list(A),list(A),list(A)).
```

This means that `append/3` has 3 arguments, and each has the same type, namely `list(A)`, so each argument can be a list of anything, as long as all 3 arguments have the same type. For the following definition of `append/3`,

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

each clause of `append/3` will be type-checked against the given signature. Notice that each clause respects the declared signature, and so `append/3` is well-typed. Finally, if the clause, `p(X,Y,Z) :- ... , append(X,Y,Z), ...`, appears in the program, then types for `X`, `Y` and `Z` are inferred according to `append/3`'s signature (and all are `list(A)`, for some `A`).

Schrijvers et. al. [3] recently developed an implementation of the Mycroft-O'Keefe type system. Their aim was to gradually introduce types in Prolog using a type-checking library that was intended to be shipped with two of the most popular Prolog implementations, SWI-Prolog and YAP. This type checking library makes it easy to interface typed and untyped code, by performing runtime checks when predicates with declared types are called from the bodies of predicates with no types declared. This guarantees that each typed predicate will always be called with values of the correct type. If no types are declared, no type-checking (at compile-time or at run-time) is performed, and the program's execution proceeds as usual. This "on demand" use of the type system makes the migration from untyped to typed Prolog easier. However, we need to make a distinction between the meaning of the term *gradually* in their approach, as opposed to ours. The goal of [3] was to change the perception that programmers have of Prolog, and drive them towards thinking about Prolog as a typed language. Thus, before writing a Prolog program, their type system enables users to think first about the type signature each predicate in the program must have in order for it to behave as the programmer envisions. This change of perception of Prolog as a typed language is meant to be done gradually with the help of their type system; in this approach, the term *gradually* corresponds to the *time* needed to change this view in the programmers' minds. On the other hand, we do not wish to change the

way programmers see Prolog and write their programs. Our work is meant to *help* the programmer better understand a program, and reason more accurately about it (why it fails, in which cases etc.). The information presented by our system to the user will be given gradually, incrementally for each particular program, so in our approach, *gradually* corresponds to the process of inferring information about the types of the predicates used in each program.

A rather different approach was intruduced by Barbuti and Giacobazzi [5]. The authors employ a fixed-point, bottom-up, abstract interpretation technique in order to infer types for Horn-Clause programs. Type declarations for predicates and constructors are very similar to the Mycroft-O'Keefe type system [1], but in [5], the system can infer types for predicates while not depending on a type signature provided by the user. For example, the type of `append/3` can be inferred only by its definition and the definition of the `list(A)` constructor, as given above.

By allowing type *union* in the domain of types, this type inference scheme makes it possible to capture the type of heterogenous data structures. The example given in [5] is one of lists containing elements of two possible types; natural numbers and characters. This is a quite useful feature for a type system for Prolog, since programmers often use the flexibility given to them by the language itself to define *metapredicates* over data structures of unknown (at compile time) types. A common example is the *univ* metapredicate, `=..` . Its first argument is always a term, but its second argument is a list containing the name of the term and the term's arguments. The importance of this kind of extension to the Mycroft-O'Keefe type system is quite significant, since it makes assigning types to metapredicates like `univ/2` more natural, as opposed to the approach taken in [1] where new kinds of type constructors were defined for this purpose.

The term "gradual" characterizes type systems that mix static with dynamic type checking in order to combine the benefits of both in a single framework, and was introduced in [2]. In a gradual type system, the user has control over which portions of the program he wishes to statically type check, by adding appropriate explicit type annotations, and the rest of the program gets dynamically type checked. Type inference in a gradual type system was first introduced in [9] for an extension of simply typed lambda calculus. A unification-based algorithm was employed for type inference, which consists of generating certain constraints and then solving them in order to get a valid typing for the program. An implementation of this type system for the extended simply typed lambda calculus was written in Isabelle [5].

This approach was taken one step further in [10], where the authors extended the techniques described in [2] in order to build a gradual type system for ActionScript [11]. Their main motivation was to use type inference to enchance the performance of existing gradually typed programs by determining (at compile time) sets of explicit casts that were unnecessary, thus reducing the necessity for runtime type checking.

[5] http://www.cl.cam.ac.uk/research/hvg/Isabelle/

Finally, the vision we have about gradually adding types to Prolog programs is in many aspects similar to the work of Sagonas and Luna [12]. They employ Dialyzer, which is a static analysis tool for Erlang, in order to uncover type mismatches between function type signatures that users have declared while developing their code and the type in each use of such functions. Fixing this kind of warnings thrown by Dialyzer in a semi-automatic way allowed them to catch programming errors and better document the intended use of functions.

3 Extending the Mycroft-O'Keefe Type System

As noted in section 1, the Mycroft-O'Keefe type system [1] allows for the kind of polymorphism where arguments of predicates and terms can be of *any* type, denoted by **type variables**. So, the only information we can get from a type signature is a restriction on the *kind* of types that each argument can take. As seen in example 1, the type constructor for lists is

```
:- type list(A) ---> [] ; [A|list(A)].
```

The only information we get from this type constructor is that lists may only contain elements of the *same* type, hence restricting the type system to accept only *monomorphic* lists. However, this is not standard Prolog behavior. As a matter of fact, we can have lists with elements of any possible type in Prolog. For example, `[a,f(X),[42],g(h(Z))]` is a valid list. In order to keep the behavior of Prolog flexible, but still be able to reason about the possible dependencies between the types of elements of a list for example, we propose the addition of a (fixed by the system) classification, or *subtyping* between the primitive types.

Example 3. Let's consider a predicate `add/3`, which takes two numbers as its two first arguments, and binds its third to their sum. We would like our predicate to be able to add both integers and floats, so there are 4 different possible signatures for `add/3`:

```
:- pred add(integer,integer,integer).
:- pred add(integer,float,float).
:- pred add(float,integer,float).
:- pred add(float,float,float).
```

Instead of having these 4 different signatures for `add/3`, we define a new type, `number`, which denotes that elements of that type can be either `integer`s or `float`s, so $number := integer \cup float$. The new signature for `add/3` is

```
:- pred add(number,number,number).
```

This signature is a generalization of the 4 signatures given above. The semantics is the straighforward given by our intuition; each argument is of type `number`, so each can be either an `integer` or a `float`.

We can consider `integer` and `float` to be subtypes of `number`, so from now on, whenever `number` is used, both `integer` and `float` can be also used. A graphical representation of our subtyping relation so far can be found in figure 1.

Apart from the basic `integer` and `float` types, in the implementation of the Mycroft-O'Keefe type system described in [3], the `atom` type is also introduced. Its purpose is to type-check predicates, the arguments of which are Prolog atoms, but not integers or floats. In our approach, we have the `atom` type for the same reason as well, but we also introduce a new type, called `atomic`, in order to handle the relationships between the `atom`, `integer`, `float` and `number` types. The following example illuminates this point:

Example 4. Let's consider a predicate `bytes/2`, which will take as first argument either a Prolog atom or a number, and bind the second argument to the number of bytes that will be needed to represent the former in a 32-bit architecture. Instead of having the following two distinct type signatures for `bytes/2`,

```
:- pred bytes(atom,integer).
:- pred bytes(number,integer).
```

we can use `atomic` to give only one signature which will capture the same information:

```
:- pred add(atomic,integer).
```

The first argument is of the `atomic` type, which means that it can be either an `integer`, a `float` or a `atom`, so $atomic := integer \cup float \cup atom$.

A graphical representation of our subtyping relation involving the `integer`, `float`, `atom`, `number` and `atomic` types is shown in figure 2.

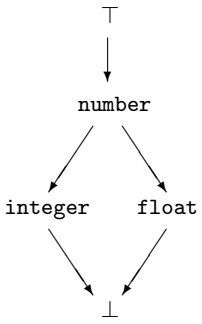

Fig. 1. Types `number,integer,float`

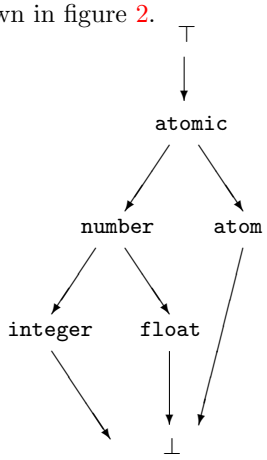

Fig. 2. Types `atomic,atom,number,integer,float`

In the previous paragraphs we explained how our extensions to the Mycroft-O'Keefe type system are used to assign types to predicates for which we have enough information to do so. But what about untyped predicates, or constructors for which we have no type information? In such cases, we want our type system to assign to the entire term or constructor a general type, which in essence

means that we want to treat it as a whole, instead of giving types to each of its arguments. We call this type `struct`, and we will assign it to any term or constructor for which there is no information about its type.

Example 5. Our next example involves a `lookup/3` predicate; it takes as arguments an *environment*, which is represented as a list of `var = value` pairs (where `var` is an atom and `value` an integer) and the name of a `var`, and returns the respective `value`. If the only type constructor available is the one of lists (1), then the most specific type we can give to `lookup/3` would be

```
:- pred lookup(list(struct),atom,integer).
```

Performing type inference on the definition of `lookup/3` would yield this type, if the only available type constructor was the one of lists. However, this is not what the programmer had in his mind as the intended type of `lookup/3`. It's conceivable that he wanted to declare a type constructor for "pairs", as

```
:- type pair(A,B) ---> A (=) B.
```

Then, the type of `lookup/3` would be

```
:- pred lookup(list(pair(atom,integer)),atom,integer).
```

Having the `struct` type available makes it easier for the programmer to catch errors in cases where special constructors are used in programs. When a type `struct` is inferred for some argument of a predicate, it can be understood as if it is telling the programmer that maybe a user-defined type constructor should be used there, which will in the end make more sense.

In order to interface the relationship between our newly defined `struct` type with the ones defined in the previous paragraphs, we need a new type, which will act as a supertype of all of them. We will use the top ($\top$) type for this purpose, which is declared in type signatures as a type variable. With respect to figure 2, `struct` would be a subtype of $\top$ and a supertype of $\bot$ with no direct arrows to the other types.

4 Type Inference

Types for various predicates are not always known to the programmer. This is especially true when importing predicates from modules, or even when using pieces of code from various other sources. Thus, a critical capability that our type system should have is to (try to) infer the types of those predicates for which there is so far no information. In this section, we describe our approach to implementing type inference for Prolog programs.

We take the approach described in [5] one step further by using a *tabled* Prolog engine for the evaluation. In order to get an answer for a call, the tabled engine has to either retrieve an answer from the global table if it exists, or get an

answer for each goal that appears in the body of each clause[6] that unifies with the original goal. The idea for our implementation is to treat Prolog variables as type variables, so the same evaluation scheme applies to our approach as well; in order to get a type for a particular predicate, we have to find the type of each predicate in the body of the clauses that define it, or just get the type from the global table. The only difference is that instead of unifying a variable with its value(type), we accumulate variable-type pairs in an environment, and do the binding in the end.

In essence, all the possible values that a variable can take are abstracted by the *intersection* of the possible types each has, so in the end we will get the most *specific* type that can be inferred for a predicate. This may seem rather illogical at first, since when two answers are produced for a predicate, we keep the most general one. The reason is that when trying to find answers for goals in Prolog, we don't care which clause of the goal will make the answer found true, as long as there is one that does. On the other hand, in type inference, the type inferred must respect *all* the clauses of the predicate. We can think of this difference as the duality between *union* and *intersection*; when we want answers for a goal we are looking for the *union* of answers, whereas when we want to find a type for a predicate, we want the *intersection* of types found.

A high-level description of the main scheme of our approach to type inference is shown in algorithm 1. Type inference for each clause is independent from other clauses, so the idea is to gather as much information as possible from each clause, combine everything together to get a new type for the head predicate, and repeat the process using the newly inferred type. This fixpoint operation is defined over the type of the head predicate, so when that stops changing between iterations, the fixpoint has been reached. One fine point of this procedure, is the way in which the intersection of types found for each clause is taken. Since the types are Prolog terms, one would think that it is enough to unify everything together, and the result would be any of the unified types. However, there is a case where at each step, simply unifying everything together will result in constructing larger and larger terms, without ever reaching the fixpoint. In order to remedy this, instead of simply unifying the types together, we use the `occurs check`, which prevents such cases from arising by failing unification of terms like `X = f(X)`.

Example 6. In this example, we will demonstrate how the type of `member/2` is inferred using as type information only the type declaration for lists (1).

```
(1) member(X,[X|_]).
(2) member(X,[_|L]) :- member(X,L).
```

I1a From clause 1 we learn that the first argument should be of type α, and the second of the type $list(\alpha)$, due to the definition of the type constructor for lists, so the first approximation for `member/2`'s type would be `member(A,list(A))`

I1b Since iteration 1 is not yet finished, we have no information regarding the type of `member/2`, so we have to rely on the head again (as we did for iteration 1a).

[6] Similarly, either by calling it, or by retrieving an answer from the global table, if one exists

Since the two arguments share no common variables, there is not much we can learn from this second clause at this point; only that the first argument should be β, and the second $list(\gamma)$, so the approximation for `member/2`'s type from the second clause is `member(B,list(C))`

U At this point, we need to unify with occurs check `member(A,list(A))` and `member(B,list(C))`. The result is `member(A,list(A))`, and that is the approximation for `member/2`'s type as retreived from the first iteration of the evaluation

I2a From clause 1, having in mind the type inferred from Iteration 1, we learn exactly what we learned in iteration 1a (since there is no body in this clause)

I2b Now that we know that an approximation of the type of `member/2` is `member(B,list(B))`, we will use that for the call to `member/2` appearing in the body of the second clause. So, since the first argument should be of type β and the second of type $list(\beta)$, then `X` and `L` should be connected in the same way; `X` should be of type β, and `L` of type $list(\beta)$. Thus, the type for `member/2` from this iteration is `member(B,list(B))`

U Unifying with occurs check `member(A,list(A))` and `member(B,list(B))` will give us a new approximation for the type of `member/2`, which will again be `member(A,list(A))`. At this point, since the type inferred for `member/2` has not changed, we have reached the fixed point of our evaluation, and we can return the type found in this iteration as the answer

The result of the previous example is consistent with what intuition says about the type and the operation of `member/2`; its second argument should be a list of things of the same type as the first argument. So when a call to `member/2` is made where this connection between the arguments is not present, the call is guaranteed to fail. For example, a call to `member(a,[1,2,3])` is guaranteed to fail, because the first argument is of type `atom`, whereas the second is of type `list(integer)`. This is the kind of error that our type system is designed to catch, as also mentioned in [1] and [3].

Algorithm 1 Type Inference

1: **repeat**
2: For a predicate H, let $\{H_1, \ldots, H_n\}$ be the heads of the clauses that define it
3: **for all** $H_i \in \{H_1, \ldots, H_n\}$ **do**
4: Let $H_i :- B_i$ be the respective clause of H
5: **for all** $B \in B_i$ **do**
6: Find the type of B and accumulate variable-type bindings for the entire clause
7: **end for**
8: The new type for H is the intersection of the types found for each H_i
9: **end for**
10: **until** No change is made to the inferred type for H

5 Graduality

After extending our type-checking mechanism to allow for subtyping, and introducing a type inference scheme based on the extended Mycroft-O'Keefe type

system, we are now ready to consider combining the two features in a single framework. There are various ways in which type checking can interface with type inference. An obvious one, would be to have two modes of operation, one for type inference and one for type checking, and let the user decide under which he wants to operate the system each time. This way there is no interaction between type inference and type checking, and this can make such a system hard to use and the programmer less productive.

The approach we take is to combine these two modes of operation, and let the system decide when to perform type checking, and when to infer types. We employ the idea of *graduality* for our type system, which, as described in [2], means that the resulting system provides the user with the advantages of both static and dynamic typing. More recent advantages in the area of type systems [9,10] have incorporated type inference in gradual type systems, mainly for improving performance of gradually typed programs. The reason we chose to implement a gradual type system for Prolog is not for efficiency of execution of Prolog programs, but for making Prolog programming in general easier and less error-prone while also being able to treat completely untyped programs the same way as they are currently treated by commercial compilers.

Adding types to a program using our type system is an iterative process. The user begins with a partially or completely untyped program (i.e., a program that may or may not contain type signatures for some of the predicates it defines) and type checking occurs where applicable. The type inference engine then infers types for all the untyped predicates (i.e., predicates for which type signatures have not been provided by the user), and if the user is satisfied with the inferred types, the proposed signature is considered as declared, and the whole process can run again. It's up to the user when he wishes to halt the type checking - type inference procedure and run his program; partially well-typed programs will behave the same way as the original untyped program would behave. Following example 5, after the first run of type inference in the program, the type inferred for `lookup/3` would be `lookup(list(struct),atom,integer)`. The programmer then realizes from the presence of the `struct` type in this signature that he needs to add the type declaration for pairs, so that the information he will get from the next run of type inference would be more precise.

6 Implementation

The main idea of our implementation is that type-checking and type-inference should be performed as a pre-processing step, before the source code is compiled and loaded into memory. Since both the type-inference and type-checking engines are written in Prolog, we are using XSB as the actual preprocessor. XSB uses *XPP*[7] to preprocess programs. Users can define their own pre-processors based on XPP, and invoke them as the last pre-processing step in the chain of XSB's compiler procedure. The pre-processing procedure is depicted in figure 3.

[7] http://www.cross-browser.com/x/docs/xpp_reference.php

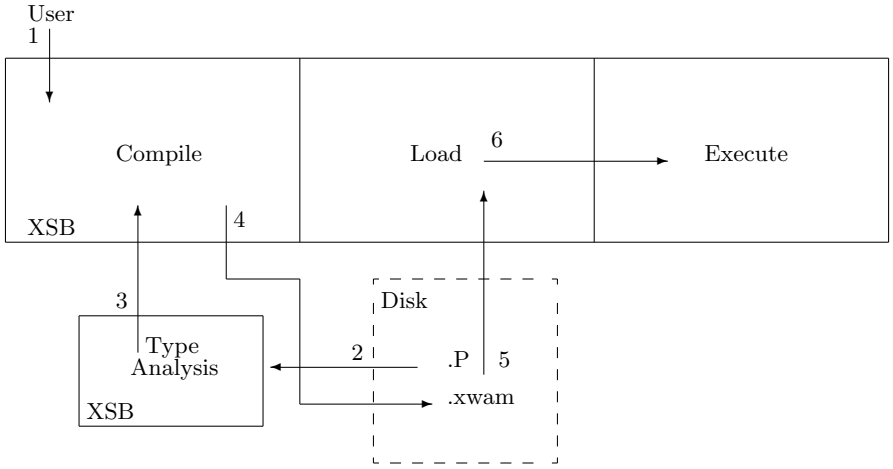

Fig. 3. Data-flow figure for the pre-processing procedure

This approach has some benefits when the implementation is to be released with a popular Prolog compiler:

- Retreiving type information about the input program, or verifying that the given type information is correct with respect to the proposed extended Mycroft-O'Keefe type system does not interfere with the program's evaluation, so any Prolog engine can be used to evaluate the post-processed type-safe program
- After the separate XSB process that was launched to perform type-checking and type-inference on the input program quits, all the information regarding the process is abolished, since there is no communication between the process that performs type analysis, and the process that is used to compile the program. This is particularly useful when working with modules; the module semantics defined for XSB is preserved since each module will be individually pre-processed. The only information loaded into XSB would be all that is generated by XSB when compiling the input program. Hence, the behavior of XSB gathering type information from modules M_1 and M_2, compiling and loading them will be the same as if type-checking and type-inference never occured

In [3], the authors describe a robust implementation for the Mycroft-O'Keefe type system, which is the basis for the implementation of our type-checking mechanism[8]. We retained most of the infrastructure and bookkeeping procedures, and only added the necessary pieces to implement the subtyping relation we described in a previous section.

[8] The authors would like to thank Tom Schrijvers for providing us with the type-checking library

The type-inference engine was built on top of the type-checking mechanism. Since all the information necessary to perform type-inference is already maintained by the type-checking code, building the former as an extension of the latter was the obvious choice. We needed, however, to implement the core of the type-inference engine, which is described in algorithm 1. There are two main challenges with implementing the type-inference engine: storing the inferred types of predicates so that they will be accessible anytime they may be needed, and *termination*; we don't want the type inference engine to fall into infinite loops because while trying to find the type of a predicate it doesn't learn anything from each iteration.

Tabling is an implementation technique invented mostly to solve this kind of problems in logic programs, hence tabling the core of the type inference engine automatically gives the correct behavior. Let us assume that the predicate that implements the fixpoint operation of type inference is `type_inference_n/2`, and that it is tabled. When we want to infer the type of a predicate, say `foo/1`, we call `type_inference_n(foo(_),Type).` When an answer to this call is found, a record will be entered into the global table of `type_inference_n/2`, for example `type_inference_n(foo(_),foo(integer))`.

If we ever need `foo/1`'s type again during the inference of some other predicate's type, calling `type_inference_n(foo(_),Type)` will just retrieve the answer from the global table, thus the need of explicitly keeping and passing around a list of the so-far inferred types is unnecessary. Tabling also solves the termination problem. If `foo/1` is defined as `foo(X) :- ..., foo(Y), ...` then calling `type_inference_n(foo(_),Type)` while evaluating the body of `foo/1` with pure Prolog, we would fall into an infinite loop, because we always make the same initial call. However, when using tabled Prolog, such positive loops are terminated by *failing* the initial call. In such cases, instead of terminating and returning with no type for the respective predicate, we can combine the variable-type bindings that are so far accumulated and give some approximation of the predicate's type.

Fortunately, XSB's tabling capabilities go beyond just storing answers in a global table for easy retreival and termination of (positive) loops (known as *variant* tabling). Swift and Warren [13] have proposed and implemented a more sophisticated tabling method, called *Answer Subsumption*. If a predicate is tabled with answer subsumption, the first answer found for a call to that predicate is entered in its global table. If another answer is found, it is joined with the one that already resides in the table, and only the join of the two is kept. The join operation is user-defined, and can generally be any lattice operation. This approach is quite useful for solving problems which need *aggregation*, and as such we will treat type inference.

If we were to try to implement algorithm 1 with variant tabling, we would store the types found for each of the clauses of a predicate into the table, and then use `findall/3` to go through them and unify them together in order to find the most specific type. There is, however, an issue with this approach. Consider the case of two mutually recursive predicates; finding the type of one, depends on finding the type for the other, and vice versa. In this case, we would have to mix calls to `findall/3` with tabling, which do not work well together. Due to

the *Completion* operation of SLG-WAM, since `findall/3` is essentially a form of negation, the computation of some calls will be delayed for a later point, so the information needed for type inference will not be available at the correct point of execution.

This problem does not occur if we decide to use answer subsumption as our tabling mechanism. As long as we use a proper join operation, the underlying engine will automatically join together answers computed for the `type_inference_n/2` predicate, and only keep the joined version in the table. When a new type is found, it will be joined with the previous entry in the table, and we will have the newest (and possibly more refined) type available. A high-level description of the implementation of `type_inference_n/2` can be found in figure 2 (lines 2-3 for clause 1, and lines 4-9 for clause 2).

The final point we need to make is about the join operation we use. The first and obvious choice would be to use *unification* as the join operation. We previously mentioned that we need to link together type variables with Prolog variables. As a matter of fact, they correspond to each other throughout the type inference process we just described. The only place where type variables and Prolog (logic) variables have different semantics and hence work differently, is when joining two types together. The explanation is similar to the difference between *union* and *intersection* of answers we mentioned previously. When two answers, `p(X)` and `p(a)`, are computed for a predicate `p/1` using unification as the join operation, the second is subsumed by the first, so the only one kept in the table is `p(X)`. However, if the argument of `p/1` corresponds to its type, `p(X)` would be the wrong answer, since we need the most *specific* type. Hence, the correct answer would be `p(a)`. The solution to this issue is a rather simple one, even if it may seem illogical at first. The only thing we need to do is keep *ground* terms in the table, and unground them only when unification is necessary outside the join operation. This is easily done with the builtin `numbervars/3` and `unnumbervars/3` predicates. A description of the join operation can be found in figure 3. The top-level predicate `type_inference/2` called from the pre-processor just calls `type_inference_n/2` and ungrounds the type found with `unnumbervars/3`.

Algorithm 2 `type_inference_n/2`

1: `type_inference_n(Pred,Type) :-` {Clause 1} {Let `Pred` be the predicate which we want to infer the type for and `Type` be the type inferred for Pred}
2: Create a copy of `Pred` with fresh variables, namely `Type`
3: Ground `Type` with `numbervars/3`
4: Create a copy of `Pred` with fresh variables, namely `Pred1` {Clause 2}
5: Find a clause with `Pred1` in the head and body `Body`
6: Get the environment `Env` consisting of variable-type pairs from `Body`
7: From `Env` compute a type for `Pred1`, namely `Type1`
8: Create a copy of `Type1` with fresh variables, namely `Type`
9: Ground `Type` with `numbervars/3`

An important feature of our type system is that no new syntax is introduced, thus type information can be computed for any pure Prolog program. Our

Algorithm 3 The join operation `type_unify/3`

1: `type_unify_n(XX,YY,Y1) :-` {Let `XX` be the type that already resides in the table, `YY` the type just computed, and `Y1` the join}
2: Create copies of `XX` and `YY` with fresh variables, namely `X` and `Y`
3: Unground `X` and `Y` with `unnumbervars/3`, creating respectively `X1` and `Y1`
4: Unify `X1` with `Y1`
5: Ground `Y1` with `numbervars/3`

approach can be characterized as a graceful attempt to assign types to Prolog programs, since type inference can only be invoked by users on demand. In order to benefit from the advantages of type information, the user just adds the `:- compiler_options([xpp_on(typecheck)])` flag in his program.

We have tested our code on various frequently used XSB modules, and the results are shown in table 1. In all cases type inference was completed instantaneously. A signature was inferred for every predicate defined in these modules, and in all cases it was at most as specific as the user intended, with the exception of `assoc_xsb.P`. The reason is that one of the clauses that defines `is_assoc/4` does not respect the type signature that the coders had in mind; the first argument should be of the user-defined type `assoc`:

```
:- type assoc ---> t ; t(any,any,any,assoc,assoc).
```

whereas in one of the clauses it is defined, the `-` atom is used instead, which creates a mismatch between the two types. As a matter of fact, that particular clause would never succeed (the last call in its body is to `fail`), so that was a programming error caught by our type inference system! We then realized that `-` was used in other places as well, to capture empty pairs, which are otherwise represented by the signature `:- type pair(A,B) ---> A - B.`, so when we added the `-` as a valid type constructor for `pair`, we got more specific results.

Name	LOC	# Def[9]	# Inf[10]
`lib/pairlist.P`	125	10	10
`lib/lists.P`	515	59	59
`lib/ugraphs.P`	850	90	90
`lib/assoc_xsb.P`	566	49	48
`lib/ordsets.P`	422	39	39
`examples/tree1k.P`	2046	1	1

Table 1. Experimental results for various XSB modules

7 Conclusion and Future Work

In this paper, we described an extension that adds sybtyping capabilities to the Mycroft-O'Keefe type system and its implementation. We believe that the extended Mycroft-O'Keefe type system is closer to Prolog's natural behavior. Our implementation of the type checking and type inference mechanisms as a preprocessor makes the use of the type system easier and more appealing to the user, while preserving XSB's modules semantics.

However, there is significant work left to be done before claiming that we have developed a type system that follows Prolog's natural behavior completely. A particular issue that needs to be addressed is the one of handling type signatures imported from different XSB modules. Currently, declarations stating which predicates are exported from and imported in each module are stored in the corresponding `.H` files. These seem like a good choice to also store type signatures for predicates defined in modules. When a user imports a predicate from a module, the respective type signature will be considered by the system as declared.

One promising direction for future work seems to be the one of *dependent types*, where types are indexed by values of other types. Also, the addition of a user-defined subtyping relation may be helpful especially in programs implementing classifications and ontologies. Finally, due to space limitations, we have not discussed how the type system we developed can be used to type meta-predicates; we will refer to this specific aspect in a future paper.

References

1. Mycroft, A., O'Keefe, R.: A polymorphic type system for Prolog. Artificial intelligence **23**(3) (1984) 295–307
2. Siek, J., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop. (2006) 81–92
3. Schrijvers, T., Santos Costa, V., Wielemaker, J., Demoen, B.: Towards Typed Prolog. In: Proceedings of the 24th International Conference on Logic Programming. ICLP '08, Berlin, Heidelberg, Springer-Verlag (2008) 693–697
4. Schrijvers, T., Bruynooghe, M.: Towards constraint-based type inference with polymorphic recursion for functional and logic languages. In: Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages. (2005) 1–16
5. Barbuti, R., Giacobazzi, R.: A bottom-up polymorphic type inference in logic programming. Science of computer programming **19**(3) (1992) 281–313
6. Lu, L.: Polymorphic type analysis in logic programs by abstract interpretation. The Journal of Logic Programming **36**(1) (1998) 1–54
7. Sagonas, K., Swift, T., Warren, D.: Xsb: An overview of its use and implementation. SUNY at Stony Brook (1993)
8. Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3) (1978) 348–375
9. Siek, J., Vachharajani, M.: Gradual typing with unification-based inference. In: Proceedings of the 2008 symposium on Dynamic languages, ACM (2008) 7
10. Rastogi, A., Chaudhuri, A., Hosmer, B.: The ins and outs of gradual type inference. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (2012) 481–494
11. Dayan, P., Abbott, L.: Essential actionscript 3.0. (2007)
12. Sagonas, K., Luna, D.: Gradual typing of Erlang programs: A Wrangler Experience. In: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG. ERLANG '08, New York, NY, USA, ACM (2008) 73–82
13. Swift, T., Warren, D.S.: Tabling with answer subsumption: implementation, applications and performance. In: Proceedings of the 12th European conference on Logics in artificial intelligence. JELIA'10 (2010) 300–312

Syntactic integration of external languages in Prolog

Jan Wielemaker[1] and Nicos Angelopoulos[2]

[1] Web and Media group, VU University Amsterdam,
De Boelelaan 1081a,
1081 HV Amsterdam, The Netherlands,
J.Wielemaker@vu.nl
[2] Netherlands Cancer Institute,
Amsterdam,
The Netherlands,
n.angelopoulos@nki.nl

Abstract. Todays applications are typically programmed in multiple languages, using SQL to access databases, JavaScript to make the (web-based) user interface interactive, etc. Prolog can cooperate to this orchestra using two views: as a logic server component or as 'glue'. In this article we concentrate on the 'glue' view, which implies that we must be able to 'talk' the language of other components. In one scenario, code snippets, such as SQL queries or HTML output, are embedded as strings in the Prolog program. Using strings however is slow and creates insecure programs that are hard to debug. Alternatively, one can use Prolog terms to represent foreign 'objects'. This latter approach has been in use for a long while in the Prolog community. In this article, we give an overview of the design choices that are available and discuss their consequences based on our experience.

1 Introduction

A modern (Prolog) application often has to interact to the external world and this interaction is increasingly based on the exchange of messages with other languages. The oldest and most known example is SQL. To query a relational database we construct the text for an SQL query from a skeleton and additional (parameter) information. This SQL query is sent to the database and the result is translated into Prolog terms using, for example, an ODBC wrapper. Other examples are (a) web-based applications generating HTML and Javascript, (b) generating graphical representations of a graph by creating a description thereof in the *dot* language, using graphviz [3] to render this and (c) interfacing to the **R** language for statistical operations.

In many languages such tasks are performed by creating strings, often concatenated from smaller building blocks or using printf() like skeletons where details are filled from runtime parameters. For example, we can use the C code below to create an SQL query to find all persons that live in a given city.

```
sprintf(cmd, "SELECT * FROM Persons\n"
             "WHERE City='%s'", city);
```

This approach is problematic. It is at the heart of what is commonly referred to as *SQL injection*. Such an injection is established by exploiting syntax features such as quoting rules of the target language. For example, we can use the value `"xxx' OR City LIKE '*"` for city, getting results for all cities rather than just a single city. Another disadvantage of this approach is that creating, but especially parsing textual expressions is a costly process in terms of CPU usage. Both misinterpretation and the CPU overhead can be avoided if the target language provides a structured alternative to the textual interface (e.g., a binding to the C language). Finally, using strings that express messages in another language often leads to inelegant, hard to maintain programs that do not take advantage of the flexibility of Prolog's data representation.

Nevertheless, the text-based approach is popular for at least two reasons. First, it is easy to understand, especially for novices and second, it is more immediate and straightforward to implement at the start. It is often the case that such integrations are born by the necessity of specific projects, that want the job done in the minimum of time. There is rarely an interest within specific projects to create general, easy to use and secure interfaces.

Prolog is rather poor in embedding long strings (section 8), but it allows for creating a shadow 'DOM structure'[3] naturally and concisely and this structure can elegantly be serialized using DCGs or translated into a series of low-level calls to the API of a target language. Using this approach, injection attempts are prevented by a correct translation from the shadow DOM to the target language. Obviously, any OO language also allows building a shadow DOM structure, but this requires a verbose mess of 'new' and constructors. Even popular server platforms such as Java Server Pages (JSP) do not take this route.

The described examples have been published elsewhere. The contribution of this article is that it provides an overview of what we consider the main design options and it contains suggestions on how the Prolog syntax could be extended to accommodate a wider range of languages.

In the remainder of this article, we will describe a number of examples and our experience with these. In section 8 we summarise the relevant dimensions to consider when developing a Prolog interface to an external language.

2 SQL

Connecting Prolog to relational databases using SQL is extensively studied. Many Prolog systems contain a low-level database interface that allows for executing an SQL query represented as text and receive the results as a Prolog term, usually row-by-row on backtracking. This interface makes the complete power of SQL available to Prolog, together with the disadvantages of text-based interaction described in the introduction. Some interfaces, e.g., SWI-Prolog's **odbc_prepare/5**[4] allow for *prepared statements*.

[3] The term 'DOM structure' is precise for HTML/XML documents. Here, we use it in a more abstract sense. 'AST' (Abstract Syntax Tree) may be more appropriate for some of the target languages.

[4] `http://www.swi-prolog.org/pldoc/man?predicate=odbc_prepare/5`

Prepared statements are executed using their handle and parameters, avoiding SQL injections.

Christophe Draxler [2] designed a still widely used high-level interface to access relational databases. The core idea is to relate database tables to predicates. Next, a normal Prolog body-term created from these predicates and a subset of normal Prolog goals is translated into an SQL query. The advantage of this is that the Prolog programmer does not even have to know SQL to pose complex queries to the relational database. The disadvantage is that not all SQL features can be expressed in this language, for example it is not possible to create a new table through this interface. Also, Prolog's approach to access tuples by argument position is problematic for accessing tables with many columns.

An alternative approach is NED [5], where the users use Prolog terms as an abstract representation of a query, where the expressivity is tailored to fit context-specific requirements. The disadvantage of this type of approach is that it introduces a complete new language that needs to be documented and studied in detail, both for Prolog and database experts.

3 HTML

HTML (or XML or SGML) is a markup language rather than a programming language, i.e., it cannot be executed. HTML documents are a frequently used component in web-based applications. Although HTML itself is not executed and therefore the security consequences of invalid HTML may seem less severe than invalid SQL, HTML can embed executable 'script' elements that does turn it into a security risc. I.e., embedding uncontrolled text in HTML can inject code into the page that is executed by the browser.

One of the early implementations for processing HTML documents as Prolog terms is Ciao's PiLLow [4] library. SWI-Prolog's HTML generation library [10] is related, but exploits DCGs that allows the user to define components that can be reused in different HTML pages.

Dealing with these markup languages using Prolog terms works well because the underlying document model is a tree where each node is either plain text or is a node with an element name (e.g., `span, div, h1`), a set of attributes and an ordered list of child nodes. Although the concrete HTML syntax (e.g., `<span class="c1">Hello World</span>` cannot be turned into valid Prolog, the underlying datamodel can easily be expressed and the translation is so simple that it can be remembered easily by the user.

There are two fundamentally different approaches here. One is to represent a document as a Prolog term, e.g., `element`(*span, [class=c1], ['Hello World']*) and the other is to represent a document as a logic *program*, a set of clauses that represent the logical structure of the document. Roger Price describes four variations of the latter approach in [6]. This is a general tradeoff in representing data in Prolog, where terms are typically more suitable for volatile datastructures used within computations and programs can be considered a rule based representation of the world.

4 JavaScript and JSON

In addition to HTML, web applications often have to emit JavaScript. Larger and static pieces of JavaScript should probably be written in a separate JavaScript source file and be included using the HTML `script` element. Small code, such as code that initialises JavaScript objects based on the current document context, should be maintained together with the (Prolog) code that generates the web page. This implies we need a solution to embed JavaScript source code into Prolog.

If we could limit the JavaScript to calls with simple parameters, this is quite feasible and comparable to the Java, R and XPCE interfaces described in later sections of this article. Structured objects can be exchanged as JSON[5]. JSON is a simple serialisation syntax for structured objects. Its syntax is valid Prolog, but using this syntax has some drawbacks. All keys and string values are written using double quotes (e.g., `"name":"Joe"`), where atoms result in a much more compact representation. If we change this, using atoms for keys and string values, JSON constants such as `null` and `true` become ambiguous. JSON support is part of the SWI-Prolog HTTP library and consist of two layers. First, we define a Prolog representation for a JSON objects using the following rules:

- A JSON object is mapped to a term `json`(*NameValueList*), where NameValueList is a list of Name=Value. Name is an atom created from the JSON string.
- A JSON array is mapped to a Prolog list of JSON values.
- A JSON string is mapped to a Prolog atom.
- A JSON number is mapped to a Prolog number.
- The JSON constants true and false are mapped –like JPL, see section 6– to @(true) and @(false).
- The JSON constant null is mapped to the Prolog term @(null)

Here is a complete example in JSON and its corresponding Prolog term:

```
{ "name":"Demo term",
  "created": {
    "day":null,
    "month":"December",
    "year":2007
  },
  "confirmed":true,
  "members":[1,2,3]
}
```

[5] `http://www.json.org/`

```
json([ name='Demo term',
       created=json([day = @null,
                     month='December',
                     year=2007]),
       confirmed = @true,
       members=[1, 2, 3]
     ])
```

The predicates **atom_json_term/3**, **json_read/3** and **json_write/3** can be used to (de)serialise Prolog JSON terms. The second layer allows for defining a mapping between JSON terms and more commonly used Prolog representations. For example, a Prolog programmer would typically represent a point in a two-dimensional space using a term `point`(*5,10*). The typical JSON representation is `{"x":5,"y":10}`, possibly extended with a `"type":"point"` property. The `json_convert` library can convert between these two representations based on a declaration like this:

```
:- json_object
        point(x:integer, y:integer).
```

JavaScript allows for lambda functions, and these are commonly used to customise objects defined in reusable libraries. A common feature is to register event hooks as lambda functions with more abstract classes. Lambda functions are specified in the full C-style syntax of JavaScript, while at their role in the program commonly asks them to be specified from Prolog where the details (constants) depend on the application context.

Many aspects of this language are currently invalid in Prolog, in particular consider functions without arguments (e.g., `myfunction()`), The relation between function head and body (`function(x, y) {...}`), array subscripts and the ⟨*object*⟩.⟨*method*⟩ notation. Even if such as mapping is feasible, we are unsure of its value considering our experience with expressing imperative code using Prolog syntax in section 5. Possibly the problem is less severe here because the JavaScript code executes in another agent (typically a web browser).

An alternative approach suggested in private conversation by Maarten van den Dungen is to use Prolog directly, using JScriptLog`http://jlogic.sourceforge.net/` to execute code in the JavaScript environment. This solves the syntax issues, but the user must be aware that the other Prolog code executes on a different engine, seeing a different set of predicates, including a different set of built-in features.

5 XPCE

Although not widely known we introduce XPCE, SWI-Prolog's graphics subsystem, as an example of a wider class of language interface challenges. XPCE ([9], reworked version in [8]), is an object oriented system, that allows objects to be manipulated from Prolog through four basic predicates:

new(*-Reference, +Class(+Arg, ...)*)
send(*+Reference, +Method(+Arg, ...)*)
get(*+Reference, +Method(+Arg, ...), -Result*)
free(*+Reference*)

In these predicates, *Reference* is a term @/1 that identifies an XPCE object and *Arg* is a reference, number, atom, term `Class`(*+Arg, ...*) or `new`(*Class*). The latter is needed to create an instance from a class without arguments because `Class()` is not valid syntax.

The above predicates allow for manipulating objects. XPCE can, similarly to JavaScript, store executable code into object properties. Being designed for use with Prolog, executable code is expressed as objects and thus can be manipulated from Prolog using the same interface. Code objects include conjunction, disjunction, negation, arithmetic, higher order operations on collections, etc. Below is a simple example of a button that colours a box when clicked. Note that the outer 'message' refers to a property (method) of class button and the inner 'message' to a class name.

```
...
new(B, button('Click for red box')),
send(B, message(message(Box, colour, colour(red)))),
...
```

XPCE takes the integration one step further by supporting subclassing of XPCE classes from Prolog. To do this, we use a Prolog syntax to describe a new subclass and its methods. The method body can use the full power of Prolog, although it is executed as **once/1**. A class definition is compiled into an XPCE class with proxy methods that call the Prolog implementations of the methods.

XPCE is a special case because the language was designed for the way it is embedded in Prolog. Notably, XPCE does not have a concrete syntax and can only be 'programmed' by managing objects using the four principal interaction functions.

XPCE/Prolog is a powerful framework for building applications, but it has a steep learning curve. In part, this is unavoidable due to the sheer size of graphical libraries. Other parts of the complexity are caused by the fact that you have two ways to store data: the Prolog way and in XPCE objects. Each can be manipulated through a language. Both languages are expressed in the same syntax, but their different semantics is confusing to programmers.

6 Java (JPL)

JPL,[6] developed by Paul Singleton, uses an approach to access Java from Prolog that is similar to XPCE.[7] Java is disclosed through the JNI interface, providing Prolog with means to create Java data structures. Next, the interface uses the Java reflection API to

[6] `http://www.swi-prolog.org/packages/jpl/`

[7] In addition, JPL provides an interface to access Prolog from Java that is loosely based on the SWI-Prolog C++ interface. We consider that irrelevant to the discussion in this article.

invoke methods on Java objects by name. This can be compared to the four principal predicates of XPCE.

There are a large number of implementations that provide access to Java from Prolog. These use three ways to access Java: (1) JNI based, (2) network based (e.g., InterProlog[8]) or (3) Prolog (and its builtin predicates) are written in Java. The implementations we know use the Java reflection interface to access methods in Java.

Extending Java from Prolog by creating Java classes and methods is not provided by JPL. In theory, the class-extension mechanism used with XPCE/Prolog can be implemented by generating, compiling and loading Java code that implements the required proxy classes. This would allow Prolog programmers to extend Java class libraries without much knowledge of Java. Given that Java has a widely known syntax, it is doubtful that many people would like to create Java classes using Prolog syntax. Creating Java (proxy) classes from Prolog, where the methods are executed in Prolog may provide a promising alternative for making Prolog available from Java.

7 R

.

The R language for statistical computing provides a rich computational framework, a programming language with a functional flavour and library of primitives for graphics, mostly diagrams. The r..eal [1] Prolog to R [7] interface went through two iterations. The first realisation used R as a slave process with pure textual exchange of results. The second generation interface, r..eal, uses, like JPL and XPCE, a native interface. The system supports three basic interactions on the <− operator, as illustrated in the session below:

```
1 ?- [library(real)].
true.
2 ?- a <- [1,2,3].
true.
3 ?- <- a.
[1] 1 2 3
true.
4 ?- A <- a.
A = [1, 2, 3].
```

The first line assigns a Prolog list as a vector to an R variable. The second prints the result of an R expression and the last converts the result of an R expression into a Prolog term. The interface describes a mapping between R and Prolog terms. Vectors are mapped to flat lists, matrices to singly nested lists, R's named list to Prolog pair lists and function calls to term structures. Communication of large data structures is facilitated via an efficient low level interface based on the C language interface of the two systems. Calls to R functions result in translation of the Prolog term into a string that is evaluated by R.

[8] http://www.declarativa.com/interprolog/

Unlike with JavaScript, mapping from Prolog terms to R expressions works well due to the simple uniform functional syntax of R. The result is a powerful interface. However, it suffers from several issues, some fundamental and some less so:

- While primitive Prolog data is converted through the R interface to C, arbitrary terms are processed as text. This results in very different performance results. Consider the code below. In the first call we convert a Prolog list to an R array (no text involved) and in the second, we create a complete textual representation (`"mean(c(1,2...,100000))"`)

```
1 ?- numlist(1,100000,L), time((a<-L, M<-mean(a))).
% 60 inferences, 0.006 CPU ...
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
M = 50000.5.

2 ?- numlist(1,100000,L), time((M<-mean(L))).
% 2,088,940 inferences, 1.129 CPU ...
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
M = 50000.5.
```

- R variables are global and can be destructively assigned to. While this provides the -often wanted- global variables to the Prolog/R infrastructure, it can be confusing the Prolog programmers.
- The syntax mapping requires some tweaks to avoid ambiguity. These include the use of dot character in names and the empty argument list notation. The interface circumvents these by adding a '.' in both cases. Thus `var.name` becomes `var..name` and `foo()` translates to `foo(.)`. The integration would be enhanced if the original code can be handled from within Prolog. Another difficulty arises from the syntax commonly used in many languages for accessing arrays. `a[i,j]` is invalid in Prolog. R..eal uses a caret operator to legalise the syntax, for example "`a^[i,j]`".

8 Criteria for designing an interface

Above, we discussed several examples that make external languages accessible from Prolog. Although we cannot claim that the presented solutions are the best possible, all these systems have gone through multiple iterations to reach at their current design.

All the described interfaces represent 'objects' of the target language using Prolog terms. In part, this may be influenced by Prolog's poor abilities to represent text, which is limited by the following:

- Lack of a long string (as, for example, Python's `"""..."""` syntax). This makes it hard embed strings that require quotes in a Prolog text. Note that the ISO Prolog standard allows both for doubling quotes to escape them and the backslash notation, which complicates the introduction of long strings.
- Complicated multi-line string syntax. Line-breaks need to be escaped with a backslash and the next line must start at the left margin to avoid additional white-space in the output.

– No simple concatenation syntax, neither between constant string fragments nor with number and other constructs.
– Representing text as atoms is limited by atom-length in many implementations. Representing text as strings wastes space on most implementations.

8.1 Choosing a Prolog syntax for the target language

We distinguish several options for representing the target language in Prolog:

– If the target language is (almost) valid Prolog syntax, using Prolog syntax that is either equivalent to the target language or introduces only small easy to understand systematic transformations, is an attractive option. This approach is easy to learn and does not require extensive documentation. The r..eal interface described in section 7 is an example. Additional operator declarations can help bridging the gap. Commonly encountered problematic syntax features are the empty parameter list, represented as "`()`", array subscripts (e.g., `a[1]`), uppercase identifiers, distinguishing quoted strings from symbols (JSON) and the dot (.) being used in identifiers or to separate methods from objects.
– In some cases, the syntax is complicated, but the datamodel is simple and can easily be represented in Prolog. The discussed markup languages (section 3) form a good example.
– In some cases, we can represent a fair deal of the intended semantics using our familiar Prolog language and compile Prolog to the target language. This is the case for the SQL interface by Christophe Draxler [2] (see section 2). His SQL mapping results in a uniform syntax and semantics at the price of not being able to access *all* functionality of the target language. For example, it is not possible to create a new table through this interface.

Fortunately, many special purpose languages can be handled by one or more of the above options. An exception is JavaScript, which has a complicated C-like syntax. Inventing a Prolog datastructure that captures all functionality of the language and is easy to understand by a Prolog programmer with some JavaScript knowledge is probably not feasible.

8.2 How does the semantics of target language relate to Prolog?

Relation between the semantics of both systems is also important. We see that a close semantic relation (SQL) can be used to find an alternative to a syntactic mapping. We also see that strong syntactic integration between two systems that have large semantic differences (R, XPCE) can lead to confusion.

8.3 Technical integration

In many of the above described examples (SQL, HTML, JavaScript), we must ultimately produce a string serialisation for the target language. In others (XPCE, Java, R) generation of text can be avoided. In other words, the target language can be linked into the same process and the systems can communicate using low-level communication that typically uses the C language as intermediate.

9 Discussion

Many of today's applications are built from components that use different programming languages and markup languages. There are roughly two ways in which we can make Prolog play in this orchestra: (1) as a logic component or (2) as 'glue', making Prolog talk to multiple different components. Some systems (e.g., Amzi!) concentrate explicitly on a minimal role as 'Logic Server'. Most system make no clear choice and SWI-Prolog probably represents the most extreme view in pushing Prolog as a 'glue' language.

Composing text of another language using simple string manipulation is often undesirable because the actual goal is to create structured objects in the target environment (let it be data structures or executable objects). Simple composition of text fragments easily leads to syntax errors that are hard to detect in the development environment or, worse, security vulnerabilities (e.g., SQL injections).

As we have seen, Prolog is rather poor in representing (long) textual expressions, but it is good in using concise syntax to build complex datastructures of which the syntax can be optimised using operators. This approach works for languages with a uniform syntax that is easily expressed (R), languages with a simple datamodel (HTML) or languages that are semantically close, such that we can translate a subset of pure Prolog into the target language (SQL).

Avoiding limits in Prolog supports this approach, i.e., unbounded integers can comfortably represent integers from any language. Compound terms with unrestricted arity allows representing any table row or array as a compound term and atoms with unlimited length and Unicode characters can represent any string. These features can all be implemented within the ISO standard.

It is feasible to extend Prolog syntax to improve the transparency of these language mappings, avoiding the need for workarounds such as discussed in section 7. In particular, we lack the commonly found syntactic constructs described below. If we can find an elegant way to add these constructs, more languages can be handled using a syntax that stays close to the syntax of the target language, improving readability and reducing the need to document unnatural changes to the syntax of the target language.

name()
: This is currently a syntax error in Prolog. Adding it introduces some ambiguities in e.g., **functor/3**. Otherwise, **atom/1** can fail on this and **compound/1** can succeed.

a[x]
: *Array syntax* This is already supported by e.g., B-Prolog.[9] In B-Prolog, `X[I1,...,In]` is a shorthand for `X^[I1,...,In]`.

name(arg...) { **body** }
: It is unclear how to turn this into valid syntax, but currently it is always a syntax error because neither a compound term, nor a {...} term can be an operator and the Prolog syntax does not allow for a sequence of two non-operator terms. This implies that it is possible to make this term legal syntax without introducing compatibility issues with the current Prolog standard and, like the array subscript notation discussed above, map it to a Prolog term.

[9] `http://www.probp.com/manual/node47.html`

a.b

'.'-separated sequences These can in part be supported by defining '.' as an infix operator, but this creates ambiguous syntax with floating point numbers. Handling the dot as an operator causes such identifiers to show up as lists, which may easily lead to ambiguities. Possibly, `a.b` can be handled at the tokenization level. For example, it could be a valid unquoted atom.

Acknowledgements

This paper is a reflection based on a long experience with connecting Prolog with other languages. The described interfaces where developed in several iterations with extensive feedback from early users. We particularly would like to than Anjo Anjewierden for this pioneering work on XPCE and Paul Singleton for his work on JPL and comments on a draft of this article.

References

1. Nicos Angelopoulos, Vitor Santos Costa, Jan Wielemaker, Joao Azevedo, Rui Camacho, and Lodewyk Wessels. R..eal: A library for statistical AI. Technical report, Netherlands Cancer Institute, 2012.
2. Christoph Draxler. Accessing relational and higher databases through database set predicates in logic programming languages. Phd thesis, Zurich University, 1991.
3. John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. Graphviz open source graph drawing tools. In Petra Mutzel, Michael Jnger, and Sebastian Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 594–597. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45848-4_57.
4. Daniel Cabeza Gras and Manuel V. Hermenegildo. Distributed WWW programming using (ciao-)prolog and the piLLoW library. *TPLP*, 1(3):251–282, 2001.
5. Frederick Maier, Donald Nute, Walter D. Potter, Jin Wang, Mayukh Dass, Hajime Uchiyama, Mark J. Twery, Peter Knopp, Scott Thomasma, and H. Michael Rauscher. Efficient integration of prolog and relational databases in the ned intelligent information system. In Hamid R. Arabnia, editor, *IKE*, pages 364–369. CSREA Press, 2003.
6. Roger Price. No more me too - different approaches to logic documents. In *Proceedings of 2nd International Workshop on Logic Programming Tools for Internet Applications*, 1997.
7. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Found. for Stat. Comp., Vienna, Austria, 2012.
8. Jan Wielemaker. *Logic programming for knowledge-intensive interactive applications*. PhD thesis, University of Amsterdam, 2009. http://dare.uva.nl/en/record/300739.
9. Jan Wielemaker and Anjo Anjewierden. An architecture for making object-oriented systems available from prolog. In *WLPE*, pages 97–110, 2002.
10. Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. Swi-prolog and the web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.