Dietmar Seipel, Michael Hanus, Salvador Abreu (Eds.)

 Declare 2017 – Conference on Declarative Programming

21st International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2017)
and
31st Workshop on Logic Programming (WLP 2017)
and
25th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2017)

Würzburg, Germany, September 2017
Proceedings

Technical Report 499
Institute of Computer Science
University of Würzburg
September 2017
Preface


Declarative programming is an advanced paradigm for modeling and solving complex problems. This method has attracted increased attention over the last decades, e.g., in the domains of data and knowledge engineering, databases, artificial intelligence, natural language processing, modeling and processing combinatorial problems, and for establishing knowledge–based systems for the web.

The conference Declare 2017 aims to promote the cross-fertilizing exchange of ideas and experiences among researches and students from the different communities interested in the foundations, applications, and combinations of high-level, declarative programming and related areas. It was be accompanied by a one–week summer school on Advanced Concepts for Databases and Logic Programming for students and PhD students.

The INAP conferences provide a forum for intensive discussions of applications of important technologies around logic programming, constraint problem solving and closely related advanced software. They comprehensively cover the impact of programmable logic solvers in the internet society, its underlying technologies, and leading edge applications in industry, commerce, government, and societal services. Previous INAP conferences have been held in Japan, Germany, Portugal, and Austria.

The Workshops on Logic Programming are the annual meeting of the Society for Logic Programming (GLP e.V.). They bring together researchers interested in logic programming, constraint programming, and related areas like databases and artificial intelligence. Previous workshops have been held in Germany, Austria, Switzerland, and Egypt.

The topics of the papers of this year’s joint conference concentrate on three currently important fields: constraint programming and solving, functional and logic programming, and declarative programming.

Declarative programming is a paradigm that expresses the logic of a computation in an abstract way. Thus, the semantics of a declarative language becomes easier to grasp for domain experts. Declarative programming offers many advantages for data and knowledge engineering, such as, e.g., security, safety, and shorter development time. During the last couple of years, a lot of research has been conducted on the usage of declarative systems in areas like answer set programming, reasoning, meta–programming, and deductive databases. Reasoning about knowledge wrapped in rules, databases, or the Semantic Web allows to explore interesting hidden knowledge. Declarative techniques for the transfor-
Information, deduction, induction, visualisation, or querying of knowledge have the advantage of high transparency and better maintainability compared to procedural approaches.

Many problems which occur in large industrial tasks are intractable, invalidating their solution by exact or even many approximate constructive algorithms. One approach which has made substantial progress over the last few years is the constraint programming paradigm. Its declarative nature offers significant advantages, from a software engineering standpoint, in the specification, implementation and maintenance phases. Several interesting aspects are in discussion: how can this paradigm be improved or combined with known, classical methods; how can real-world situations be modelled as constraint problems; what strategies may be pursued to solve a problem, once it’s been specified; or what is the experience of applications in really large industrial planning, simulation and optimisation tasks?

Another area of active research is the use of declarative programming languages, in particular, functional and logic languages, to implement more reliable software systems. The closeness of these languages to logical models provides new methods to test and verify programs. Combining different programming paradigms is beneficial from a software engineering point of view. Therefore, the extension of the logic programming paradigm and its integration with other programming concepts are active research branches. The successful extension of logic programming with constraints has been already mentioned. The integration of logic programming with other programming paradigms has been mainly investigated for the case of functional programming so that types, modules, higher-order operators, or lazy evaluation can also be used in logic-oriented computations.

The three conferences INAP 2017, WLP 2017, and WFLP 2017 were jointly organized at the University of Würzburg, Germany, by the University of Würzburg and the Society for Logic Programming (GLP e.V.). We would like to thank all authors who submitted papers and all conference participants for the fruitful discussions. We are grateful to the members of the programme committee and the external referees for their timely expertise in carefully reviewing the papers, and we would like to express our thanks to the University of Würzburg for hosting the conference at the new Central Lecture Building Z6.

September 2017 Dietmar Seipel, Michael Hanus, Salvador Abreu
Organization

Program Chair

Dietmar Seipel  
University of Würzburg, Germany

Program Commitee of INAP

Slim Abdennadher  
German University of Cairo, Egypt
Salvador Pinto Abreu  
University of Évora, Portugal (Co-Chair)
Molham Aref  
Logic Blox Inc, Atlanta, USA
Chitta Baral  
Arizona State University, Tempe, USA
Joachim Baumeister  
University of Würzburg
Stefan Brass  
University of Halle, Germany
François Bry  
Ludwig-Maximilian University of Munich, Germany
Philippe Codognet  
UPMC, Paris, France
Vitor Santos Costa  
University of Porto, Portugal
Agostino Dovier  
University of Udine, Italy
Thomas Eiter  
Vienna University of Technology, Austria
Thom Frühwirth  
University of Ulm, Germany
Parke Godfrey  
York University, Toronto, Canada
Gopal Gupta  
UT, Dallas, USA
Jorge Lobo  
ICREA and Universitat Pompeu Fabra, Barcelona, Spain
Grzegorz J. Nalepa  
AGH University, Kraków, Poland
Vitor Nogueira  
University of Évora, Portugal
Enrico Pontelli  
New Mexico State University, Las Cruces, USA
Dietmar Seipel  
University of Würzburg, Germany (Chair)
Hans Tompits  
Vienna University of Technology, Austria
Masanobu Umeda  
Kyushu Institute of Technology, Japan
Program Committee of WLP / WFLP

Slim Abdennadher                German University in Cairo, Egypt
Sergio Antoy                    Portland State University, USA
Olaf Chitil                     University of Kent, UK
Jürgen Dix                      Clausthal University of Technology, Germany
Moreno Falaschi                 Università di Siena, Italy
Michael Hanus                   Kiel University, Germany (Chair)
Sebastian Joosten               University of Innsbruck, Austria
Oleg Kiselyov                   Tohoku University, Japan
Herbert Kuchen                  University of Münster, Germany
Sibylle Schwarz                 HTWK Leipzig, Germany
Dietmar Seipel                  University of Würzburg, Germany
Tom Schrijvers                  KU Leuven, Belgium
Martin Sulzmann                 Karlsruhe University of Applied Sciences, Germany
Hans Tompits                    Technical University of Vienna, Austria
German Vidal                    Universidad Politécnica de València, Spain
Janis Voigtländer               University of Duisburg–Essen, Germany
Johannes Waldmann               HTWK Leipzig, Germany

Local Organization

Dietmar Seipel                  University of Würzburg, Germany
Falco Nogatz                    University of Würzburg, Germany

External Referees for INAP and WLP / WFLP

Zhuo Chen                       UT, Dallas, USA
Daniel Gall                     University of Ulm, Germany
Falco Nogatz                    University of Würzburg, Germany
# Table of Contents

**Preface** .......................................................... III

**Constraints**

Constraint Solving on Hybrid Systems .............................. 3  
*Pedro Roque, Vasco Pedro, Salvador Abreu*

Run-time Analysis of Temporal Constrained Objects .............. 19  
*Jinesh M Kannimoola, Bharat Jayaraman, Krishnashree Achuthan*

Implementation of Logical Retraction in Constraint Handling Rules  
with Justifications ................................................. 35  
*Thom Frühwirth*

The Proportional Constraint and Its Pruning ....................... 50  
*Armin Wolf*

A Confluence Checker for Constraint Handling Rules with Persistent  
Constraints ............................................................ 60  
*Frank Richter, Daniel Gall, Thom Frühwirth*

An Operational Semantics for Constraint-logic Imperative Programming . 76  
*Jan C. Dageförde, Herbert Kuchen*

Hypertree Decomposition: The First Step Towards Parallel Constraint  
Solving .............................................................. 91  
*Ke Liu, Sven Loeffler, Petra Hofstedt*

**Declarative Systems**

An Approach for Representing Answer Sets in Natural Language . . . . . . 107  
*Min Fang, Hans Tompits*

Techniques for Efficient Lazy-Grounding ASP Solving ............... 123  
*Lorenz Leutgeb, Antonius Weinzierl*

The Syllogistic Reasoning Task: Reasoning Principles and Heuristic  
Strategies in Modeling Human Clusters ............................. 139  
*Emmanuelle-Anna Dietz Saldanha, Steffen Hölldobler, Richard Mörbitz*
In Praise of Impredicativity: A Contribution to the Formalisation of Meta-Programming ........................................... 155
François Bry

An Abstract Machine for Push Bottom-Up Evaluation with Declarative Output .................................................. 171
Stefan Brass

Extracting and Representing Entities from Open Sources of Information in the Agatha Project .......................... 187
Gonçalo Carnaz, Roy Bayot, Vítor Beires Noqueira, Teresa Gonçalves, Paulo Quaresma

Functional and Logic Programming

Concolic Testing of Functional Logic Programs ...................... 201
Jan Rasmus Tikovsky

CalcuList: a Functional Language Extended with Imperative Features .... 217
Domenico Saccà, Angelo Furfaro

CPM: A Declarative Package Manager with Semantic Versioning ........ 233
Michael Hanus, Jonas Oberschweiber

Declarative XML Schema Validation with SWI-Prolog ................ 242
Falco Nogatz, Jona Kalkus, Dietmar Seipel

plspec – A Specification Language for Prolog Data ............... 252
Philipp Körner, Sebastian Krings

Closed Types for Logic Programming ............................... 268
João Barbosa, Mário Florido, Vítor Santos Costa

Controlling LEGO® EV3 robots with Prolog ....................... 284
Sibylle Schwarz, Mario Wenzel

How I Teach Functional Programming .............................. 292
Johannes Waldmann

Author Index ................................................................. 308
Constraints
Constraint Solving on Hybrid Systems

Pedro Roque, Vasco Pedro, and Salvador Abreu

Universidade de Évora/LISP
d11735@alunos.uevora.pt, vp@di.uevora.pt, spa@di.uevora.pt

Abstract. Applying parallelism to constraint solving seems a promising approach and it has been done with varying degrees of success. Early attempts to parallelize constraint propagation, which constitutes the core of traditional interleaved propagation and search constraint solving, were hindered by its essentially sequential nature. Recently, parallelization efforts have focused mainly on the search part of constraint solving, as well as on local-search based solving. Lately, a particular source of parallelism has become pervasive, in the guise of GPUs, able to run thousands of parallel threads, and they have naturally drawn the attention of researchers in parallel constraint solving.

We address challenges faced when using multiple devices for constraint solving, especially GPUs, such as deciding on the appropriate level of parallelism to employ, load balancing and inter-device communication, and present our current solutions.

Keywords: Constraint solving, Parallelism, GPU, Intel MIC, Hybrid systems

1 Introduction

Constraint Satisfaction Problems (CSPs) allow modeling problems like the Costas Array problem [6], and some real-life problems like planning and scheduling [2], resources allocation [7] and route definition [3].

CPU’s parallelism is already being used with success to speed up the solving processes of harder CSPs [5,16,19,21]. However, very few constraint solvers contemplate the use of GPUs. In fact, Jenkins et al. recently concluded that the execution model and the architecture of GPUs are not well suited to computations displaying irregular data access and code execution patterns such as backtracking search [10].

We are currently developing a constraint solver named Parallel Heterogeneous Architecture Toolkit (PHACT) that is already capable of achieving state-of-the-art performances on multi-core CPUs, and can also speed up the solving process by adding GPUs and processors like Intel Many Integrated Cores (MICs\(^1\)) to solve the problems.

\(^1\) Intel MICs are coprocessors that combine many Intel processor cores onto a single chip with dedicated RAM.
The next section introduces the main CSP concepts and Section 3 presents some related work. Section 4 describes the architecture of PHACT, and in Section 5 the results achieved with PHACT, when solving some CSPs on multiple combinations of devices and when compared with some state-of-the-art solvers, are displayed and discussed. Section 6 presents the conclusions and directions for future work.

2 CSPs concepts

A CSP can be briefly described as a set of variables with finite domains, and a set of constraints between the values of those variables. The solution of a CSP is the assignment of one value from the respective domain to each one of the variables, ensuring that all constraints are met.

For example, the Costas Array problem consists in placing \( n \) dots on a \( n \times n \) matrix such that each row and column contain only one dot and all vectors between dots are distinct. It can be modeled as a CSP with \( n + n(n - 1)/2 \) variables, \( n \) of which correspond to the dots and each one is mapped to a different matrix column. The domain of these \( n \) variables is composed by the integers that correspond to the matrix rows where each dot may be placed. The remaining \( n(n - 1)/2 \) variables constitute a difference triangle, whose rows cannot contain repeated values [6].

The methods for solving CSPs can be categorized as incomplete or complete. Incomplete solvers do not guarantee that an existing solution will be found, being mostly used for optimization problems and for large problems that would take too much time to fully explore. Incomplete search is beyond the scope of this paper and will not be discussed here. On the contrary, complete methods guarantee that if a solution exists, it will be found.

3 Related work

Searching for CSP solutions in a backtracking approach can be represented in the form of a search tree. To take advantage of parallelism this search tree may be split into multiple subtrees and each one of them explored in a different thread that may be running on a different core, device or machine. This is the approach generally found in parallel constraint solvers, which run on single or distributed multi-core CPUs [5,16,19,21].

Pedro developed a CSP solver named Parallel Complete Constraint Solver (PaCCS) capable of running from a single core CPU to multiple multi-core CPUs in a distributed system [16]. Distributing the work among the threads through work stealing techniques and the Message Passing Interface (MPI) to allow communication between them, this solver achieved almost linear speedups for most of the problems tested.

Régis et al. implemented an interface responsible for decomposing an initial problem into multiple sub-problems, filtering out those found to be inconsistent [20]. After generating the sub-problems it creates multiple threads, each
one corresponding to an execution of a solver (e.g., Gecode [22]), to which a
sub-problem is sent at a time for exploration.

For some optimization and search problems, where the full search space is
explored, these authors achieved average gains of 13.8 and 7.7 against a sequen-
tial version, when using Gecode through their interface or just Gecode, respec-
tively [20]. On their trials, the best results were achieved when decomposing the
initial problem into 30 sub-problems per thread and running 40 threads on a
machine with 40 CPU cores.

While solving CSPs through parallelization has been a subject of research for
decades, the usage of GPUs for that purpose is a recent area, and as such there
aren’t many published reports of related work. To our knowledge, there are only
two published papers related with constraint solving on GPUs [1,4]. From these
two, only Campeotto et al. presented a complete solver [4].

Campeotto et al. developed a CSP solver with Nvidia’s Compute Unified
Device Architecture (CUDA), capable of using simultaneously a CPU and an
Nvidia GPU to solve CSPs [4]. On the GPU, this solver implements an approach
different from the one mentioned before, namely, instead of splitting the search
tree over multiple threads, it splits each constraint propagation over multiple
threads. Constraints relating many variables are propagated on the GPU, while
the remaining constraints are filtered sequentially by the CPU. On the GPU, the
propagation and consistency check for each constraint is assigned to one or more
blocks of threads according to the number of variables involved. The domain of
each variable is filtered by a different thread.

Campeotto et al. reduced the data transfer to a minimum by transferring
to the GPU only the domains of the variables that weren’t labeled yet and the
events generated during the last propagation. Events identify the changes that
happened to a domain, like becoming a singleton or having a new maximum
value, which allows deciding on the appropriate propagator to apply.

Campeotto et al. obtained speedups of up to 6.61, with problems like the
Langford problem and some real problems such as the modified Renault prob-
lem [11], when comparing a sequential execution on a CPU with the hybrid
CPU/GPU version.

4 Solver architecture

PHACT is a complete solver, capable of finding a solution for a CSP if one exists.
It is meant to be able to use all the (parallel) processing power of the devices
available on a system, such as CPUs, GPUs and MICs, to speed up solving
constraint problems.

The solver is composed of a master process which collects information about
the devices that are available on the machine, such as the number of cores and
the type of device (CPU, GPU or MIC), and calculates the number of sub-
search spaces that will be created to distribute among that devices. For each
device there will be one thread (communicator) responsible for communicating
with that device, and inside each device there will be a range of threads (search
engines) that will perform labeling, constraint propagation and backtracking on one sub-search space at a time. The number of search engines that will be created inside each device will depend on the number of cores and type of that device, and may vary from 8 on a Quad-core CPU to more than 100,000 on a GPU.

PHACT may be used to count all the solutions of a given CSP, to find just one solution or a best one (for optimization problems).

**Framework**

PHACT is implemented in C and OpenCL [13], which allows its execution on multiple types of devices from different vendors and the capability of being executed on Linux or on Microsoft Windows.

We present some OpenCL concepts, in order to better understand PHACT’s architecture:

- **Compute unit** One or more processing elements and their local memory. In Nvidia GPUs each Streaming Multiprocessor (SM) is a compute unit. AMD GPUs have their own components called Compute Units that match this definition. For CPUs and MICs, the number of available compute units is normally equal to or higher than the number of threads that the device can execute simultaneously [13];
- **Kernel** The code that will be executed on the devices;
- **Work-item** An instance of the kernel (thread);
- **Work-group** Composed of one or more work-items that will be executed on the same compute unit, in parallel. All work-groups for one kernel on one device have the same number of work-items;
- **Host** CPU where the application responsible for managing the execution of the kernels is run;
- **Device** A device where the kernels are executed (CPU, GPU, MIC).

In the implementation described here, the master process and the threads responsible for communicating with the devices run on the OpenCL host and the search engines run on the devices. The OpenCL host may also be treated as a device, in which case it will be simultaneously controlling and communicating with the devices and running search engines. Each search engine corresponds to a work-item, and all work-items execute the same kernel code, which implements the search engine.

**Search space splitting and work distribution**

For distributing the work between the devices, PHACT splits the search space into multiple sub-search spaces. Search-space splitting is effected by partitioning the domains of one or more of the variables of the problem, so that the resulting sub-search spaces partition the full search space. The number and the size of the sub-search spaces thus created depend on the number of work-items which will be used, and may go up to a few millions.
Example 1 shows the result of splitting the search space of a CSP with three variables, $V_1$, $V_2$ and $V_3$, all with domain \{1, 2\}, into 4 sub-search spaces, $SS_1$, $SS_2$, $SS_3$ and $SS_4$.

**Example 1.** Creation of 4 sub-search spaces.

\[
SS_1 = \{V_1 = \{1\}, V_2 = \{1\}, V_3 = \{1, 2\}\} \\
SS_2 = \{V_1 = \{1\}, V_2 = \{2\}, V_3 = \{1, 2\}\} \\
SS_3 = \{V_1 = \{2\}, V_2 = \{1\}, V_3 = \{1, 2\}\} \\
SS_4 = \{V_1 = \{2\}, V_2 = \{2\}, V_3 = \{1, 2\}\}
\]

Since each device will have multiple search engines running in parallel, the computed partition is organized into blocks of contiguous sub-search spaces that will be handled by each device, one at a time. The number of sub-search spaces that will compose each block will vary along the solving process and depends on the performance of each device on solving the current problem.

The communicator threads running on the host launch the execution of the search engines on the devices, hand each device one block of sub-search spaces to explore, and coordinate the progress of the solving process as each device finishes exploring its assigned block. The coordination of the devices consists in assessing the state of the search, distributing more blocks to the devices, signaling to all the devices that they should stop (when a solution has been found and only one is wanted), or updating the current bound (in optimization problems).

**Load balancing**

An essential aspect to consider when parallelizing some task is the balancing of the work between the parallel components. Creating sub-search spaces with balanced domains, when possible, is no guarantee that the amount of work involved in exploring each of them is even similar. To compound the issue, we are dealing with devices with differing characteristics and varying speeds, making it even harder to statically determine an optimal, or even good, work distribution.

Achieving effective load balancing between devices with such different architectures as CPUs and GPUs is a complex task [10]. When trying to implement dynamic load balancing, two important OpenCL limitations arise, namely when a device is executing a kernel it is not possible for it to communicate with other devices [8], and the execution of a kernel can not be paused or stopped. Hence, techniques like work stealing [5,17], which requires communication between threads, will not work with kernels that run independently on different devices and load balancing must be done on the host side.

To better manage the distribution of work, the host could reduce the amount of work it sends to the devices each time, by reducing the number of sub-search spaces in each block. This would make the devices synchronize more frequently on the host and allow for a finer control over the behavior of the solver. When working with GPUs, though, the number and the size of data transfers between the devices and the host should be as small as possible, because these are very
time consuming operations. So, a balance must be struck between the workload of the devices and the amount of communication needed.

PHACT implements a dynamic load balancing technique which adjusts the size of the blocks of sub-search spaces to the performance of each device solving the current problem, when compared to the performance of the other devices.

Initially each device $d$ explores two small blocks of sub-search spaces to get the average time, $\text{avg}(d)$, it needs to explore one sub-search space. The size of those blocks may be distinct among devices as it is calculated according to the number of threads that each device is capable of running simultaneously and its clock speed. When two or more devices finish exploring those first two blocks, their rank, $\text{rank}(d)$ is calculated according to Equation (1), where $m$ is the total number of devices.

\[
\text{rank}(d) = \frac{1}{\text{avg}(d)} \cdot \frac{1}{\sum_{i=1}^{m} \frac{1}{\text{avg}(i)}}, \quad \text{avg}(i) > 0
\]  

(1)

The rank of a device consists of a value between 0 and 1, corresponding to the relative speed of the device against all the devices that were used for solving a block of sub-search spaces. Faster devices will get a higher rank than slower devices, and the sum of the ranks of all the devices will be 1. The rank is then used to calculate the size of the next block of sub-search spaces to send to the device, by multiplying its value by the number of sub-search spaces that are yet to be explored.

Since the size of the first two blocks of sub-search spaces explored by each device is small, to prevent slow devices from dominating the solving process, it often only allows for a rough approximation of the speed of a device. So, in the beginning, only 1/3 of the remaining sub-search spaces are considered when computing the size of the next block to send to a device.

For the first device to finish its first two blocks, it will not be possible to calculate its rank, as it would need the average time of at least one more device. In this case, that device will get a new block with twice the size of the previous ones, as this device is probably the fastest device solving the current problem.

As search progresses, every time a device finishes exploring another block, its average time and rank are updated. The value of the average time of a device is the result of dividing the total time that the device was exploring sub-search spaces by the total number of sub-search spaces that it explored already.

As the rank value stabilizes, the size of the new block of sub-search spaces for the device will be the corresponding percentage from all unexplored sub-search spaces. Table 1 exemplifies the calculation of the number of search spaces that will compose the block of search spaces which will be sent for each device as soon as each of them finishes its previous block. This is repeated until a device waiting for work is estimated to need less than one second to solve all the remaining sub-search spaces, in which case it will be assigned all of them.

\[^2\text{If a device takes less than one second to explore a block of search spaces, most of that time was spent communicating with the host and initializing its data structures.}\]
Table 1. Example of the calculation of blocks size when using three devices.

<table>
<thead>
<tr>
<th>Device</th>
<th>Average time per search space (ms)</th>
<th>Rank</th>
<th>Remaining sub-search spaces to explore</th>
<th>Size of the next block of sub-search spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device 1</td>
<td>0.00125</td>
<td>0.625</td>
<td>1233482</td>
<td>770926</td>
</tr>
<tr>
<td>Device 2</td>
<td>0.00236</td>
<td>0.331</td>
<td>462556</td>
<td>153106</td>
</tr>
<tr>
<td>Device 3</td>
<td>0.01782</td>
<td>0.044</td>
<td>309450</td>
<td>13616</td>
</tr>
</tbody>
</table>

Another challenge GPUs pose is that they achieve the best performance when running hundreds or even thousands of threads simultaneously. But to use that level of parallelism, they must have enough work to keep that many threads busy. Otherwise, when a GPU receives a block with less sub-search spaces than the number of threads that would allow it to achieve its best performance, the average time needed to explore one sub-search space increases sharply.

For example the Nvidia GeForce GTX 980M takes about 1.1s to find all the solutions for the n-Queens 13 when splitting the problem in 742,586 sub-search spaces, and 2.4s when split in only 338 sub-search spaces. This challenge is also valid for CPUs, but not so problematic due to their lesser degree of parallelism when compared with the GPUs.

To overcome that challenge, sub-search spaces may be further divided inside a device, by applying a multiplier factor \( m \) to the size of a block and turning a block of sub-search spaces into a block with \( m \) times the original number of sub-search spaces, that will be created as presented in Example 1.

Communication

To reduce the amount of data that is transferred to each device, all of them will receive the full CSP, that is, all the constraints, variables and their domains, at the beginning of the solving process. Afterwards, when a device must be instructed to solve a new block of sub-search spaces, instead of sending all the sub-search spaces to the device, only the information needed to create those sub-search spaces is sent.

If a device is to solve sub-search spaces \( SS_2 \) and \( SS_3 \) from Example 1, it will receive the information that the tree must be expanded down to depth 2, that the values of the first variable are repeated 2 times and the values of the second variable are repeated 1 time only (not repeated). With this information the device will know that the values of the first variable are repeated 2 times, so the third sub-search space (\( SS_3 \)) will get the second value of that variable, and so on to the expansion depth. The values of the variables that were not expanded are simply copied from the original CSP that was passed to the devices at the beginning of the solving process.

Each time a work-item needs a new sub-search space to explore, it increases by one the number of the first/next sub-search space that is yet to be explored on that device and creates the sub-search space corresponding to the number before being increased. Then it will do labeling, propagation and backtracking until either all the sub-search spaces of that block have been explored, when
all the solutions must be found, or only one solution is wanted and one of the work-items on that device finds a solution.

PHACT represents the variable domains as 32-bit bitmaps, multiples of 64-bit bitmaps, or as (compact) intervals. When using intervals, PHACT is slower than when using bitmaps, but intervals are meant to be used instead of larger bitmaps on systems where the size of the RAM is an issue.

Implementation details

Several tests were made to find the best number of work-groups to use for each type of device. It was found that for CPUs and MICs the best results were achieved with the same number of work-groups as the amount of compute units of the device. For GPUs, the predefined number of work-groups is 4096 due to the increased level of parallelism allowed by this type of devices.

The user can specify how many sub-search spaces must be created or let PHACT estimate that number. If PHACT must estimate the number of sub-search spaces that will be generated, it will sum all the work-items that will be used in all the devices and multiply that value by 40 if all the solutions must be found for the current CSP, or by 100 if only one solution is required or when solving an optimization problem. After several tests these values (40 and 100) were found as allowing to achieve a good load balancing between the devices, and as such they are the predefined values.

When looking for just one solution or optimizing, the amount of work sent to each device is reduced by generating more sub-search spaces and decreasing the size of the blocks sent to the devices, which makes each one of them faster to explore, to make sure all the devices are synchronized on the host more frequently.

As for the number of work-items per work-group, CPUs and MICs are assigned one work-item per work-group, as their compute units can only execute one thread at a time.

On the contrary, each GPU compute unit can execute more than one thread simultaneously. For example, the Nvidia GeForce GTX 980 has 16 SMs with 128 CUDA cores\(^3\) each, making a total of 2048 CUDA cores. Nevertheless, each SM is only capable of executing simultaneously 32 threads (using only 32 CUDA cores at the same time) making it capable of running 512 threads simultaneously [15].

Each SM has very limited resources that are shared between work-groups and their work-items, thus limiting the number of work-items per work-group that can be used according to the resources needed by each work-item. The main limitation is the size of the local memory of each SM that is shared between all the work-items of the same work-group and between some work-groups (8 work-groups for the Nvidia GeForce GTX 980).

For this reason, PHACT estimates the best number of work-items per work-group to use for GPUs, by limiting the amount of local memory required to

\(^3\) A CUDA core is a processing element capable of executing one integer or floating instruction per clock for a thread.
the size of the available local memory on the GPU. When the available local memory is not enough to efficiently use at least one work-item per work-group, PHACT will only use the global memory of the device which is much larger but also much slower and 32 work-items per work-group, as each SM is only capable of running 32 threads simultaneously.

The techniques described in this section allow PHACT to use all the devices compatible with OpenCL to solve a CSP. It splits the search space in multiple search spaces that are distributed among the devices in blocks to reduce the number of communications between the host and the devices. The size of each block is calculated according to the speed of the respective device when solving the previous blocks to try to achieve a good load balancing between the devices. The size of the data transfers between the devices and the host is reduced by replacing the blocks of fully created search spaces with a small data set containing the information needed for a device to generate those search spaces.

5 Results and discussion

PHACT was evaluated on finding all the solutions for five different CSPs and on optimizing one other CSP, each one with two different sizes. Those tests were executed on one, two and three devices and on four different machines running Linux to evaluate the speedups when adding more devices to help the CPU.

PHACT performance was compared with those of PaCCS and Gecode 5.1.0 on these four machines. The four machines have the following characteristics:

M1 Machine with 32 GB of RAM and:
- Intel Core i7-4870HQ (8 compute units);
- Nvidia GeForce GTX 980M (12 compute units).

M2 Machine with 64 GB of RAM and:
- Intel Xeon E5-2690 v2 (referred to as Xeon 1, 40 compute units);
- Nvidia Tesla K20c (13 compute units).

M3 Machine with 128 GB of RAM and:
- AMD Opteron 6376 (64 compute units);
- Two AMD Tahitis (32 compute units each). These two devices are combined in an AMD Radeon HD 7990, but are managed separately by OpenCL.

M4 Machine with 64 GB of RAM and:
- Intel Xeon CPU E5-2640 v2 (referred to as Xeon 2, 32 compute units);
- Two Intel Many Integrated Core 7120P (240 compute units each).

Tables 2, 3, 4 and 5 present the elapsed times and speedups when solving multiple problems on M1, M2, M3 and M4, respectively. Five of the six problems models were retrieved from the Minizinc Benchmarks suite [12]. The Langford Numbers problem was retrieved from CSPLib [9], due to the absence of reified constraints on PHACT and PaCCS, that are used in the Minizinc Benchmarks model, which would lead to different constraints being used among the three
solvers. PaCCS does not have the “absolute value” constraint implemented, so it was not tested with the All Interval problem.

This set of problems allowed to evaluate the solvers with 8 different constraints combined with each other in different ways. All the solutions were found for the problems whose name is followed by “(count)” on the tables, and the optimal solution was searched for the problem identified with “(Optim.)”.

For simplicity, the 4 tables have the resources used on the respective machine identified as R1, R2 and R3, where R1 means using only a single thread on the CPU, R2 means using all the threads of that machine, R3 mean using all the threads on the CPU and one device (Geforce, Tesla, Tahiti or MIC), and R4 means using all the threads on the CPU and two identical devices (MICs or Tahitis). It must be noted that only PHACT is capable of using M3 and M4 resources.

Table 2 shows that using the Geforce to help I7 allowed speedups of up to 4.66. However, in two problems, using also the Geforce resulted in more time needed to solve the same problems. This result is mainly due to the small number of work-items per work-group that was effectively used on Geforce, due to the local memory limitations detailed in Section 4.

### Table 2. Elapsed times and speedups on M1.

<table>
<thead>
<tr>
<th>CSP</th>
<th>Resources</th>
<th>PHACT Elapsed (s)</th>
<th>Speedup vs. fewest resources</th>
<th>PaCCS Elapsed (s)</th>
<th>Speedup vs. fewest resources</th>
<th>GECODE Elapsed (s)</th>
<th>Speedup vs. fewest resources</th>
<th>PHACT Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Interval 14</td>
<td>R1</td>
<td>872.31</td>
<td>1488.10</td>
<td>1.36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>101.17</td>
<td>8.62</td>
<td>304.78</td>
<td>3.90</td>
<td>3.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>317.82</td>
<td>4.66</td>
<td>1321.50</td>
<td>2.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All Interval 15</td>
<td>R1</td>
<td>70.64</td>
<td>1295.94</td>
<td>2.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>1477.15</td>
<td>3303.50</td>
<td>2.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>317.82</td>
<td>4.66</td>
<td>1321.50</td>
<td>2.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Costas Array 13</td>
<td>R1</td>
<td>70.64</td>
<td>1295.94</td>
<td>2.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>317.82</td>
<td>4.66</td>
<td>1321.50</td>
<td>2.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>499.24</td>
<td>3.45</td>
<td>2142.73</td>
<td>2.95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Costas Array 15</td>
<td>R1</td>
<td>70.64</td>
<td>1295.94</td>
<td>2.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>317.82</td>
<td>4.66</td>
<td>1321.50</td>
<td>2.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>499.24</td>
<td>3.45</td>
<td>2142.73</td>
<td>2.95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Golomb Ruler 11</td>
<td>R1</td>
<td>112.58</td>
<td>495.32</td>
<td>2.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>317.82</td>
<td>4.66</td>
<td>1321.50</td>
<td>2.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>499.24</td>
<td>3.45</td>
<td>2142.73</td>
<td>2.95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Golomb Ruler 12</td>
<td>R1</td>
<td>112.58</td>
<td>495.32</td>
<td>2.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>317.82</td>
<td>4.66</td>
<td>1321.50</td>
<td>2.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>499.24</td>
<td>3.45</td>
<td>2142.73</td>
<td>2.95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Langford Numb. 13</td>
<td>R1</td>
<td>64.85</td>
<td>76.95</td>
<td>1.19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>19.54</td>
<td>3.22</td>
<td>21.04</td>
<td>1.08</td>
<td>24.84</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>171.65</td>
<td>0.93</td>
<td>197.97</td>
<td>1.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latin 5</td>
<td>R1</td>
<td>41.58</td>
<td>76.95</td>
<td>1.84</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>10.37</td>
<td>1.19</td>
<td>11.04</td>
<td>1.04</td>
<td>9.69</td>
<td>0.82</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>187.75</td>
<td>1.39</td>
<td>226.07</td>
<td>2.69</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Market Split s4-07</td>
<td>R1</td>
<td>14.48</td>
<td>3.28</td>
<td>21.04</td>
<td>1.08</td>
<td>24.84</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>289.49</td>
<td>385.62</td>
<td>1.34</td>
<td>2671.88</td>
<td>127.59</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>317.82</td>
<td>4.66</td>
<td>1321.50</td>
<td>2.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Market Split s6-01</td>
<td>R1</td>
<td>10.37</td>
<td>11.04</td>
<td>1.04</td>
<td>9.69</td>
<td>0.82</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>212.86</td>
<td>891.21</td>
<td>4.22</td>
<td>153.94</td>
<td>0.95</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>187.75</td>
<td>1.39</td>
<td>226.07</td>
<td>2.69</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The slowdown noted when optimizing the Golomb Ruler with 12 marks is also due to the impossibility of different devices communicating with each other while their kernels are running, as stated in Section 4. This is problematic when
optimizing, as once a device finds a better solution, it cannot tell the other devices to find only solutions better than the one it just found. Instead it will finish exploring its block of sub-search spaces and only after that it will inform the host about the new solution, and only after this point, when another device finishes its block, it will be informed about the new solution that must be optimized. Due to this limitation, the devices spend some time looking for solutions that may already be worse than the ones found by other devices. This problem was also noted on the other three machines.

As for the Langford Numbers problem with 14 numbers, the worse result when adding the Geforce was due to the largely unbalanced sub-search spaces that are generated leading to most of sub-search spaces being easily detected as inconsistent, and only a few containing most of the solutions. This is problematic, because as each thread explores each sub-search space sequentially, in the end only a few threads will be working on the harder sub-search spaces while the others are idle. This problem was also noted on the other three machines.

In most of the problems, PHACT was faster than PaCCS, achieving speedups of up to 5.37. When working with smaller problems like the “Latin 5 (Count)”, PHACT is slower than PaCCS due to the time needed to initialize the devices and compiling the OpenCL kernel.

When comparing with Gecode, PHACT achieved good speedups on all the problems, except on Market Split, which is a simple problem with only one constraint type which may have a faster propagator on Gecode. On the contrary, with the Latin problem, Gecode has some issues which result on PHACT achieving a speedup of up to 127.85 when using only the CPU.

Table 3 presents the results on solving the same problems on M2. Using the Tesla GPU to help the Xeon 1 resulted in most of the cases in a slowdown. This is due to the fact that Tesla was the slowest GPU used on the tests, being no match for Xeon 1. In fact, the work done by Tesla didn’t compensate the time spent by Xeon 1 (host) to control Tesla (device).

On this machine, PHACT and PaCCS exhibited similar performances, resulting in a geometric mean speedup of 1.01 favoring PHACT. Comparing with Gecode, PHACT was faster on all the problems with all the resources combinations.

The results for the M3 machine are presented in Table 4. This machine possesses the CPU used on the tests that has the greater number of cores (64), and it is paired up with two Tahiti GPUs, that are faster than Tesla, but slower than Geforce. So it is very hard for the Tahitis to display some performance gains when compared with a 64 cores CPU. However, with the All Interval 15 problem, they were capable of speeding up the solving process by 1.48 times. The issues with Golomb Ruler and Langford Number discussed before in this section, were also noted on this machine.

When comparing with PaCCS, PHACT achieved speedups that ranged from 0.07 on a very small problem to 4.67. PHACT was faster than Gecode in all the tests, except when optimizing Golomb Ruler 12 with the Opteron and one Tahiti. The issues of Gecode with the Latin 6 problem were more visible on
### Table 3. Elapsed times and speedups on M2.

<table>
<thead>
<tr>
<th>CSP</th>
<th>Resources</th>
<th>PHACT Elapsed (s)</th>
<th>PHACT Speedup vs. fewest resources</th>
<th>PaCCS Elapsed (s)</th>
<th>PaCCS Speedup vs. fewest resources</th>
<th>GECODE Elapsed (s)</th>
<th>GECODE Speedup vs. fewest resources</th>
<th>Speedup</th>
<th>PHACT speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Interval 14 (Count)</td>
<td>R1</td>
<td>928.77</td>
<td>1.356.75</td>
<td>1.47</td>
<td>R2</td>
<td>288.99</td>
<td>3.190.19</td>
<td>9.74</td>
<td>1.47</td>
</tr>
<tr>
<td>All Interval 15 (Count)</td>
<td>R2</td>
<td>429.59</td>
<td>2834.76</td>
<td>6.69</td>
<td>R3</td>
<td>364.69</td>
<td>1.29</td>
<td>1.98</td>
<td></td>
</tr>
<tr>
<td>Costas Array 13 (Count)</td>
<td>R1</td>
<td>74.91</td>
<td>186.80</td>
<td>2.49</td>
<td>186.84</td>
<td>2.49</td>
<td>186.84</td>
<td>2.49</td>
<td></td>
</tr>
<tr>
<td>Costas Array 15 (Count)</td>
<td>R2</td>
<td>4.79</td>
<td>7.81</td>
<td>21.74</td>
<td>1.63</td>
<td>10.35</td>
<td>18.21</td>
<td>21.16</td>
<td></td>
</tr>
<tr>
<td>Gecode Ruler 11 (Optim.)</td>
<td>R2</td>
<td>10.96</td>
<td>17.71</td>
<td>21.74</td>
<td>2.06</td>
<td>29.85</td>
<td>14.08</td>
<td>24.44</td>
<td></td>
</tr>
<tr>
<td>Gecode Ruler 12 (Optim.)</td>
<td>R2</td>
<td>11.22</td>
<td>253.73</td>
<td>22.22</td>
<td>293.17</td>
<td>2.37</td>
<td>3.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Langford Numbers 13 (Count)</td>
<td>R2</td>
<td>4.51</td>
<td>36.41</td>
<td>0.41</td>
<td>106.58</td>
<td>2.21</td>
<td>4.48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latin 5 (Count)</td>
<td>R1</td>
<td>1.68</td>
<td>0.20</td>
<td>0.17</td>
<td>3.94</td>
<td>2.35</td>
<td>1.46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latin 6 (Count)</td>
<td>R2</td>
<td>1.96</td>
<td>0.36</td>
<td>0.07</td>
<td>9.67</td>
<td>0.62</td>
<td>0.62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Market Split s4-07 (Count)</td>
<td>R2</td>
<td>2.41</td>
<td>0.70</td>
<td>0.07</td>
<td>15.61</td>
<td>2.16</td>
<td>2.44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Market Split s5-01 (Count)</td>
<td>R2</td>
<td>4.89</td>
<td>146.65</td>
<td>3.21</td>
<td>1161.31</td>
<td>25.05</td>
<td>2.44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4. Elapsed times and speedups on M3.

<table>
<thead>
<tr>
<th>CSP</th>
<th>Resources</th>
<th>PHACT Elapsed (s)</th>
<th>PHACT Speedup vs. fewest resources</th>
<th>PaCCS Elapsed (s)</th>
<th>PaCCS Speedup vs. fewest resources</th>
<th>GECODE Elapsed (s)</th>
<th>GECODE Speedup vs. fewest resources</th>
<th>Speedup</th>
<th>PHACT speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Interval 14 (Count)</td>
<td>R1</td>
<td>1806.78</td>
<td>2363.80</td>
<td>1.31</td>
<td>R2</td>
<td>68.28</td>
<td>838.62</td>
<td>12.82</td>
<td>31.69</td>
</tr>
<tr>
<td>All Interval 15 (Count)</td>
<td>R2</td>
<td>135.33</td>
<td>856.00</td>
<td>24.20</td>
<td>R3</td>
<td>175.26</td>
<td>2.06</td>
<td>114.31</td>
<td></td>
</tr>
<tr>
<td>Costas Array 13 (Count)</td>
<td>R1</td>
<td>336.82</td>
<td>345.57</td>
<td>2.30</td>
<td>334.95</td>
<td>2.30</td>
<td>334.95</td>
<td>2.30</td>
<td></td>
</tr>
<tr>
<td>Costas Array 15 (Count)</td>
<td>R2</td>
<td>3.99</td>
<td>7.91</td>
<td>20.69</td>
<td>4.60</td>
<td>23.39</td>
<td>10.61</td>
<td>6.17</td>
<td></td>
</tr>
<tr>
<td>Gecode Ruler 11 (Optim.)</td>
<td>R1</td>
<td>378.81</td>
<td>1225.71</td>
<td>3.23</td>
<td>640.38</td>
<td>1.70</td>
<td>640.38</td>
<td>1.70</td>
<td></td>
</tr>
<tr>
<td>Gecode Ruler 12 (Optim.)</td>
<td>R2</td>
<td>32.37</td>
<td>20.72</td>
<td>31.24</td>
<td>2.44</td>
<td>31.24</td>
<td>2.44</td>
<td>31.24</td>
<td>2.44</td>
</tr>
<tr>
<td>Langford Numbers 13 (Count)</td>
<td>R3</td>
<td>138.88</td>
<td>173.95</td>
<td>1.26</td>
<td>190.70</td>
<td>1.38</td>
<td>190.70</td>
<td>1.38</td>
<td></td>
</tr>
<tr>
<td>Latin 5 (Count)</td>
<td>R2</td>
<td>6.38</td>
<td>20.89</td>
<td>3.24</td>
<td>47.35</td>
<td>4.88</td>
<td>47.35</td>
<td>4.88</td>
<td></td>
</tr>
<tr>
<td>Latin 6 (Count)</td>
<td>R1</td>
<td>4.02</td>
<td>40.91</td>
<td>1.26</td>
<td>387.30</td>
<td>9.56</td>
<td>387.30</td>
<td>9.56</td>
<td></td>
</tr>
<tr>
<td>Market Split s4-07 (Count)</td>
<td>R1</td>
<td>103.43</td>
<td>0.30</td>
<td>0.40</td>
<td>2.38</td>
<td>0.96</td>
<td>0.96</td>
<td>1.26</td>
<td></td>
</tr>
<tr>
<td>Market Split s5-01 (Count)</td>
<td>R4</td>
<td>56.01</td>
<td>0.91</td>
<td>0.12</td>
<td>1.21</td>
<td>0.91</td>
<td>0.91</td>
<td>1.21</td>
<td></td>
</tr>
</tbody>
</table>

Pedro Roque, Vasco Pedro, Salvador Abreu
this machine, where it needed more than 38 hours to solve it, although PHACT and PaCCS solved it in less than 2 minutes. Gecode was faster when using only one thread to solve the Latin 5 problem than when using all the threads of the CPU on each machine, which suggests that the method used for load balancing between threads is very inefficient for this problem.

Table 5 presents the results on the M4 machine. This machine possesses two MICs whose architecture is more similar to the CPUs than to GPUs, so, they are more prepared for solving sequential problems than GPUs. That difference was noted with the Langford Numbers problem, where they were capable of achieving a speedup of 1.51 despite the unbalanced sub-search spaces. When counting all the solutions for the Costas Array 15, the two MICs allowed a top speedup of 1.90.

<table>
<thead>
<tr>
<th>CSP</th>
<th>Resources</th>
<th>PHACT</th>
<th>PaCCS</th>
<th>GECODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed (s)</td>
<td>Speedup vs. fewest resources</td>
<td>Elapsed (s)</td>
<td>Speedup vs. fewest resources</td>
<td>PHACT speedup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All Interval 14 (Count)</td>
<td>R1</td>
<td>1,397.78</td>
<td></td>
<td>1,930.32</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>835.90</td>
<td>36.86</td>
<td>310.89</td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>310.84</td>
<td>2.49</td>
<td>310.84</td>
</tr>
<tr>
<td></td>
<td>R4</td>
<td>245.44</td>
<td>2.58</td>
<td>245.44</td>
</tr>
<tr>
<td>Latin 5 (Count)</td>
<td>R1</td>
<td>11.36</td>
<td>262.02</td>
<td>2.27</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>7.34</td>
<td>15.92</td>
<td>19.83</td>
</tr>
<tr>
<td>Costas Array 15 (Count)</td>
<td>R1</td>
<td>2,297.43</td>
<td>1.34</td>
<td>2,297.43</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>1.21</td>
<td>11.42</td>
<td>16.09</td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>1.21</td>
<td>11.42</td>
<td>16.09</td>
</tr>
<tr>
<td></td>
<td>R4</td>
<td>279.53</td>
<td>0.70</td>
<td>1.57</td>
</tr>
<tr>
<td>Market Split s4-07 (Count)</td>
<td>R1</td>
<td>113.68</td>
<td>120.83</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>114.42</td>
<td>11.34</td>
<td>16.09</td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>114.42</td>
<td>11.34</td>
<td>16.09</td>
</tr>
<tr>
<td></td>
<td>R4</td>
<td>114.42</td>
<td>11.34</td>
<td>16.09</td>
</tr>
<tr>
<td>Market Split s5-01 (Count)</td>
<td>R1</td>
<td>80.58</td>
<td>213.38</td>
<td>2.64</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>68.03</td>
<td>1.19</td>
<td>3.14</td>
</tr>
<tr>
<td></td>
<td>R3</td>
<td>68.03</td>
<td>1.19</td>
<td>3.14</td>
</tr>
<tr>
<td></td>
<td>R4</td>
<td>68.03</td>
<td>1.19</td>
<td>3.14</td>
</tr>
</tbody>
</table>

When compared with PaCCS and Gecode the results are very similar to the ones achieved on the other machines, being faster than Gecode in all but one problem and faster than PaCCS in about 1/3 of the tests.

Figure 1 presents the geometric mean of the speedups achieved by PHACT against PaCCS and Gecode, showing that PHACT was faster than Gecode on all the machines with all the resources combinations.
PHACT was also faster than PaCCS when using all the resources combinations on the four machines, except on M2 and M4 when using only the CPU. This may be due to the fact that PaCCS uses work stealing to distribute the work among threads, which allows them to share work until all the work is done. PHACT is unable to share the same search-space between threads, which makes some threads finish their work before the others and consequently worsen performance.

We can also observe that the difference in performance between PHACT and Gecode is greater on the machines that have a CPU with more cores, which shows that the load balancing techniques implemented in PHACT are more efficient for the problems that were presented here. When compared with PaCCS, that relation is no longer noticed and the results are very closed between the two solvers when using only the CPUs.

Using all the available resources on the four machines allowed PHACT to increase its performance when compared to PaCCS and Gecode, which shows that its greater versatility can compensate.

6 Conclusion and future work

To our knowledge, PHACT is the only constraint solver capable of using simultaneously CPUs, GPUs, MICs and any other device compatible with OpenCL to solve CSPs in a faster manner.

Although GPUs are not particularly efficient for this type of problems, they can speed up the solving process and in some cases, be even faster than the CPU of the same machine.

PHACT has been tested with 6 different CSPs on 4 different machines with 2 and 3 devices each, namely Intel CPUs and MICs, Nvidia GPUs, and AMD
CPUs and GPUs, allowing it to achieve speedups up to 4.66 when compared with using only the CPU of the machine.

On the four machines used for testing, PHACT achieved a geometric mean speedup that ranged from 0.82 to 2.83 when compared with PaCCS, and 1.71 to 28.44 when compared with Gecode.

The use of all the devices compatible with OpenCL to solve a CSP allowed PHACT to improve its performance against PaCCS and Gecode when compared with using only the CPUs.

Campeotto et al. [4] use the GPUs to propagate the constraints that relate many variables. Although this technique seems to have significant synchronization requirements between host and device, we intend to test this approach in the future.

PHACT is yet being improved to try to overcome the lack of synchronization between devices when optimizing. The solution may pass by a more frequent communication between host and devices, taking into account the number of solutions already found and making the communication more frequent for problems with more solutions.

As for the unbalanced sub-search spaces that lead to only a few threads working in parallel while the others have already finished their work, we are analysing a work-sharing strategy [18] that may be executed when all the sub-search spaces generated for the block has ended but some threads are still working.

A MiniZinc/FlatZinc [14] reader is also being implemented to allow the direct input of problems already modeled in this language.

Acknowledgments

This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) under grant UID/CEC/4668/2016 (LISP). Some of the experimentation was carried out on the khromeleque cluster of the University of Évora, which was partly funded by grants ALENT-07-0262-FEDER-001872 and ALENT-07-0262-FEDER-001876.

References


Run-time Analysis of
Temporal Constrained Objects

Jinesh M Kannimoola\(^1\), Bharat Jayaraman\(^2\), and Krishnashree Achuthan\(^1\)

\(^1\) Amrita Center for Cybersecurity Systems and Networks, Amrita School of Engineering, Amritapur, Amrita Vishwa Vidyapeetham, Amrita University, India
\(^2\) Department of Computer Science and Engineering, State University of New York at Buffalo, USA

\textit{Abstract.} The programming paradigm of constrained objects is a declarative variant of the object-oriented paradigm wherein objects define the structure of a system and declarative constraints (rather than imperative methods) define its behavior. Constrained objects have many uses in the engineering domain and computation in this paradigm is essentially constraint solving. This paper is concerned with an extension of constrained objects called temporal constrained objects, which are especially appropriate for modeling dynamical systems. The main extensions are series variables and metric temporal operators to declaratively specify time-varying behavior. The language TCOB exemplifies this paradigm and the execution of TCOB programs consists of constraint solving within a time-based simulation framework. One of the challenges in TCOB is identifying errors owing both to the complexity of programs and the underlying constraint solving methods. We address this problem by extracting a run-time trace of the execution of a TCOB program and providing an analysis of the cause of error. The run-time trace also serves as a basis, in many cases, for constructing a finite-state machine which in turn can be used for ‘model-checking’ properties of the system. The paper also presents abstraction techniques for dealing with simulations that result in large state spaces.

\textit{Keywords:} Temporal Constraints Objects, Time-based Simulation, Run-time Verification, Finite State Models, Error Analysis, Predicate Abstraction, Visualization

1 Introduction

Constrained objects are a natural modeling approach for complex structures with two essential characteristics: (i) They are compositional in nature, i.e., a complex structure is built (recursively) of smaller structures. (ii) The behavior of an individual component by itself and its relation to other components are regulated by laws, or rules. The language COB exemplifies this concept, and it
has been shown to be useful especially in modeling complex engineering structures [7]. COB makes use of Java-like classes, inheritance, and aggregation for modeling structure and makes use of declarative constraints (rather than imperative methods) for modeling behavior. The emergent behavior of a collection of constrained objects is determined by a process of constraint solving over the attributes of objects.

Temporal constrained objects [8] are an extension of constrained objects that are particularly suited for modeling the time-dependent behavior of complex dynamic systems. The new feature here is the series variable, which records the sequence of changes to some entity of interest in a dynamic system. Time is considered as a metric quantity; a built-in variable Time represents the current time and it is automatically incremented by one unit to record the passage of time. For example, while the current and voltage across a resistor in a DC circuit can be modeled using ordinary variables, in an AC circuit the current and voltage change with time and hence are better modeled with series variables. It is common, in such examples, for constraints to be placed over consecutive values in the time-sequence. The language TCOB extends COB with series variables as well as metric temporal operators, which are a metric variant of the classic temporal operators of LTL [3]. Together they are effective in specifying the overall dynamic behavior of a variety of complex structures. The execution model for TCOB involves a time-based simulation along with constraint solving at each time-step.

Run-time analysis refers to methods and tools for monitoring the run-time behavior of a program or system with the goal of debugging and verifying its behavior. An important aspect of run-time analysis is run-time verification, which attempts to bridge the gap between formal verification and software testing [9]. Most of these techniques are based upon a finite execution trace of a program [12]. The execution trace records the major events that occurred during execution, such as variable/field read’s and write’s, method call/return, and object creation. In this paper we investigate the usability of a similar approach in the paradigm of temporal constrained objects, an important difference being that state updating is not permitted in our paradigm. The time-based simulation used in temporal constrained objects naturally lends itself to analysis based upon a linear execution trace. Detecting errors when constraint solving is interwoven with a time-based simulation is especially challenging, but we show how run-time analysis together with run-time visualization greatly help in addressing this challenge. We construct a temporal constraint dependency graph at run-time in order to clarify temporal dependencies and help identification of errors.

We also extract a finite state machine from the execution trace and formulate properties of interest as verification conditions in a propositional temporal logic [3]. A state consists of the values of a set of variables chosen by the user. The set of states is the set of distinct combinations of values taken by these variables during the course of program execution. Since state updating is not possible in temporal constrained objects, state changes are possible only because series
variables may assume different values as time progresses. Of course, it is possible that the series variables assume the same values at two different points in time, i.e., it is possible that states repeat. Sometimes there could be a large number of states, and there is a need to construct a reduced run-time model that clarifies the emergent behavior at a high level. We propose an approach which we call **predicate abstraction** in order to reduce the number of states without losing an abstract view of the system. In our approach, we can directly encode the predicate abstraction rules as constraints in the system. Thus, our run-time analysis is a combination of visualization, error-detection and verification.

The remainder of this paper is organized as follows. Section 2 discusses the related literature in this field; Section 3 introduces the concept of temporal constrained objects with the aid of examples; Section 4 presents the run-time analysis of temporal constrained objects; and, finally, Section 5 presents conclusions and areas of further work.

## 2 Related Work

Run-time verification is based on extracting a trace from a running system and using it to detect observed behaviors satisfying or violating certain properties [9]. Binary code instrumentation [4] is one of the most commonly used mechanisms for trace extraction. Ducasse et al [5] discuss dynamic program analysis and debugging in different logic programming environments. Different approaches are used for extracting an execution trace from a logic program, including source code instrumentation, meta-interpreter instrumentation and compiled code instrumentation.

As the textual representation of the trace is usually not very informative and often difficult to interpret and understand, tools such as JIVE [12] and Java Path Finder [4] support various diagrams built from the execution trace for the easy debugging of Java programs. In the logic programming context, in addition to domain specific visualization, most logic programming environments support step-by-step execution of programs with limited graphical debugging capability. A good example is SWI-Prolog [11] which provides a step-by-step debugger with views of the call-stack and variable bindings.

The concept of variable binding in constraint languages is more complex than in imperative languages. Carro et al [2] presents a visual representation of finite domain CLP programs. This approach mainly focuses on the evolution of variables during the labeling phase of constraint evaluation. The reference [2] also introduces the concept of abstract representation of a variable that has a large number of possible values.

Run-time verification integrates tools and techniques proposed in the formal methods field for the analysis of execution trace [4]. Run-time verification is similar to model checking approach except that the model is built from finite set of traces. RV-Match, RV-Predict and RV-Monitor [4] are some of the examples which use formal analysis methods such as a symbolic execution engine, a semantic debugger, a model checker, and a full-fledged deductive program verifier for...
the debugging and verification of sequential and object oriented programs. Reference [1] introduces a three-valued semantics to include the meaning of partial observation in run-time verification, where an inconclusive decision represents the fact that the trace is not long enough to determine the truth value for temporal specification.

The proposed approach in this paper has much in common with JIVE [12] which supports run-time verification of state diagrams extracted from one or more execution traces. This includes checking consistency of run-time with design-time state diagrams, as well as checking properties stated in CTL. In our approach, the execution trace is implicitly present in the values of series variables at different points in time. In a way, series variables simplify the task of extracting an execution trace.

3 Temporal Constrained Objects

Temporal constrained objects extend the basic paradigm of constrained objects to support time-varying properties of dynamic systems. The TCOB execution follows a discrete time simulation based on the value of built-in variable $Time$. The user can attain any granularity of time by multiplying a suitable scaling factor with $Time$, e.g., $MyTime = 0.01 \times Time$. The default initial value for $Time$ is equal to 1 unless the different value is specified by the user. A TCOB program defines a collection of classes, each of which contains a set of attributes, constraints, predicates, and constructors [8]. Each of temporal constrained object is an instance of some class whose outline is as follows.

```
class_definition ::= [abstract] class class_id [extends class_id] { body }
body ::= { attributes attributes ]
          | constraints constraints ]
          | predicates predicates ]
          | constructors constructors ]
```

As in Java, single inheritance is defined by the extends keyword. An abstract class is a class without a constructor and cannot be instantiated. An attribute is a typed identifier, which support both primitive and user-defined types. The keyword series is used to define series variable. The series variable takes on an unbounded sequence of values over time, and temporal constraints are defined in terms of past and future values of the series variable. For every series variable $v$, the TCOB expression $v$ and $v'$ refers to the immediate previous and next values of $v$ respectively. These operators can be juxtaposed to refer to successive values of $v$ in the past or future. The past values of $v$ at any point in time are referred to by $v$, $v'$, $v''$, and the future values of $v$ at any point in time are referred to by $v'$, $v''$, $v'''$, ... TCOB also allows a series of variables to be initialized by explicitly assigning values at specific time points. Appendix A shows the complete grammar of TCOB syntax.
Constraints are relations over the attributes of classes. TCOB supports simple, conditional, quantified and creational constraints over typed attributes. It also provides two metric temporal operators, F and G, analogous to ‘always’ (□) and ‘eventually’ (◇) operators in (non-metric) temporal logic. In the following, evaluation of F and G operators are carried out with respect to the current value of Time. The constraints specified with metric temporal operator is called metric temporal constraints, which appears in the body part of conditional constraint. The basic meaning of metric temporal constraints are

1. \( F p \) states that constraint \( p \) must hold at some time point in the future;
2. \( F<\tau> p \) states that constraint \( p \) must hold exactly after \( \tau \) units of time; and
3. \( F<\tau1,\tau2> p \) states the constraint \( p \) must hold sometime starting after \( \tau1 \) units of time but before \( \tau2 \) units of time.
4. \( G p \) states that constraint \( p \) must hold at all time points in the future;
5. \( G<\tau> p \) states that constraint \( p \) must hold at all time points after \( \tau \) units of time; and
6. \( G<\tau1,\tau2> p \) states the constraint \( p \) must hold at all time points starting after \( \tau1 \) units of time but before \( \tau2 \) units of time.

The following section explains the PID controller and traffic lights example from [8].

**PID Controller Example:** A Proportional Integral Derivative (PID) controller is one of the most commonly used control-loop feedback mechanism in industrial automation. The PID controller consists of three components: controller, plant, and sensor [10]. The output of the controller is given as the control input to the plant. The sensor collects the output from the plant and calculates the error based on the desired output. The estimated error is fed back as the controller input. The controller is defined by the equation.

\[
u(t) = K_p e(t) + K_i \int_0^t e(t) dt + K_d \frac{de(t)}{dt}\]

Here \( u(t) \) represents the output from the controller; \( e(t) \) is the error feedback to the controller; and the \( K_p, K_i \), and \( K_d \) are the non-negative coefficients. The first term denotes the present value of the error; the second term indicates the past values of error; and the last term accounts for future values of error. The corresponding TCOB formulation is given below:

```java
class controller{
  attributes
    int Kp, Ki, Kd;
    series real Error,ESum,Out;
  constraints
    Kp > 0; Kd > 0; Ki > 0;
    ESum = Error + 'ESum;
    Out = (Kp * Error) + (Ki * ESum) + Kd * (Error - 'Error);
```
constructors controller(KP, KI, KD) {
  Kp = KP; Ki = KI; Kd = KD;
  Error<1> = 0; ESum<1> = 0; Error<2> = 0; ESum<2> = 0;
}
}

The series variables Error and Out define the input and output of the controller respectively. The constraint section models the PID controller equation. For the plant model, we consider a simple mass spring damper problem. The modeling equation is

\[
M \frac{d^2x}{dt^2} + b \frac{dx}{dt} + kx = F
\]

where \( x \) is the displacement of body, \( M \) represents the mass of the body, and \( b \) and \( k \) are the damping constant and spring constant, and \( F \) is the force applied on the body to position it. In each iteration, the controller calculates a new force and the plant takes it as the input. In the equilibrium condition of this experiment, the displacement value remains at one.

class plant{
  attributes
    series real Fo, X, V;
    int M,B,K;
  constraints
    V = X - 'X;
    Fo = M * (V - 'V) + B * V + K * X;
  constructor plant(M1,B1,K1,X1,V1)
  { M = M1; B = B1; K = K1; X<1> = X1; V<1> = V1; }
}

The sensor takes the output from the plant, and the system works out the error and feedback to the controller. Appendix B gives the definition of sensor and system class.

The error is the difference between the expected value (here the displacement=1) and sensed value. In our experiments, we modeled the system by assigning the value one to the coefficients \( K_p, K_i \) and \( K_d \). We can fine tune this value based on the system behavior.

Traffic Light Example: Consider two traffic lights which control respectively the traffic in east-west and north-south intersections by sensing the traffic on these roads and controlling the duration of the green and red lights depending upon the number of vehicles on each road. We refer to these as “intelligent” traffic lights and they are modeled by class int_light below. The traffic_sensor class generates random values (between 0 and 5) to represent the number of new vehicles that have arrived at every time point. The int_light class sums up these values to determine the total number of vehicles waiting while the light is not green. When the light changes to green, the number of waiting vehicles determines the duration of the green light (5 time steps for each vehicle); if this
number is zero, the green light is skipped. The remaining constraints specify the safety and synchronization properties between traffic lights.

class int_light{
  attributes
    enum Color = [red, green, yellow];
    series Color C;
    traffic_sensor Ts; int_light Tl;
    series int NumV;
  constraints
    C = green --> NumV = 0;
    not ( C = green ) --> NumV = ‘NumV + Ts.V;
    ‘C = red & C = yellow & NumV > 0 -->
        G<1,NumV * 5> C = green & F<NumV * 5> C = yellow;
    ‘C = green & C = yellow & Tl.NumV = 0 -->
        G<1,NumV * 5> C = green & F<NumV * 5> C = yellow;
    Tl.C = yellow --> C = yellow; Tl.C = green --> C = red;
  constructors int_light(C1,C2,TL){
    Tl = TL; C<1> = C1; C<2> = C2; NumV<2> = Ts.V<2>;
    Ts = new traffic_sensor();
  }
}

class system {
  attributes
    int_light NS, EW;
  constructor system(){
    NS = new int_light(red,yellow,EW);
    EW = new int_light(green,yellow,NS);
  }
}

The system class initializes the two traffic lights with initial colors. The constructor call NS = new int_light(red,yellow,EW) creates the North-South traffic light with initial colors and traffic light in the opposite direction. The NS object makes use of EW object attributes for color synchronization. Note that EW is unbound when NS is initialized, but the implementation of conditional constraints handles this situation by keeping the unbounded object constraints in the constraint store and automatically invoking it when EW becomes bound.

4 Run-time Analysis

Run-time analysis of temporal constrained objects consists of three main components: visualization, error detection, and verification. Fig. 1 gives an overview of these three aspects, which are all driven by an execution trace. In order to obtain this execution trace, the body of every class definition may include an
optional ‘monitor’ clause which specifies which class attributes are to be monitored during execution. At run-time, the underlying system creates an execution trace file containing, for every time-point of execution, the attribute name, object reference and value for every designated attribute in every monitor clause in the program. The following is a sample trace file for the PID controller example, assuming we have added a monitor clause for attribute $X$ in class \texttt{plant}.

\begin{verbatim}
Time = 2, Obj = P, Var = X, Val = NaV
Time = 2, Obj = P, Var = X, Val = 0.3870967741935484
Time = 3, Obj = P, Var = X, Val = NaV
Time = 3, Obj = P, Var = X, Val = 0.1768990634755463
\end{verbatim}

The monitor clause effectively causes a listener to be placed on each attribute to be monitored. The monitor outputs a value ‘NaV’ (Not a Value) if the attribute is undefined.

4.1 Run-time Visualization

Visualization conveys information in a more readable and efficient way and, in our approach, it serves as the foundation for debugging and verification. Our analysis framework provides a web-based GUI (Fig. 2a), where the user can upload an execution trace file generated by the monitored program and then select one or more attributes for visualization. Currently, the framework supports three types of diagrams, as described below.

\textit{Timed State Diagram}. A state diagram is a more precise way to portray the evolution of a system over time. The timed state diagram describes the values of variables at each discrete time-points. Once a non-series variable is bound,
its value remains fixed at every time-point whereas series variables may assume different values at different time-points. For example, Fig. 2b shows the timed state diagram of a traffic light example.

**Abstract State Diagram.** The abstract state diagram shows the abstract view of the system by extracting the distinct states of the execution. The timed state diagram shows the linear progression of the system with respect to time. This diagram is not an appropriate choice when we wish to visualize the repetitive behavior of a system. The following procedure can build the abstract state diagram from a trace by considering the state vector in each trace entry.

```
Trace T;
Abstract_State_Graph AS = {}
foreach E<time,state> in T
if (E.state not in AS)
    AS = AS U {E.state};
```
Fig. 2c present the abstract view of traffic light example, whereas the equivalent timed state diagram would contain as many states as the number of simulation steps.

**Timed Line Diagram.** The timed line diagram plots the values of the chosen variables over time. Unlike other diagrams, it supports only numerical attributes but is often useful in identifying incorrectness by direct inspection of the form the output diagram. For example, the Fig. 2d shows the timed line diagram for displacement($X$) in the PID controller example. It captures the correctness of PID controller implementation, since the displacement should eventually reach a stable state.

### 4.2 Run-time Error Detection

Several factors contribute to an erroneous output of a TCOB program and it is hard to pinpoint the cause of failure in a complex large-scale simulation. Here we illustrate the various methods to identify the cause of errors in TCOB simulation using run-time analysis.

**Case 1: Constraint Failure.** The execution of a TCOB program involves a discrete-time simulation with constraint satisfaction performed at each time-point, where the constraints may involve non-series as well as series variables. Each time-step can be viewed as a *computation frame* that involves values of series variables from next and previous time-steps. The size of the computation frame depends on the next and previous reference of series variables in the constraints; the default size is 1 when there are no such references.

We illustrate below with a simple example of a moving object whose position at each step depends on the previous position and a constant value. The programmer erroneously enters a negative value ($-1$) for the constant. As a result, the unary constraint $P > 0$ fails at Time = 3 since the value of the series variable $P$ is 2 at Time = 1 and it decreases by 1 at each time-step.

```java
class example {
  attributes
    series int P; int C;
  constraints
    P > 0;
    P = 'P + C;
  constructors example()
    { C = -1; P<1> = 2; }
  monitor P,C;
}
```

In general, the interweaving of constraint solving within a time-based simulation makes it is difficult to detect this type of programming error in a larger system. In order to address this problem, we propose the use of a *temporal constraint dependency graph* using the computation frame of TCOB execution. In
this example, the size of computation frame is 2. Each computation frame main-
tains two sets: bound variables (BV) and unbound variables (UV). The bound
variables set maintains the set of ground variables along with their values. Both
sets are computed before constraint solving is initiated at each time step. Fig.
3a gives the computation frame and temporal constraint dependency graph at
\textbf{Time} = 2. The edge between nodes indicates a (binary) constraint relation and
the self-loop indicate the unary constraints. The vertical dotted lines delineate
the computation frame.

The computation frame moves by one time unit at each execution step. For
example, at \textbf{Time} = 3 the computation frame is shown in Fig. 3b. The dotted
edge indicates the relation in previous computation frame. During the constraint
solving at \textbf{Time} = 3, the variable \texttt{P<3>} is assigned to 0 and causes a violation
of the unary constraint \texttt{P>0}. The user can quickly backtrack and find which
assignment or constraint leads to this inconsistency. The red edge shows the
constraint violation in the system.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3}
\caption{Temporal Constraint Dependency Graph}
\end{figure}

\textit{Case 2: Incorrect and Undefined Answers.} Incorrectness arises when constraint
satisfaction results in a successful outcome but the computed answer for one or
more variables is incorrect. The probable cause could be incorrect constraints,
erroneous initialization or possibly incorrect assembly of objects. The user must
have enough domain knowledge to distinguish which of these cases is the real
cause. When the programmer has some partial knowledge about the output, s/he
can encode it as a constraint in the program.

One of the primary advantages of programming with constraints is their
ability to compute with partial (or incomplete) information. Sometimes partial
information is not adequate to compute a specific value for a variable. This
usually happens in TCOB due to either missing variable initialization or missing
constraints. Consider the following constructor from the \texttt{controller} class in the
PID controller example.
controller(KP,KI,KD){
    Kp = KP; Ki = KI; Kd = KD;
    Error<2> = 0; Error<1> = 0;
}

Suppose we monitor the series variables ESum, Error and Out. Fig. 4 presents the state diagram of execution, which is built from the execution trace of the PID controller program. The figure clearly shows that the constraints failed to determine exact values for ESum and Out using the available information at time step 2. The current value ESum depends on Error and previous value of ESum. The initialization ESum<1> = 0 in the constructor can correct this problem.

4.3 Run-time Verification

We now illustrate the concept of run-time verification for temporal constrained objects. In a model-based approach [3], we check the consistency of a model $M$ against a specification $\phi$. In run-time verification, execution traces are used to build a run-time model and properties are verified for this model. Both timed as well as the abstract state diagram can serve as the basis of a model. However, the abstract state diagram is a more compact view of the timed state diagram; it is essentially a (run-time) Kripke structure [3] in the terminology the model checking. The safety and liveness verification conditions are specified as LTL formula; if an LTL condition is true in the abstract state diagram it is also true in the timed state diagram.

For example, the abstract state diagram for the traffic light example of section 3 is given in Fig. 2c. In this example, safety means that two lights are not green at the same time, and liveness means that the lights always changing without being stuck at any state. The corresponding LTL formulation is as follows, where $NS$ and $EW$ refer to north-south and east-west respectively. From Fig. 2c, it is evident that all these conditions are satisfied by our model.

- $\Box \neg ((NS.C = \text{green}) \land (EW.C = \text{green}))$
- $\Box (NS.C = \text{green} \implies \Diamond NS.C = \text{red})$
- $\Box (NS.C = \text{red} \implies \Diamond NS.C = \text{green})$
- $\Box (EW.C = \text{green} \implies \Diamond EW.C = \text{red})$
- $\Box (EW.C = \text{red} \implies \Diamond EW.C = \text{green})$
There is an important difference between a run-time state diagram derived from a finite execution trace and the design-time state diagrams used in model-checking. Whereas cycles in the design-time state diagrams represent nonterminating execution paths, cycles in run-time state diagrams represent finite execution paths. For example, in Fig. 2c, the self-loops are executed only a finite number of time-steps.

**Predicate Abstraction** Predicate abstraction is a form of abstract interpretation wherein we can reduce the size of the model constructed by abstracting details [6]. For example, if in some analysis we care only whether an integer variable is negative, zero, or positive, we can effectively abstract the infinite set of integers by a set with just three values. TCOB simulations can result in a large number of states causing a state explosion problem during run-time verification. Predicate abstraction is very useful in reducing the run-time state diagrams. This abstraction can directly specified as declarative constraints in a TCOB class definition.

For example, consider the series variable \( X \) from the `plant` class in PID controller example. This models the displacement, a real number, which has minute differences from one time-step to another. We need to verify this variable \( X \) eventually reaches a stable state, where value is approximately equal to required displacement value, namely, 1.

In such scenarios, we can use predicate abstraction by grouping the displacement value into a few different ranges using constraints, as shown below.

\[
\begin{align*}
X > 0.4 & \land X < 0.8 &\rightarrow PV = 0.5; \\
X < 0.4 &\rightarrow PV = 0.0; \\
X > 0.8 & \land X < 1.2 &\rightarrow PV = 1; \\
X > 1.2 &\rightarrow PV = 1.5
\end{align*}
\]

Here \( PV \) is a series variable. For a 1000-step simulation, there would be 1000 different values \( X \), but predicate abstraction reduces them to three values as described in Fig. 5. The state diagram below upholds the following LTL condition: \( \Box(\Diamond PV = 1) \). This formula represents the stability of PID controller by ensuring the displacement reaches the user-specified value.
5 Conclusions and Further Work

The main contribution of this paper is a set of techniques for run-time analysis of temporal constrained objects. The execution of temporal constrained objects involves a time-based simulation together with constraint solving at each time-step, where the constraints could involve ordinary variables as well as series variables, which may take different values at each time-step. We have developed techniques for error detection as well as reasoning about the correctness of execution, i.e., run-time verification. In the former case, we make use a temporal constraint dependency graph and in the latter case, we construct finite state machines which serve as a basis for (run-time) model-checking using propositional temporal logic. In both cases, we make crucial use of the execution trace of the program.

The ideas discussed in this paper have been implemented as part of the TCOB language and execution environment. The TCOB compiler translates TCOB programs to SWI-Prolog programs. Run-time analysis is carried out starting from the execution trace that is generated based upon the monitor clauses in the program. Temporal constrained objects simplify the task of constructing an execution trace, because this is implicitly present in the values bound to the series variables. The state diagrams were constructed using the PlantUML drawing package. As part of our future work, we propose to apply our run-time analysis methodology to larger applications and combine execution, visualization, error analysis and run-time verification in more integrated manner.

References

Appendix

A. TCOB Grammar

```
class_definition ::= [abstract] class class_id [extends class_id] { body }
body ::= [attributes attributes]
       [constraints constraints]
       [predicates predicates]
       [constructors constructors]
attributes ::= [decl ;] +
decl ::= [series] type id_list
type ::= primitive | class_id | type [ ]
primitive ::= real | int | bool | char | string
id_list ::= attribute_id [ , attribute_id ] +
constraints ::= [constraint ;] +
constraint ::= creational | quantified | simple
creational ::= attribute = new class_id (terms)
quantified ::= forall var in enum : constraint |
               exists var in enum : constraint
simple ::= conditional | constraint_atom
conditional ::= literals --> condi_body
condi_body ::= mto_literals [& mto_literals] +
mto_literals ::= literal | mto_constraint
mto_constraint ::= F constraint_atom | F < interval > constraint_atom |
                 F < interval, interval > constraint_atom | G constraint_atom |
                 G < interval > constraint_atom |
                 G < interval, interval > constraint_atom
constraint_atom ::= term relop term | cpred_id (terms)
relop ::= = | != | > | < | >= | <=
```
B. PID Controller

class controller {
    attributes
    int Kp, Ki, Kd;
    series real Error, ESum, Out;
    constraints
    Kp > 0; Kd > 0; Ki > 0;
    ESum = Error + 'ESum;
    Out = Kp * Error + Ki * ESum
    + Kd * (Error - 'Error);
    constructors
    controller(KP, KI, KD) {
        Kp = KP; Ki = KI; Kd = KD;
        Error<1> = 0; ESum<1> = 0;
        Error<2> = 0; ESum<2> = 0;
    }
}

class plant {
    attributes
    series real Fo, X, V;
    int M, B, K;
    constraints
    V = X - 'X;
    Fo = M * (V - 'V) + B * V + K * X;
    constructor
    plant(M1, B1, K1, X1, V1) {
        M = M1; B = B1; K = K1;
        X<1> = X1; V<1> = V1;
    }
}

class sensor {
    attributes
    series real Output;
    plant P;
    constraints
    P.X = Output;
    constructor sensor(P1) {
        P = P1;
    }
}

class system {
    attributes
    plant P; controller C;
    sensor S; real Dvalue;
    constraints
    P.Fo = C.Out;
    Dvalue = C.Error' + S.Output;
    constructor system() {
        P = new plant(1, 10, 20, 0, 1);
        C = new controller(1, 1, 1);
        S = new sensor(P);
        Dvalue = 1;
    }
}
Implementation of Logical Retraction in
Constraint Handling Rules with Justifications

Thom Frühwirth
Ulm University, Germany
thom.fruhwirth@uni-ulm.de

Abstract. In previous work we added justifications to Constraint Handling Rules (CHR) to enable logical retraction of constraints for dynamic algorithms. We presented a straightforward source-to-source transformation to implement this conservative extension. In this companion paper, we improve the performance of the transformation. We discuss its worst-case time complexity in general. Then we perform experiments. We benchmark the dynamic problem of maintaining shortest paths under addition and retraction of paths. The results validate our complexity considerations.

1 Introduction

Justifications have their origin in truth maintenance systems (TMS) [7] for automated reasoning. Derived information (a formula) is explicitly stored and associated with the information it originates from by means of justifications. With the help of justifications, conclusions can be withdrawn (undone) by retracting their premises. By this logical retraction, inconsistencies can be repaired by retracting one of the reasons for the inconsistency.

In the formalism and programming language Constraint Handling Rules (CHR) [5, 6], conjunctions of atomic formulae (constraints) are rewritten by rule applications. When algorithms are written in CHR, constraints represent both data and operations. CHR is already incremental by nature, i.e. constraints can be added at runtime. Logical retraction then adds decrementality. To accomplish logical retraction in CHR, we have to be aware that CHR constraints can also be deleted by rule applications. These constraints may have to be restored when a premise constraint is retracted. With logical retraction, any algorithm written in CHR becomes fully dynamic1. Operations can be undone and data can be removed at any point in the computation without compromising the correctness of the result.

In [3], we formally defined a correct conservative extension of CHR with justifications (CHR-J). We gave a straightforward source-to-source transformation that adds justifications for user-defined constraints. A scheme of two rules sufficed to allow for logical retraction (deletion, removal) of constraints during

1 Dynamic algorithms for dynamic problems should not be confused with dynamic programming.
computation. Without the need to recompute from scratch, these rules retract not only the constraint but also undo all consequences of the rule applications that involved the constraint.

The runtime performance of the previous translation scheme is not optimal, however. In this paper, we present an improved source-to-source transformation for logical retraction of constraints with justifications in CHR (CHR\(^J\)). This transformation only imposes a constant factor overhead as long as justifications are not used for retraction. We will argue that the worst-case time complexity for any number of retractions is in general proportional to the number of rule applications, i.e. derivation length. The complexity of an algorithm expressed in CHR is usually a polynomial in the derivation length. Therefore retraction indeed has typically less complexity than recomputation from scratch at the expense of storing removed constraints. The added space complexity is again bounded by the derivation length. In our experiments, we will consider the dynamic problem of maintaining shortest paths under addition and retraction of paths.

Minimum Example. Given a multiset of numbers represented as conjunction \(\min(n_1), \min(n_2), \ldots, \min(n_k)\). The constraint (predicate) \(\min(n_i)\) means that the number \(n_i\) is a candidate for the minimum value. The following CHR rule filters the candidates.

\[
\min(N) \setminus \min(M) \leftrightarrow N=M \mid \text{true}.
\]

The rule consists of a left-hand side, on which a pair of constraints has to be matched, a guard check \(N=M\) that has to be satisfied, and an empty right-hand side denoted by \(\text{true}\). In effect, the rule takes two \(\min\) candidates and removes the one with the larger value (constraints after the \(\setminus\) symbol are deleted). Note that the \(\min\) constraints behave both as operations (removing other constraints) and as data (being removed).

CHR rules are applied exhaustively. Here the rule keeps on going until only one, thus the smallest value, remains as single \(\min\) constraint, denoting the current minimum. If another \(\min\) constraint is added during the computation, it will eventually react with a previous \(\min\) constraint, and the correct current minimum will be computed in the end. Thus the algorithm as implemented in CHR is incremental. It is not decremental, though: We cannot logically retract a \(\min\) candidate. While removing a candidate that is larger than the minimum would be trivial, the retraction of the minimum itself requires to remember all deleted candidates and to find their minimum. As we will see, with the help of justifications, this logical retraction will be possible automatically.

Related Work. The work of Armin Wolf on Adaptive CHR [10] introduced justifications into CHR. Different to our work, this technically involved approach requires to store detailed information about the rule instances that have been applied in a derivation in order to undo them. Adaptive CHR had a low-level implementation in Java [9], while we give an implementation in CHR itself by source-to-source transformations.
The more recent work of Gregory Duck [1] introduces SMCHR, a tight integration of CHR with a Boolean Satisfiability (SAT) solver for quantifier-free formulae including disjunction and negation as logical connectives. It is mentioned that for clause generation, SMCHR supports justifications for constraints.

Overview of the Paper. In the next section we recall abstract syntax and refined operational semantics for CHR. In Section 3, we describe CHR with justifications for logical retraction of constraints and its previous implementation by a straightforward source-to-source transformation. In Section 4, our current work is to optimize this implementation and to discuss its worst-case run-time complexity. In Section 5, we report on the results of experiments with our new implementation for the dynamic problem of maintaining shortest paths in a graph under addition (insertion) and deletion (retraction) of paths. The paper ends with conclusions and directions for future work.

2 Preliminaries

We recall abstract syntax and refined operational semantics of CHR [5] in this section.

2.1 Abstract Syntax of CHR

Constraints are relations, distinguished predicates of first-order predicate logic. We differentiate between two kinds of constraints: built-in (pre-defined) constraints and user-defined (CHR) constraints which are defined by the rules in a CHR program.

Definition 1. A CHR program is a finite set of rules. A (generalized) simpagation rule is of the form

\[ r : H_1 \setminus H_2 \Leftrightarrow C \mid B \]

where \( r \) is an optional name (a unique identifier) of a rule. In the rule head (left-hand side), \( H_1 \) and \( H_2 \) are conjunctions of user-defined constraints, the optional guard \( C \) is a conjunction of built-in constraints, and the body (right-hand side) \( B \) is a goal. A goal is a conjunction of built-in and user-defined constraints. A state is a goal. Conjunctions are understood as multisets of their conjuncts.

In the rule, \( H_1 \) are called the kept constraints, while \( H_2 \) are called the removed constraints. At least one of \( H_1 \) and \( H_2 \) must be non-empty. If \( H_1 \) is empty, the rule corresponds to a simplification rule, also written

\[ s : H_2 \Leftrightarrow C \mid B. \]

If \( H_2 \) is empty, the rule corresponds to a propagation rule, also written

\[ p : H_1 \Rightarrow C \mid B. \]

In this work, we restrict given CHR programs to rules without built-in constraints in the body except true and false.
2.2 Operational Semantics of CHR

We follow the exposition in [8] in this subsection. Given a query, the rules of the program are applied to exhaustion. A rule is applicable, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard check of the rule holds. More formally, the guard is logically implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice (in contrast to Prolog). When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule, when a propagation rule is applied, the body of the rule is added to the goal without removing any constraints. When a simpagation rule is applied, only the head constraints right to the backslash symbol are removed, the head constraints before are kept.

As in Prolog, almost all CHR implementations execute queries from left to right and apply rules top-down in the textual order of the program. This behavior has been formalized in the so-called refined semantics that was also proven to be a concretization of the standard operational semantics [2]. In this refined semantics of actual implementations, a CHR constraint in a query can be understood as a procedure that goes efficiently through the rules of the program in the order they are written, and when it matches a head constraint of a rule, it will look for the other, partner constraints of the head in the constraint store and check the guard until an applicable rule is found. We consider such a constraint to be active. If the active constraint has not been removed after trying all rules, it will be delayed and put into the constraint store as data. Constraints from the store will be reconsidered (woken) if newly added built-in constraints constrain variables of the constraint, because then rules may become applicable since their guards are now implied.

Hash Indexing in CHR. To achieve optimal time complexity, (near) constant-time addition, finding and removal of CHR constraints is required. Most current CHR libraries in Prolog are based on the KU Leuven CHR system. It supports indexes for terms via attributed variables, and in SWI Prolog also hash tables for ground terms and arrays for dense integers. The HAL CHR system and few other implementations also feature balanced trees, which are usually somewhat slower than hash tables. The hash table based indexes in SWI Prolog work at the argument level. In other words, for efficient constraint lookups, these arguments have to be ground during computation. Thus, for optimal performance, the SWI Prolog CHR system depends on mode and type information specified in constraint declarations.

3 CHR with Justifications (CHR$^J$)

We present a conservative extension of CHR by justifications following [3]. If justifications are not used, programs behave as without them. Justifications annotate atomic CHR constraints. A straightforward source-to-source transformation extends the rules with justifications.
3.1 CHR with Justifications for Logical Retraction

We start with adding justifications to CHR constraints and states.

**Definition 2 (CHR Constraints and Initial States with Justifications).**
A justification \( f \) is a unique identifier. Given an atomic CHR constraint \( G \), a CHR constraint with justifications is of the form \( G^F \), where \( F \) is a set of justifications.

An initial state with justifications is of the form \( \bigwedge_{i=1}^n G_i^{f_i} \) where the \( f_i \) are distinct justifications.

We now define a source-to-source translation from rules to rules with justifications. Let \( \text{kill} \) (retract) and \( \text{rem} \) (remember removed) be to unary reserved CHR constraint symbols. This means they are only allowed to occur in rules as specified in the following.

**Definition 3 (Translation to Rules with Justifications).**
Given a generalized simpagation rule
\[
    r : \bigwedge_{i=1}^l K_i \setminus \bigwedge_{j=1}^m R_j \leftarrow C \mid \bigwedge_{k=1}^n B_k
\]
Its translation to a simpagation rule with justifications is of the form
\[
    rf : \bigwedge_{i=1}^l K_i^{F_i} \setminus \bigwedge_{j=1}^m R_j^{F_j} \leftarrow C \mid \bigwedge_{j=1}^m \text{rem}(R_j^{F_j})^F \setminus \bigwedge_{k=1}^n B_k^F
\]
where \( F = \bigcup_{i=1}^l F_i \cup \bigcup_{j=1}^m F_j \).

The translation ensures that the head and the body of a rule mention exactly the same justifications. The reserved CHR constraint \( \text{rem}/1 \) (remember removed) stores the constraints removed by the rule together with their justifications.

**Shorthand Notation.** By abuse of notation, let \( A^J, B^J, C^J \ldots \) be conjunctions or corresponding states whose atomic CHR constraints are annotated with justifications according to the above definition of the rule scheme. Similarly, let \( \text{rem}(R)^J \) denote the conjunction \( \bigwedge_{j=1}^m \text{rem}(R_j^{F_j})^F \).

We showed previously that rule applications correspond to each other in standard CHR and in CHR\(^J\).

**Lemma 1 (Equivalence of Program Rules).** [3] There is a computation step \( S \rightarrow_r T \) with simpagation rule
\[
    r : H_1 \setminus H_2 \leftarrow C \mid B
\]
if and only if there is a computation step with justifications \( S^J \rightarrow_{rf} T^J \wedge \text{rem}(H_2)^J \) with the corresponding simpagation rule with justifications
\[
    rf : H_1^J \setminus H_2^J \leftarrow C \mid \text{rem}(H_2)^J \wedge B^J.
\]
Since computations are sequences of connected computation steps, this lemma implies that computations in standard CHR program and in CHR\(^J\) correspond to each other. Thus CHR with justifications is a conservative extension of CHR.
**Logical Retraction Using Justifications.** We use justifications to retract a CHR constraint from a computation without the need to recompute from scratch. This means that all its consequences due to rule applications it was involved in are undone. CHR constraints added by those rules are removed and CHR constraints removed by the rules are re-added (inserted). To specify and implement this behavior, we give a scheme of two rules, one for retraction and one for re-adding of constraints. The reserved CHR constraint \( \text{kill}(f) \) (retract) undoes all consequences of the constraint with justification \( f \).

**Definition 4 (Rules for CHR Logical Retraction).** For each \( n \)-ary CHR constraint symbol \( c \) (except the reserved \( \text{kill} \) and \( \text{rem} \)), we add a rule to kill constraints and a rule to revive removed constraints of the form:

\[
\begin{align*}
\text{kill} &: \text{kill}(f) \setminus G^F \leftrightarrow f \in F | \text{true} \\
\text{revive} &: \text{kill}(f) \setminus \text{rem}(G^F) \leftrightarrow f \in F | G^F,
\end{align*}
\]

where \( G = c(X_1, \ldots, X_n) \), where \( X_1, \ldots, X_n \) are different variables.

Note that a constraint may be revived and subsequently killed. This is the case when both \( F_c \) and \( F \) contain the justification \( f \).

We proved previously correctness of logical retraction: the result of a computation with retraction is the same as if the constraint would never have been introduced in the computation. We showed that given a computation starting from an initial state with a \( \text{kill}(f) \) constraint that ends in a state where the \( \text{kill} \) and \( \text{revive} \) rules have been applied to exhaustion, then there is a corresponding computation without constraints that contain the justification \( f \).

**Theorem 1 (Correctness of Logical Retraction).** [3] Given a computation

\[
A^J \land G^f \land \text{kill}(f) \mapsto^* B^J \land \text{rem}(R)^J \land \text{kill}(f) \not\mapsto \text{kill,revive},
\]

where \( f \) does not occur in \( A^J \). Then there is a computation without \( G^f \) and \( \text{kill}(f) \)

\[
A^J \mapsto^* B^J \land \text{rem}(R)^J.
\]

### 3.2 Previous Implementation

We recall the implementation of [3] for CHR with justifications (CHR\(^J\)).

**Constraints with Justifications.** CHR constraints annotated by a set of justifications are realized by a binary infix operator \( \#\# \), where the second argument is a list of justifications:

\( C^{(F_1,F_2,\ldots)} \) is realized as \( C \#\# [F_1,F_2,\ldots] \).

For convenience, we add rules that add a new justification to a given constraint \( C \). For each constraint symbol \( c \) with arity \( n \) there is a rule of the form

\[
\text{addjust} @ c(X_1,X_2,\ldots X_n) \leftrightarrow c(X_1,X_2,\ldots X_n) \#\# [F].
\]

where the arguments of \( X_1,X_2,\ldots X_n \) are different variables.
Rules with Justifications. A CHR simpagation rule with justifications is realized as follows:

\[
rf : \bigwedge_{i=1}^{l} K_{Fi} \setminus \bigwedge_{j=1}^{m} R_{Fj} \Leftrightarrow C \mid \bigwedge_{i=1}^{m} \text{rem}(R_{Fj}) \neq \bigwedge_{k=1}^{n} B_{Fk} \text{ where } F = \bigcup_{i=1}^{l} F_{i} \cup \bigcup_{j=1}^{m} F_{j}
\]

\[
rf \circ K_{1} \# \# F_{K_{1}},... \setminus R_{1} \# \# F_{R_{1}},... \Leftrightarrow C \mid \\
\text{union([FK_{1},... FR_{1},...],Fs), rem(R_{1}##FR_{1}) ## Fs,...B_{1} ## Fs,...}
\]

where the auxiliary predicate \text{union}/2 computes the ordered duplicate-free union of a list of lists\(^2\).

Rules kill, remove and revive. Justifications are realized as flags that are initially unbound logical variables. This eases the generation of new unique justifications and their use in killing. Concretely, the reserved constraint \text{kill}(f) is realized as built-in equality F=r, i.e. the justification variable gets bound. If \text{kill}(f) occurred in the head of a \text{kill} or \text{revive} rule, it is moved to the guard as equality test F==r.

\[
\text{revive} : \text{kill}(f) \setminus \text{rem}(C_{F_{c}}) \Leftrightarrow f \in F \mid C_{F_{c}}
\]

\[
\text{kill} : \text{kill}(f) \setminus \text{C} \Leftrightarrow f \in F \mid \text{true}
\]

\[
\text{revive} \circ \text{rem}(C##FC) \# \# Fs \Leftrightarrow \text{member}(F,Fs),F=r \mid C \# \# FC.
\]

\[
\text{remove} \circ C \# \# Fs \Leftrightarrow \text{notfunctor}(C,\text{rem}),\text{member}(F,Fs),F=r \mid \text{true}.
\]

The check \text{notfunctor}(C,rem) ensures that C is not a \text{rem} constraint. The check for set membership in the guards is expressed using the standard nondeterministic Prolog built-in predicate \text{member}/2.

Logical Retraction with \text{killc}/1. We extend the translation to allow for retraction of derived constraints. The constraint \text{killc}(C) logically retracts constraint C. The two rules \text{killc} and \text{killr} try to find the constraint C - also when it has been removed and is now present in a \text{rem} constraint. The associated justifications point to all initial constraints that where involved in producing the constraint C. For retracting the constraint, it is sufficient to remove one of its producers. This introduces a choice implemented by the \text{member} predicate.

\[
\text{killr} \circ \text{killc}(C), \text{rem}(C##FC) \# \# Fs \Leftrightarrow \text{member}(F,FC),F=r.
\]

\[
\text{killc} \circ \text{killc}(C), C \# \# Fs \Leftrightarrow \text{member}(F,Fs),F=r.
\]

Note that in the first rule, we bind a justification F from FC, because FC contains the justifications of the producers of constraint C, while Fs also contains those that removed it by a rule application.

\(^2\) More precisely, a simplification rule is generated if there are no kept constraints and a propagation rule is generated if there are no removed constraints.
**Dynamic Minimum Example.** Translating the minimum rule to one with justifications results in:

\[
\text{min}(A)\#B \lor \text{min}(C)\#D \iff A < C \lor \text{union}([B,D],E), \text{rem}(\text{min}(C)\#D)\#E.
\]

The following shows an example query and the resulting answer in SWI-Prolog:

```prolog
?- min(1)\#[A], min(0)\#[B], min(2)\#[C].
\text{rem}(\text{min}(1)\#[A])\#[A,B], \text{rem}(\text{min}(2)\#[C])\#[B,C], \text{min}(0)\#[B].
```

The constraint \text{min}(0) remained. This means that 0 is the minimum. The constraints \text{min}(1) and \text{min}(2) have been removed and are now remembered. Both have been removed by the constraint with justification B, i.e. \text{min}(0).

We now logically retract with \text{killc} the constraint \text{min}(1) at the end of the query. The \text{killr} rule applies and removes \text{rem}(\text{min}(1)\#[A])\#[A,B]. (In the rule body, the justification A is bound to r - to no effect, since there are no other constraints with this justification.)

```prolog
?- min(1)\#[A], min(0)\#[B], min(2)\#[C], \text{killc}(\text{min}(1)).
\text{rem}(\text{min}(2)\#[C])\#[A,C], \text{min}(0)\#[B].
```

What happens if we retract the current minimum \text{min}(0)? The constraint \text{min}(0) is removed by binding justification B. The two \text{rem} constraints for \text{min}(1) and \text{min}(2) involve B as well, so these two constraints are re-introduced and react with each other. Note that \text{min}(2) is now removed by \text{min}(1) (before it was \text{min}(0)). The result is the updated minimum, which is 1.

```prolog
?- min(1)\#[A], min(0)\#[B], min(2)\#[C], \text{killc}(\text{min}(0)).
\text{rem}(\text{min}(2)\#[C])\#[A,C], \text{min}(1)\#[B].
```

### 4 Optimizing the Implementation

We would like to avoid any overhead complexity-wise when computing with justifications as long as we do not use them for retraction. We are ready to accept a constant factor penalty. While the insertion of \text{rem} constraints takes constant time, the computation of the union of justifications is linear in the sizes of its input justification sets. The idea is to delay this computation until it is needed due to a retraction. We actually never compute the union of justifications, but will use the \text{union} constraints as data to find the necessary justifications. We describe the modifications for this new implementations and then discuss the complexity of this approach.

#### 4.1 New Improved Implementation

To retract a constraint with justification F, the constraint \text{killd}(F) (kill down) finds its initial justifications. The arguments of the delayed \text{union} constraints are unbound variables now (except for the singleton sets of the justifications from the initial constraints in the query). The constraint \text{killd} has to find the \text{union}
constraint with its justification in the output and follow all its input justifications (which are represented by a list). It proceeds recursively with the help of killl (kill list) until it reaches an initial justification. On the way, we can stop if we see a justification again that we have already seen.

\[
\text{already_seen} \circ \text{killd}(F) \setminus \text{killd}(F) \iff \text{true}.
\]
\[
\text{go_to_initial} \circ \text{union}(FL,F) \setminus \text{killd}(F) \iff \text{killl}(FL).
\]

\[
\text{killl} \circ \text{killl}([]) \iff \text{true}.
\]
\[
\text{killl} \circ \text{killl}(\{F\}) \iff \text{true}.
\]
\[
\text{killl} \circ \text{killl}(\{F|FL\}) \iff \text{killd}(F), \text{killl}(FL).
\]

Then the auxiliary constraint killone (kill one) chooses one of these justifications in turn and removes it.

\[
\text{choice} \circ \text{killone}, \text{killd}(\{F\}) \iff (F=r, \text{waitrem} ; \text{killone}).
\]
\[
\text{done} \circ \text{killone} \iff \text{false}.
\]

The rule choice uses Prolog’s disjunction in the body. In the first disjunct, the binding of justification \( F \) to the constant \( r \) marks it as to be killed and wakes up all constraints in which this justification occurs. In this way, constraints are retracted and revived, respectively. The auxiliary constraint waitrem delays re-addition of previously removed constraints via the rule revive until all constraints have been retracted by the remove rule. This improves the performance. The recursion on killone in the second disjunct ensures that all justifications are eventually tried. Note that as a consequence, in rule done we must fail (not succeed), since we then have exhausted trying all justifications.

Now we also have to kill all output justifications of unions that have this killed justification as input justification, i.e. we go upwards.

\[
\text{go_upwards} \circ \text{union}(FL,F) \iff \text{member}(F1,FL), F1==[r] \mid F=[r].
\]

Note that we will only pass a subset of the union constraints that killld visited, those that involve the chosen initial justification. We will also pass additional other union constraints as consequence of this.

Finally, for retraction, we remove constraints with killed justifications and we revive remembered constraints with killed justifications. We translate program constraints \( C \) with justifications \( F \) of the form \( c(X1,\ldots,Xn)##F \) into \( c(X1,\ldots,Xn,F) \) to support argument-wise indexing if necessary.

\[
\text{remove} \circ c(X1,\ldots,Xn,[r]) \iff \text{true}.
\]
\[
\text{revive} \circ \text{waitrem} \setminus \text{rem}(c(X1,\ldots,Xn,FC),[r]) \iff c(X1,\ldots,Xn,FC).
\]
\[
\text{waitrem} \iff \text{true}.
\]

Here we put waitrem to work to trigger the re-addition of constraints in the revive rule. Having done so, waitrem is removed at the very end.
4.2 Worst-Case Time Complexity

We now discuss the complexity of our optimized implementation in terms of the input size and derivation length following the principles of [4]. Let \( k \) be the largest number of head constraints in a given program. Note that \( k \) is a constant. Let \( c \) be the number of CHR constraints in the initial state (query). Let \( n \) be the derivation length of a computation, i.e. the number of rule applications (transitions).

The complexity of the original computation is at least \( n \), because there are \( n \) rule applications that take at least constant time each. If the computation does not fail, each initial constraint is processed, which adds \( c \) to the lower bound of the complexity, which thus is \( n + c \). Typically, \( n \) is larger than \( c \), so we may assume just \( n \).

All rule tries (application attempts) and rule applications take constant time, mostly because of the index on the justification. There is no overhead in runtime complexity until a constraint is killed: the union constraint and the rem constraints are just added to the constraint store. Since the number of rem constraints is bounded by \( k \), complexity does not increase, if constraints can be added (inserted) in constant time. Based on these observations, we can also see that the space complexity is bounded by \( O(n) \).

The union constraints have at most \( k \) input justifications that already have been introduced. The result is the output justification, represented by a new fresh logical variable. The union constraints form a directed acyclic graph (dag) with bounded width \( k \), where the nodes are the justification set variables and where there is an directed edge (arc) from each input to the output justification for each union in a derivation. Since the output justification is always new, the corresponding graph is acyclic. It is typically not a tree, since a union may have input justifications from arbitrary previous unions.

There are at most \( n \) unions in a computation of length \( n \). Thus there are at most \( n + c \) new justification nodes and \( c \) initial justifications. Therefore we have at most \( n + c \) different nodes. The number of edges is at most \( k \) for each union and is therefore of order \( O(n) \). The constraint kill has to go along at most \( kn \) edges, pass at most \( n + c \) different nodes and stop at most at \( c \) initial justifications.

The constraint killone will chose the next initial justification in constant time. There may be up to \( c \) choices. Once we have chosen this initial justification to use for killing and retraction, we use the rule go_upwards to find all effected justifications with the help of the union constraint. We may have up to \( n \) non-initial justifications to revive and remove (kill) constraints in turn. Typically, the number will be much smaller, because \( n \) refers to all union constraints in the derivation. For each justification, there can only be a bounded number of remembered (\( k \)) and added constraints, because the number of head and body constraints in rules is bounded in a given program.

The killing of a justification and the retraction of constraints is accomplished by binding the justification variable. This will wake up all constraints in which the variable occurs. These are the union constraints and the all program constraints that have this justification. Thus the rule go_upwards and remove are
immediately applicable, while the \texttt{revive} rule applications have to wait for the constraint \texttt{waitrem}.

In summary, the overall worst-case complexity of retracting a constraint with one choice of an initial justification is of order $O(n)$ (assuming $n > c$). The complexity trying each of the up to $c$ found initial constraints is then $O(nc)$. Note that the complexity of removing all constraints or all initial $c$ justifications in a computation is also bounded by $O(n)$, since the number of remembered and added constraints is also of order $O(n)$.

The additional cost of processing the revived re-added constraints is of course dependent on the given program and has to be added to the above complexity results. In the worst case, it amounts to a complete recomputation from scratch (cf. minimum example). It may be constant in the best case. If all rules of the program can be tried and applied in constant time, the derivation length $n$ that was needed for $c$ initial constraints may provide a $O(n)$ worst case complexity for computations with the revived constraints, thus leaving the overall worst-case complexity at $O(n)$.

5 Experiments

Experiments were run with SWI Prolog 6.2.1. in standard configuration on an Apple Mac Mini with OS X 10.9.5 2,5 GHz Intel Core i5 and 4 GB RAM. For compilation of the CHR files debugging was switched off and full optimization enabled. We explicitly specified the arguments for indexing of program constraints in a declaration. This lead to a constant-factor improvement of the runtime over automatic indexing provided by the CHR compiler.

We also introduced \texttt{passive} declarations in the rules that handle the justifications for retraction where feasible. These annotate head constraints in rules. Such a constraint is then treated as data only that has to be searched for in the constraint store. No active code is generated for that constraint, i.e. it does not behave as an operation anymore that looks for its matching partner constraints. This optimizations avoids useless rule tries. Note that some of these passive constraints are also automatically inferred by the compiler.

The programs used can be found in the appendix of the full online version of this paper.

5.1 Dynamic All-Pair Shortest Paths

We want to find the shortest distance between all pairs of nodes in a complete directed graph whose edges are annotated with non-negative distances. Initially, for each edge, there is a corresponding path with the distance of the edge. For every other pair of nodes, the unknown distances are initialized with $\infty$. Then the following rule suffices to solve the problem:

\begin{verbatim}
shorten @ path(I,K,D1), path(K,J,D2) \ path(I,J,D3) <=>
  D4 is D1+D2, D3>D4 | path(I,J,D4).
\end{verbatim}
A currently shortest path between nodes I and J is replaced by the sum of the
distances between paths I to K and K to J if this new distance is shorter. Note
that the graph is complete. If the rule is not applicable anymore, all paths must
be shortest. From the shorten rule we generated the following rules augmented
with justifications

add_justification @ path(A,B,C) <=> path(A,B,C,[D]).
shorten @ path(A,B,C,D), path(B,E,F,G) \ path(A,E,H,I) <=>

Example. The answer output has been slightly edited to improve readability.

?-path(a,b,1),path(b,a,2),path(a,c,3),path(c,a,0),path(b,c,1),path(c,b,4).
rem(path(c,b,4,[A]),B), rem(path(a,c,3,[C]),D), rem(path(b,a,2,[E]),F),
union([[G],[H],[A]],B), union([[H],[I],[C]],D), union([[I],[G],[E]],F),
path(c,b,1,B), path(a,c,2,D), path(b,a,1,F), path(b,c,1,[I]),
path(c,a,0,[G]), path(a,b,1,[H])

Initial justifications are in square brackets as single elements of lists. Thus the
last three paths in the answer were not shortened, while the other three paths
were shortened once, as can be seen by the deleted original path/3 constraints
for them. From the first arguments of the delaying union/2 constraints we can
also read off the constraints that lead to a shorter path.

For our experiments, the shorten rule was then instrumented to count rule
tries (in the guard) and applications (in the body) with the help of Prolog’s global
variables. We explicitly added indexing information for the compiler because it
slightly improved the performance on our examples. This means there is an hash
index on the first and second argument of the path/4 constraint and it can also
be accessed without index.

Random Graph Generation and Shortest Paths. We generated complete graphs
from a given number of nodes represented by integers. For every pair of different
nodes, a path is generated with a random distance between 1 and the number
of nodes. This is accomplished by the rule:

gengraph(N), node(A), node(B) ==> random(1,N,D), path(A,B,D).

In Figure 1 the number of nodes of the random directed graph is given, leading to a quadratic number of paths. Column Apply reports the number of applications of the shorten rule, while column Try shows how often this rule
has been tried. Finally, Time reports the execution time in seconds. The time
is roughly proportional to the number of rules tries indicating that indexing
reduces the time for finding the three matching head constraints indeed to a
constant.
Complexity. Let $v$ be the number of nodes in the graph. There can be at most $v^2$ shortest path, one between each pair of nodes, so $c = v^2$. With indexes on the nodes in a path, the rule `shorten` can be applied in constant time, given one of the path constraints. The worst-case derivation length depends on the scheduling of paths for rule application. The optimal complexity is $O(v^3)$ when the scheduling of the Floyd-Warshall algorithm is used. It assumes an order on nodes and processes paths by their smallest nodes. We do not specify the scheduling and therefore expect a higher polynomial complexity in $v$. To reach the optimal complexity was not the scope of this work, since here we are interested in increasing the performance of logical retraction in comparison with the previous implementation.

Back to our experiments reported in Figure 1: for a complete graph with $v$ nodes and $v^2$ paths, the average execution time is of order $O(v^4)$ as was confirmed by computing the interpolating polynomial with WolframAlpha. This also holds for the number of rule tries and applications. So the derivation length $n$ is quadratic in the number of paths $c$, i.e. $O(c^2)$. The previous implementation has a similar complexity, but a higher constant factor.

Logical Retraction of Paths. In Figure 2 we can see that the times for retracting all shortest paths in a complete random directed graph vary. The columns `Apply` and `Try` refer to accumulated recomputations of shortest paths after retraction of paths. `Down` reports the number of rule applications for going to the initial justifications through `union` constraints, while `Up` counts the propagation of the
New Implementation

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Apply</th>
<th>Try</th>
<th>Down</th>
<th>Up</th>
<th>Remove</th>
<th>Revive</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>30</td>
<td>481</td>
<td>369</td>
<td>167</td>
<td>196</td>
<td>167</td>
<td>0.032</td>
</tr>
<tr>
<td>12</td>
<td>37</td>
<td>581</td>
<td>208</td>
<td>147</td>
<td>180</td>
<td>147</td>
<td>0.030</td>
</tr>
<tr>
<td>14</td>
<td>42</td>
<td>832</td>
<td>1990</td>
<td>242</td>
<td>281</td>
<td>242</td>
<td>0.075</td>
</tr>
<tr>
<td>14</td>
<td>54</td>
<td>894</td>
<td>1592</td>
<td>278</td>
<td>317</td>
<td>278</td>
<td>0.076</td>
</tr>
<tr>
<td>14</td>
<td>77</td>
<td>1245</td>
<td>1939</td>
<td>318</td>
<td>350</td>
<td>318</td>
<td>0.095</td>
</tr>
<tr>
<td>16</td>
<td>111</td>
<td>1901</td>
<td>5490</td>
<td>491</td>
<td>539</td>
<td>491</td>
<td>0.203</td>
</tr>
<tr>
<td>16</td>
<td>55</td>
<td>1413</td>
<td>3481</td>
<td>383</td>
<td>428</td>
<td>383</td>
<td>0.141</td>
</tr>
<tr>
<td>16</td>
<td>135</td>
<td>2513</td>
<td>1496</td>
<td>447</td>
<td>508</td>
<td>447</td>
<td>0.193</td>
</tr>
<tr>
<td>18</td>
<td>132</td>
<td>3158</td>
<td>7735</td>
<td>595</td>
<td>663</td>
<td>595</td>
<td>0.341</td>
</tr>
<tr>
<td>18</td>
<td>96</td>
<td>1869</td>
<td>4810</td>
<td>514</td>
<td>581</td>
<td>514</td>
<td>0.215</td>
</tr>
<tr>
<td>18</td>
<td>199</td>
<td>4598</td>
<td>8323</td>
<td>657</td>
<td>732</td>
<td>657</td>
<td>0.465</td>
</tr>
<tr>
<td>21</td>
<td>180</td>
<td>4963</td>
<td>51274</td>
<td>962</td>
<td>1033</td>
<td>962</td>
<td>1.170</td>
</tr>
<tr>
<td>21</td>
<td>207</td>
<td>4671</td>
<td>203119</td>
<td>917</td>
<td>966</td>
<td>917</td>
<td>2.980</td>
</tr>
<tr>
<td>21</td>
<td>188</td>
<td>4559</td>
<td>75441</td>
<td>908</td>
<td>985</td>
<td>908</td>
<td>1.381</td>
</tr>
</tbody>
</table>

Fig. 2. Removing All Shortest Paths from Random Graphs

killed justification to the roots. The counts for Down and thus the time needed vary, the variation seems to increase the larger the graph is. This number depends on the number of updates to particular intermediate shortest paths, i.e on the depth of the justification dag.

Remove and Revive show the number of actual removals of constraints and re-addition of previously removed path constraints. These last two numbers are similar, with slightly more removals than revivals. (Note that re-added constraints may be removed afterwards.) The numbers for Up and Revive are identical, because the shorten rule always removes a single path constraint.

Overall, the complexity is once again quartic, $O(v^4)$. This corresponds once again to the derivation length and thus is in line with our complexity considerations in the previous section. It also means that the overhead of the recomputations is neglectable complexity-wise. Indeed, comparing the two figures, we can see that it typically takes less time to remove each shortest paths one by one and recompute all effected paths each time than to compute all the shortest paths initially. Moreover, the numbers of path recomputations are about a fourth of the number of initial path computations.

Note that recomputing from scratch would result in $O(v^2)$ recomputations (one for each retracted path) of complexity $O(v^4)$ each and thus in a polynomial of higher degree. The previous implementation also has a worse polynomial complexity for retracting constraints. For a graph of size 14, the previous implementation is already about an order of magnitude slower.
6 Conclusions

We presented an improved source-to-source transformation for logical retraction of constraints with justifications in CHR (CHR\textsuperscript{J}). This transformation only imposes a constant factor overhead as long as justifications are not used for retraction. We argued that the worst-case time complexity for any number of retractions is in general proportional to the number of rule applications, i.e., derivation length. The complexity of an algorithm expressed in CHR is usually a polynomial in the derivation length. Therefore retraction indeed has typically less complexity than recomputation from scratch at the expense of storing removed constraints. The added space complexity is again bounded by the derivation length. In our experiments, we benchmarked the dynamic problem of maintaining shortest paths under addition and retraction of paths. The results verify our complexity considerations. For future work, we would like to further improve the implementation and benchmark it, taking care of proper indexing. At the same time, we would like to investigate how logical as well as classical algorithms like union-find behave when they become dynamic in CHR\textsuperscript{J}.

References

The Proportional Constraint and Its Pruning

Armin Wolf
IT4Energy Center
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
D-10589 Berlin
armin.wolf@fokus.fraunhofer.de

Abstract. Motivated by the necessity to model the energy loss of energy storage devices, a Proportional Constraint is introduced in finite integer domain Constraint Programming. Therefore rounding is used within its definition. For practical applications in finite domain Constraint Programming, pruning rules are presented and their correctness is proven. Further, it is shown by examples that the number of iterations necessary to reach a fixed-point while pruning depends on the considered constraint instances. However, fixed-point iteration always results in the strongest notion of bounds consistency. Furthermore, an alternative modeling of the Proportional Constraint is presented. The run-times of the implementations of both alternatives are compared showing that the implementation of the Proportional Constraint on the basis of the presented pruning rules performs always better on sample problem classes.

Keywords: bounds consistency, finite domain Constraint Programming, fixed-point iteration, Proportional Constraint, pruning rules

1 Motivation and Overview

Within the publicly funded project WaveSave\(^1\) we are concerning cost-optimized trans-sectoral operation plans for hybrid energy systems within buildings. Those energy systems may consist of Combined Heat and Power (CHP) systems, Photovoltaic (PV) systems, heat pumps, boilers etc. as well as energy storage systems like hot water tanks or batteries. The operation of such systems is time critical and highly dynamic: Such systems have to react immediately to deviations in order to ensure the energy supply of the buildings and theirs users. Deviations might be caused by disturbances or uncertain forecasts. In order to generate cost-optimal operations plans (aka schedules) for the components of such hybrid energy systems in buildings, we model them as Constraint Optimization Problems (COP).

\(^*\) The presented work is funded by the German Federal Ministry for Economic Affairs and Energy within the project “WaveSave” (BMWi, funding number 03ET1312A).

\(^{1}\) cf. http://www.it4energy-zentrum.de/de/it4energy/wavesave
For an evaluation of configurations of energy systems in buildings with respect to their overall costs including investment, operation, emission, maintenance etc. over their live-time, we applied Mixed Integer Programming (MIP) while using similar approaches as presented in [2–4]. In our approach the MIP models are automatically generated from a domain-specific XML descriptions defining the characteristics of the energy system components, forecast data on energy use and supply, current states of the energy system components, etc. For MIP modeling the Coliop Mathematical Programming Language (CMPL) (cf. http://www.coliop.org/) is used offering the opportunity to use different solvers like the open-source MIP solver CbC or the commercial MIP solver CPLEX.

To our knowledge it is rather difficult to consider domain-specific heuristics in MIP solvers, e.g. to consider preferences or soft constraints, e.g. in order to adapt schedules to changed constraints within the time-critical context of online operation scheduling of the energy systems components. Therefore we decided to apply finite integer domain Constraint Programming (fdCP) allowing heuristic search to model and solve such COP adequately within this highly dynamic context. This means that good solutions have to be found or adapted within reasonable short time.

Due to its nature, finite integer domain Constraint Programming only supports integer variables such that linear equations are considered as diophantine equations, i.e. only the integer solutions are sought. However, this is not suitable for any modeling of energy storage devices (cf., e.g. [3, 4]) in the context of the WaveSave project, which requires the consideration of energy losses of energy storages to the environment over time.

Example 1. Let an energy storage be given having a characteristic energy loss of 3 % within a given time unit. Further, let the energy load within time unit \( t \) be \( L_t \) of such a storage. Then the load \( L_{t+1} \) within the next time unit \( t+1 \) is determined by at least the part which is proportional to the factor of loss, i.e. \( L_{t+1} = 0.997 \cdot L_t + \ldots \).

Another situation in the energy context where the modeling of a proportional relationship with diophantine equations is not adequate occurs when the energetic behavior of CHP systems has to be modeled: Any CHP system has a specific current characteristic \( \sigma > 0 \) denoting the ratio of the supplied electric power \( P_{el} \) and the usable heat flow \( \dot{Q} \).

Example 2. Let a Stirling engine based CHP system be given having a current characteristic \( \sigma = 0.34 \) then it holds that the supplied electric power \( P_{el} \) is proportional to the usable heat flow \( \dot{Q} \) which varies over time according to the operation mode of the CHP system:

\[
P_{el} = 0.34 \cdot \dot{Q}.
\]

Assuming that the energy loads of a storage or the electric and thermal powers of CHP systems over time are decision variables \( A \) and \( B \), simple linear
equations like \( B = t \cdot A \) where \( t \in \mathbb{R} \) are modeled in a straight-forward manner in any Linear Programming system. However, in fdCP this is not the case. There, the decision variables have integer domains such that only integer solutions are considered. Consequently, \( L_{t+1} = 0.997 \cdot 45689 \) has no integral solution. However \( L_{t+1} = 45552 = \text{round}(0.997 \cdot 45689) \) seems to be an acceptable approximation in this case.

The work is organized as follows: First we present some related work, then we define the Proportional Constraint and some pruning rules. Further the correctness of these rules is proven and it is shown that iterative pruning leads to the strongest notion of bounds consistency. Then an alternative modeling of the Proportional Constraint based on linear inequalities is presented and the run-times of these two approaches on sample problem classes are compared. Finally we conclude with some remarks on the implementation and the use of this constraint.

2 Related Work

Linear equations \( y = \alpha \cdot x \) where the variable \( y \) is proportional to another variable \( x \) – where \( \alpha \) is a scalar value – are special cases of weighted sums, i.e. \( y = \alpha_1 \cdot x_1 + \cdots + \alpha_n \cdot x_n (n > 0) \). Weighted sums are already considered in [6]. Applying the pruning rules defined there on finite domain integer variables, the resulting consistency only ensures that there exist real solutions which is not adequate in the context of our WaveSave project. Thus we decided to extend our object-oriented constraint solving library firstCS [7] which already implements weighted sums with an adequate proportional constraint \( y = \alpha \cdot x \) for finite domain integer variables.

3 The Proportional Constraint

In application domains of finite domain Constraint Programming such as optimized operation of energy systems there is a need to model a proportional energy loss when using energy storage devices as already mentioned. Therefore and for other applications as well, we define the binary Proportional Constraint:

**Definition 1 (Proportional Constraint).** Let \( t > 0 \) be a real value and \( A, B \) finite domain constraint variables having integer domains \( \text{dom}(A) \) respective \( \text{dom}(B) \). For convenience, let \( \min(X) = \min(\text{dom}(X)) \) and \( \max(X) = \max(\text{dom}(X)) \) for any domain variable \( X \). The (binary) Proportional Constraint

\[
\text{round}(t \cdot A) = B
\]

is satisfied, if for any value \( a \in \text{dom}(A) \) there is a value \( b \in \text{dom}(B) \) respective if for any value \( b \in \text{dom}(B) \) there is a value \( a \in \text{dom}(A) \) such that \( \text{round}(t \cdot a) = b \) holds. Such value pairs \((a,b)\) or labelings \( \Theta = \{A \mapsto a, B \mapsto b\} \) satisfying the constraint are called solutions. There, \( \text{round}(\cdot) \) is the rounding function as defined by \( \text{round}(x) = \lfloor x + 0.5 \rfloor \), where \( \lfloor y \rfloor \) is the greatest integer value less than or equal to \( y \) for any real value \( y \).
The definition of the Proportional Constraint is sound in the sense that for any $t > 0$ and integer value $a$ there is another integer value $b$ such that $\text{round}(t \cdot a) = b$. For a there is obviously $b = \text{round}(t \cdot a)$. For $b$ and $0 < t \leq 1$ there is always an integer value $a \in \left[\left(\frac{b - 0.5}{t}, \frac{b + 0.5}{t}\right]\right]$ satisfying the constraint, because it holds

\[
\begin{align*}
t &\leq 1 \Leftrightarrow \quad t \leq (b + 0.5) - (b - 0.5) \\
&\Leftrightarrow \quad 1 \leq \frac{b + 0.5}{t} - \frac{b - 0.5}{t}
\end{align*}
\]

However, for $t > 1$ this is not always the case: Let $b = 1$ and $t = 1.9$, then there is not any integer value $a$ such that $1 = \text{round}(1.9 \cdot a)$ holds.

In order to implement and use such a constraint in a Constraint Programming system some pruning rules have to be defined, reducing the domains of the involved variables without losing any solutions and resulting in a fixed-point when iterated such that the pruned domains of the variables hopefully satisfy some notion of consistency. Our definition of some pruning rules for the Proportional Constraint requires a special kind of “floor” function mapping reals to integers. It is defined as follows:

**Definition 2.** For any real value $x$ let the function $\lfloor \cdot \rfloor : \mathbb{R} \rightarrow \mathbb{Z}$ be defined by

\[
\lfloor x \rfloor = \begin{cases} 
  x - 1 & \text{if } x = \lfloor x \rfloor, \\
  \lfloor x \rfloor & \text{otherwise.}
\end{cases}
\]

This definition of the function $\lfloor \cdot \rfloor$ is sound in the sense that for any $x \in \mathbb{R}$ there is exactly one $y \in \mathbb{Z}$ such that $y = \lfloor x \rfloor$ holds.

For the defined Proportional Constraint we propose the following pruning rules:

**Definition 3 (Pruning Rules).** For any Proportional constraint

\[
\text{round}(t \cdot A) = B
\]

with $t > 0$ and finite domain constraint variables $A$ and $B$ having integer domains $\text{dom}(A)$ respective $\text{dom}(B)$ let

\[
\begin{align*}
\text{dom}^*(B) &= \text{dom}(B) \cap \left[\text{round}(t \cdot \min(A)), \text{round}(t \cdot \max(A))\right] \\
\text{dom}^*(A) &= \text{dom}(A) \cap \left[\left\lfloor\left(\min^*(B) - 0.5\right)/t\right\rfloor, \left\lceil\left(\max^*(B) + 0.5\right)/t\right\rceil\right]
\end{align*}
\]

be some pruning rules where $\min^*(B) = \min(\text{dom}^*(B))$ and $\max^*(B) = \max(\text{dom}^*(B))$ for the sake of convenience. Further, $\lceil y \rceil$ is the smallest integer value greater than or equal to $y$ for any real value $y$.

These rules are potentially reducing the domains of $A$ and $B$, i.e. $\text{dom}^*(A) \subseteq \text{dom}(A)$ and $\text{dom}^*(B) \subseteq \text{dom}(B)$ hold.
The indicator * in $\text{dom}^*(A)$ respective in $\text{dom}^*(B)$ is used to distinguish between the original and the updated domains of the variables $A$ and $B$ which will replace $\text{dom}(A)$ respective $\text{dom}(B)$ in any next iteration of these pruning rules.

Example 3. Let the Proportional Constraint $\text{round}(2.1 \cdot A) = B$ with $\text{dom}(A) = \{0, 1, 2, 3\}$ and $\text{dom}(B) = \{2\}$ be given. After applying the pruning rules defined in Definition 3 it holds that $\text{dom}^*(B) = \{2\} \cap \{0, 6\} = \{2\}$ and $\text{dom}^*(A) = \{0, 1, 2, 3\} \cap [1, 1] = \{1\}$. This means that pruning determines the solution $\Theta = \{A \mapsto 1, B \mapsto 2\}$ correctly.

Obviously, the question arises whether the pruning rules are in general correct or whether there are any integer solutions of the constraint which will be lost while pruning? – The following proposition answers this question:

Proposition 1. Let $t > 0$ be a real value and $A, B$ finite domain constraint variables having integer domains $\text{dom}(A), \text{dom}(B)$. Further, let $\text{round}(t \cdot a) = b$ for any $a \in \text{dom}(A)$ and any $b \in \text{dom}(B)$. Then it also holds that $a \in \text{dom}^*(A)$ and $b \in \text{dom}^*(B)$. In other words the propagation rules are correct, i.e. there isn’t any integer solution lost.

Proof. Let $a \in \text{dom}(A)$ and $b \in \text{dom}(B)$ be any two integer values such that $\text{round}(t \cdot a) = b$ holds. Due to the fact that $\min(A) \leq a \leq \max(A)$ holds and the function $f(x) = \text{round}(t \cdot x)$ is monotonic, it holds $f(\min(A)) \leq f(a) \leq f(\max(A))$ and thus $b \in \text{dom}^*(B)$. According to the definition of $\text{round}$ exactly one of the following two cases is valid:

1. $t \cdot a = \text{round}(t \cdot a) + \varepsilon$ with $0 \leq \varepsilon < 0.5$,  
2. $t \cdot a = \text{round}(t \cdot a) - \varepsilon$ with $0 < \varepsilon \leq 0.5$.

Let us suppose that the first case is valid. Due to the facts that $b \in \text{dom}^*(B)$ and $b = \text{round}(t \cdot a)$ it holds that

$$
\left\lfloor (\min^*(B) - 0.5) / t \right\rfloor \leq \left\lceil (b - 0.5) / t \right\rceil \\
= \left\lceil (\text{round}(t \cdot a) - 0.5) / t \right\rceil \\
= \left\lceil (t \cdot a - \varepsilon) - 0.5 \right\rceil / t \\
= \left\lceil a - (0.5 + \varepsilon) / t \right\rceil \\
\leq a
$$

and

$$
\left\lceil \max^*(B) + 0.5 \right\rceil / t \geq \left\lceil (b + 0.5) / t \right\rceil \\
= \left\lceil (\text{round}(t \cdot a) + 0.5) / t \right\rceil \\
= \left\lceil (t \cdot a - \varepsilon) + 0.5 \right\rceil / t \\
= \left\lceil a + (0.5 - \varepsilon) / t \right\rceil \\
\geq a
$$
given that \((0.5 - \varepsilon)/t > 0\) holds.

Now, let us suppose that the second case is valid. Due to the facts that
\(b \in \text{dom}^* (B)\) and \(b = \text{round}(t \cdot a)\) it holds that

\[
\lceil (\min^*(B) - 0.5)/t \rceil \leq \lceil (b - 0.5)/t \rceil \\
= \lceil (\text{round}(t \cdot a) - 0.5)/t \rceil \\
= \lceil (t \cdot a + \varepsilon) - 0.5)/t \rceil \\
= \lceil a - (0.5 - \varepsilon)/t \rceil \\
\leq a
\]

(5)
given that \((0.5 - \varepsilon)/t \geq 0\) holds. Further it holds that

\[
\lceil (\max^*(B) + 0.5)/t \rceil \geq \lceil (b + 0.5)/t \rceil \\
= \lceil (\text{round}(t \cdot a) + 0.5)/t \rceil \\
= \lceil (t \cdot a + \varepsilon) + 0.5)/t \rceil \\
= \lceil a + (0.5 + \varepsilon)/t \rceil \\
\geq a
\]

(6)

An iterated application of the pruning rules defined in Definition 3 on the
constraint variables’ domains either reduces these finite domains until they be-
come empty or will not be further reduced. In any case, the iteration stops after
a finite number of steps, such that \(\text{dom}^*(A) = \text{dom}(A)\) and \(\text{dom}^*(B) = \text{dom}(B)\)
holds, i.e. a fixed-point is reached (cf. [1]). However, how many iterations are
necessary for reaching a fixed-point? – The following proposition answers this
question:

**Proposition 2.** The number of iterations of the pruning rules necessary to
reach a fixed-point has no fixed upper bound: The number of iterations strongly
depends on the constraint instance, in particular on the structure and on the size
of the domains of the variables.

**Example 4 (Counter Examples).** Let \(t = 3.0\) and for any integer value \(n > 1\) let
\(\text{dom}(A) = \{1, 2, 3, \ldots, n\}\) and \(\text{dom}(B) = \{3, 5, 8, \ldots, 3n - 1\}\) be given. Then a
fixed-point is reached after at least \(n - 1\) iterations. The same holds for \(t = 0.3\)
and any integer value \(n > 1\) if \(\text{dom}(A) = \{10, 20, 30, \ldots, 10 \cdot n\}\) and \(\text{dom}(B) = \{3, 5, 8, \ldots, 3n - 1\}\), then a fixed-point is reached after \(n - 1\) iterations, too.

Finally we show that after a fixed-point iteration of the pruning rules
(cf. Def. 3) the domains of the variables of the Proportional Constraint are
bounds consistent in the strongest sense – cf. [5] for a detailed analysis of different
notions of bounds consistency. From there we adopted the following definition:

\[^2\] The formal proof by induction is left to the interested reader.
Definition 4. A domain $D$ is bounds($D$) consistent for a constraint $c$ where $\text{vars}(c) = \{x_1, \ldots, x_n\}$, if for each variable $x_i$ with $1 \leq i \leq n$ and for each $d_i \in \{\min(x_i), \max(x_i)\}$ there exist integers $d_j$ with $d_j \in \text{dom}(x_j)$ where $1 \leq j \leq n$, $j \neq i$ such that the labeling $\Theta = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ is an integer solution of $c$.

This definition considers $n$-ary constraints and thus binary constraints like the Proportional Constraint as well.

Proposition 3. Let a Proportional Constraint

$$\text{round}(t \cdot A) = B$$

be given with $t > 0$ and finite domain constraint variables $A$ and $B$ having integer domains $\text{dom}(A)$ respective $\text{dom}(B)$. Furthermore, it is assumed that the pruning rules (cf. Def. 3) are iterated until a fixed-point is reached, i.e. it holds that $\text{dom}^n(A) = \text{dom}(A)$ and $\text{dom}^n(B) = \text{dom}(B)$. Then it holds that the domain $D$ (cf. Def. 4) consisting of $\text{dom}(A)$ and $\text{dom}(B)$ is bounds($D$) consistent.

Proof.

Let $a = \min(A)$. Then, there is an integer value $b$ such that $b = \text{round}(t \cdot a)$ holds. Now we assume that $b \notin \text{dom}(B)$ respective that $b \neq \min(B)$. It follows that $b < \min(B)$ due to the pruning rule (1) and the monotonicity of the rounding function. Thus $\text{round}(t \cdot a) \leq \min(B) - 1$ holds and further $t \cdot a \pm \varepsilon \leq \min(B) - 1$ (cf. case distinction in the proof of Proposition 1). This implies that $a < (\min(B) - 0.5)/t$ and finally $a < \lfloor(\min(B) - 0.5)/t \rfloor$ holds. This contradicts $a = \min(A)$, i.e. the assumption is wrong, it holds that $b \in \text{dom}(B)$ respective that $b = \min(B)$.

Let $a = \max(A)$. Then, there is an integer value $b$ such that $b = \text{round}(t \cdot a)$ holds. Now we assume that $b \notin \text{dom}(B)$ respective that $b \neq \max(B)$. It follows that $b > \max(B)$ due to the pruning rule (1) and the monotonicity of the rounding function. Thus $\text{round}(t \cdot a) \geq \max(B) + 1$ holds and further $t \cdot a \pm \varepsilon \geq \max(B) + 1$ (cf. case distinction in the proof of Proposition 1). This implies that $a \geq (\max(B) + 0.5)/t$ and finally $a > \lceil(\max(B) + 0.5)/t \rceil$ because $a$ is an integer value. This contradicts $a = \max(A)$, i.e. the assumption is wrong, it holds that $b \in \text{dom}(B)$ respective that $b = \max(B)$.

Let $b = \min(B)$. We further distinguish two additional sub-cases:

(a) We further suppose that there is an integer value $a$ such that $b = \text{round}(t \cdot a)$ holds. Now we assume that $a \notin \text{dom}(A)$ respective that $a \neq \min(A)$. It follows that $a < \min(A)$ due to the pruning rule (2) and the monotonicity of the rounding function. Thus $t \cdot a < \min(B) - 0.5$ holds implying that $b = \text{round}(t \cdot a) \leq t \cdot a + 0.5 < \min(B)$, i.e. the assumption was wrong, it holds that $a \in \text{dom}(A)$ respective that $a = \min(A)$.
(b) Now we assume that for each integer value $a$ it holds that $b \neq \text{round}(t \cdot a)$ even for $a = \min(A)$. Thus $b > \text{round}(t \cdot \min(A))$. According to the case distinction in the proof of Proposition 1 it holds that $b \geq t \cdot \min(A) \pm \varepsilon + 1$ and thus in either case $b \geq t \cdot \min(A) + 0.5$. It follows that $(\min(B) - 0.5)/t > \min(A)$ and thus $\lceil (\min(B) - 0.5)/t \rceil > \min(A)$ are holding. This contradicts the pruning rule (2). The assumption is wrong, i.e. there is an integer value $a$ such that $b = \text{round}(t \cdot a)$ holds. The case (b) never occurs.

Let $b = \max(B)$. Again, we further distinguish two additional sub-cases:

(a) We further suppose that there is an integer value $a$ such that $b = \text{round}(t \cdot a)$. Now we assume that $a \notin \text{dom}(A)$ respective that $a \neq \max(A)$. It follows that $a > \max(A)$ due to the pruning rule (2) and the monotonicity of the rounding function. Thus $t \cdot a \leq \max(B) + 0.5$. Due to the fact that $\max(B)$ is an integer value it holds that $b = \text{round}(t \cdot a) > \max(B)$, i.e. the assumption was wrong, it holds that $a \in \text{dom}(A)$ respective that $a = \max(A)$.

(b) Now we assume that for each integer value $a$ it holds that $b \neq \text{round}(t \cdot a)$ even for $a = \max(A)$. Thus $b < \text{round}(t \cdot \max(A))$. According to the case distinction in the proof of Proposition 1 it holds that $b \leq t \cdot \max(A) \pm \varepsilon - 1$ and thus in either case $b \leq t \cdot \max(A) - 0.5$. It follows that $(\max(B) + 0.5)/t \leq \max(A)$ and thus $\lceil (\max(B) + 0.5)/t \rceil < \max(A)$ are holding because $\max(A)$ is an integer value. This contradicts the pruning rule (2). The assumption is wrong, i.e. there is an integer value $a$ such that $b = \text{round}(t \cdot a)$ holds. The case (b) never occurs.

\[\square\]

4 Alternative Modeling of the Proportional Constraint

For any rational factor $t > 0$ within a Proportional Constraint $\text{round}(t \cdot A) = B$ it is possible to model this constraint equivalently on the basis of a weighted sum constraints which are well established in fdCP ([6]):

**Proposition 4.** Let $t = p/q$ where $p$ and $q$ are positive integer values. Then for any two finite domain constraint variables $A, B$ the constraint

\[-\frac{q}{2} < q \cdot B - p \cdot A \leq \frac{q}{2}\]  \hspace{1cm} (7)

is equivalent to the Proportional Constraint $\text{round}(t \cdot A) = B$ in the sense that any solution $\{A \mapsto a, B \mapsto b\}$ is a solution of (7) and vice-versa.

**Proof.** Let $\{A \mapsto a, B \mapsto b\}$ be a solution of the Proportional Constraint, i.e. $b = \text{round}(t \cdot a)$. According to the case distinction in the proof of Proposition 1 it holds $q \cdot B - p \cdot A = q \cdot \text{round}(p/q \cdot a) - p \cdot a = q \cdot p/q \cdot a \pm \varepsilon - p \cdot a = \pm \varepsilon$. Due

\[\text{Many thanks to the anonymous reviewer who suggested this approach.}\]
to the fact that $-q/2 < -\varepsilon$ and $\varepsilon \leq q/2$ holds for any $q > 1$, it also holds that \( \{A \mapsto a, B \mapsto b\} \) is a solution of (7).

Let \( \{A \mapsto a, B \mapsto b\} \) be a solution of (7), i.e. $-q/2 < q \cdot b - p \cdot a \leq q/2$ holds and thus $-1/2 < b - p/q \cdot a \leq 1/2$. Consequently, $b = \text{round}(t \cdot a)$ is the only integer value satisfying this condition and thus \( \{A \mapsto a, B \mapsto b\} \) is also a solution of the Proportional constraint.

\[ \square \]

5 Run-Time Comparison

For a run-time comparison of the Proportional Constraint and its alternative modeling (cf. (7)) we implemented the Proportional Constraint with the pruning rules presented in Definition 3 in our finite domain constraint solving library firstCS [7], which already offers weighted sum constraints. Then, we modeled the following classes of problem instances

- A(n): \( t = 0.3, \text{dom}(A) = \{10, 20, 30, \ldots, 10n\}, \text{dom}(B) = \{3, 5, 8, \ldots, 3n-1\} \).
- B(n): \( t = 0.997, \text{dom}(A) = \{1, 2, 3, \ldots, n\}, \text{dom}(B) = \{1, 2, 3, \ldots, n\} \).
- C(n): \( t = 0.003, \text{dom}(A) = \{1, 2, 3, \ldots, n\}, \text{dom}(B) = \{1, 2, 3, \ldots, n\} \).

For problem class A (cf. Example 4) we perform initial pruning resulting in one solution. For the problem classes B and C we perform initial pruning and additional pruning while searching all solutions. The search strategy chooses variable A then B and their values in increasing order performing backtracking in cases where dead ends are reached, i.e. a domain of a variable becomes empty.

The following table shows the results of our run-time comparison of the Proportional Constraint (PC) and its alternative modeling (ALT) executed on a Windows computer with Windows 10 Pro (64 bit), Intel i7 CPU, 2.60 GHz, 12 GByte RAM. The computations for all problem instances were repeated 10 times. We compared best run-times and average run-times (in msec.) showing that the execution of the Proportional Constraint is always faster – in the best cases 79 %, in the worst case 2 % and in average 45 %:

<table>
<thead>
<tr>
<th>Instance</th>
<th>PC avg</th>
<th>PC best</th>
<th>ALT avg</th>
<th>ALT best</th>
<th>ALT/PC avg</th>
<th>ALT/PC best</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(10000)</td>
<td>310.2</td>
<td>297</td>
<td>539.5</td>
<td>440</td>
<td>174 %</td>
<td>148 %</td>
</tr>
<tr>
<td>A(20000)</td>
<td>1238.8</td>
<td>1199</td>
<td>1277.8</td>
<td>1227</td>
<td>103 %</td>
<td>102 %</td>
</tr>
<tr>
<td>A(40000)</td>
<td>4035.3</td>
<td>3999</td>
<td>4985.4</td>
<td>4864</td>
<td>124 %</td>
<td>122 %</td>
</tr>
<tr>
<td>A(80000)</td>
<td>15582.6</td>
<td>15280</td>
<td>19188.7</td>
<td>18885</td>
<td>123 %</td>
<td>124 %</td>
</tr>
<tr>
<td>B(10000)</td>
<td>103.3</td>
<td>78</td>
<td>170.7</td>
<td>109</td>
<td>165 %</td>
<td>140 %</td>
</tr>
<tr>
<td>B(20000)</td>
<td>149.5</td>
<td>125</td>
<td>200.1</td>
<td>172</td>
<td>134 %</td>
<td>138 %</td>
</tr>
<tr>
<td>B(40000)</td>
<td>196.3</td>
<td>187</td>
<td>278.3</td>
<td>250</td>
<td>142 %</td>
<td>137 %</td>
</tr>
<tr>
<td>B(80000)</td>
<td>276.5</td>
<td>250</td>
<td>452.9</td>
<td>406</td>
<td>164 %</td>
<td>162 %</td>
</tr>
<tr>
<td>C(10000)</td>
<td>86.0</td>
<td>78</td>
<td>153.6</td>
<td>140</td>
<td>179 %</td>
<td>179 %</td>
</tr>
<tr>
<td>C(20000)</td>
<td>123.4</td>
<td>109</td>
<td>218.8</td>
<td>156</td>
<td>177 %</td>
<td>143 %</td>
</tr>
<tr>
<td>C(40000)</td>
<td>191.4</td>
<td>172</td>
<td>281.5</td>
<td>265</td>
<td>147 %</td>
<td>154 %</td>
</tr>
<tr>
<td>C(80000)</td>
<td>273.4</td>
<td>250</td>
<td>409.7</td>
<td>359</td>
<td>150 %</td>
<td>144 %</td>
</tr>
</tbody>
</table>
6 Conclusion

Within this work a Proportional Constraint for finite integer domains is defined and according pruning rules are presented and analyzed. It is shown that pruning based on these rules is correct and results in the strongest notion of bounds consistency (cf. [5]). The introduced Proportional Constraint is implemented in our object-oriented constraint solving library firstCS [7] and compared with an alternative approach based on linear inequalities already available in firstCS.

The Proportional Constraint is used in the context of the WaveSave project to model the energy loss of energy storages like heat tanks or batteries over time and the relationship between the supplied electric powers and the usable heat flows of CHP systems. It is noteworthy that for other applications we implemented a more general version of the Proportional Constraint for any $t \in \mathbb{R}$: For $t = 0$ the pruning rules are trivial: If $0 \not\in \text{dom}(B)$ holds, $\text{dom}^*(B)$ will become empty as well as $\text{dom}^*(A)$. If $0 \in \text{dom}(B)$ holds, $\text{dom}^*(B) = \{0\}$ and $\text{dom}^*(A) = \text{dom}(A)$ will hold. For $t < 0$ the pruning rules for $t > 0$ are adapted accordingly respecting the fact that $B = \text{round}(-t \cdot A)$ while pruning the domain of $A$ and $-B = \text{round}(-t \cdot A)$ while pruning the domain of $B$.

References

A Confluence Checker for Constraint Handling Rules with Persistent Constraints

Frank Richter, Daniel Gall, and Thom Frühwirth

Institute of Software Engineering and Programming Languages,
Ulm University, Germany

Abstract. In the abstract operational semantics of Constraint Handling Rules (CHR), propagation rules, i.e. rules that only add information, can be applied again and again. This trivial non-termination is typically avoided by a propagation history. A more declarative approach are persistent constraints. Constraints that are introduced by propagation rules are made persistent and cannot be removed. Now a propagation rule is only applied, if its derived constraints are not already persistent.

The operational semantics with persistent constraints $\omega_!$ differs substantially from other operational semantics, hence the standard confluence test cannot be applied. In this paper, a confluence test for $\omega_!$ is presented. Since $\omega_!$ breaks monotonicity of CHR, a weaker property is established that is shown to suffice for a decidable confluence criterion for terminating $\omega_!$ programs. The confluence test is implemented using a source to source transformation.

Keywords: Constraint Handling Rules, constraint programming, persistent constraints, confluence, propagation rules, source to source transformation

1 Introduction

Constraint Handling Rules (CHR) [3] is a declarative, multiset- and rule-based programming language. There exist several operational semantics.

The simplest and most basic operational semantics for CHR is the very abstract semantics. Its behavior is close to the logical reading of the rules. This leads to the problem of trivial non-termination with rules that do not remove any constraints and so are applicable any number of times. This class of rules is known as propagation rules. The way most operational semantics avoid this trivial non-termination problem is by adding a token store. This token store is used to ensure that a propagation rule is only applied once with the same constellation of constraints.

Such a token store can cause states with the same logical reading to exhibit a different operational behavior. It also hinders effective concurrent execution of CHR programs, since it needs to be distributed adequately [1].

The operational semantics with persistent constraints, denoted as $\omega_!$, adds a second constraint store for so called persistent constraints. Those are constraints
that represent any number of these constraints and cannot be removed. This allows it to stay close to the abstract semantics and avoid trivial non-termination without a token store, but the transition rules lose the monotonicity property, which is used in several proofs. CHR in general offers powerful program analyses and is suitable for concurrent execution [1]. Since \( \omega \) avoids a token store, its concurrent execution is not compromised. For a program to be easily used in a concurrent execution the property of confluence is in general important. It implies that the order in which the rules are executed does not influence the result.

The contribution to this topic presented in our work is:

- The introduction of a modified state equivalence definition for \( \omega \) to correct a flaw in the original definition. Additionally, a criterion for state equivalence according to the new definition is introduced. (Section 3)
- A confluence test for programs that terminate in \( \omega \). (Section 4)
- An implementation of \( \omega \) as source to source transformation. This transformation is based on the transformation presented in [2] and is realized in SWI Prolog. (Section 5.1)
- A tool that can check terminating programs for confluence in \( \omega \). The checker has an extension that offers support for more built-ins. This tool is a modification and extension of the confluence checker for the abstract semantics of CHR [6]. (Section 5)

The paper starts with a short introduction to CHR and the introduction of \( \omega \) in the preliminaries section. This is followed by a section with the extended state equivalence definition. The following section uses this definition to present a confluence test for \( \omega \). The next section builds on this by introducing a tool that can check programs for confluence in \( \omega \) with the help of a source to source implementation for \( \omega \).

2 Preliminaries

CHR is a rule based programming language, that needs a host language to provide support for built-in predicates. It consists of three different kind of rules. CHR has different operational semantics. This section starts by presenting the syntax of CHR. The very abstract semantics are introduced. Finally, the idea for the persistent semantics together with their definition is presented as they are originally introduced by [1].

2.1 Syntax

A CHR Program consists of a finite set of rules of the form \( r \bowtie H_k \leftarrow H_r \leftrightarrow C|B \). A rule has an optional name \( r \). There are built-in and CHR constraints of the form \( c(t_1, \ldots, t_n) \), where \( c \) is a constant symbol, \( n \) is the arity and \( t_1 \ldots t_n \) are first-order terms. Reasoning on built-in constraints can be done through...
a satisfaction-complete and decidable constraint theory CT, while CHR constraints are simply user defined constraints [2]. Each type of rule has a head that may not be empty and consists of CHR constraints, a guard $C$ that may be empty and consists of built-in constraints and a body $B$ that may not be empty. The body can consist of built-in constraints as well as CHR constraints. $H_k$ is the kept head and $H_r$ the removed head. In simplification rules the kept head is empty, while in propagation rules the removed head is empty and $\Rightarrow$ is used instead of $\Leftarrow$. If neither head is empty it is a simpagation rule [3, p. 54].

### 2.2 Very Abstract Semantics $\omega_{va}$

The very abstract operational semantics of CHR is given by a nondeterministic state transition system [3, p. 55].

**Definition 1 (State).** A state is a conjunction of built-in and CHR constraints. An initial state (initial goal) is an arbitrary state and a final state is one where no more transitions are possible [3, p. 56].

For the transitions, rules are used in head normal form (HNF). This means that each argument of a head constraint is a unique variable. A rule can be represented in HNF by replacing each of its head arguments $t_i$ with a new variable $V_i$ and adding the equation $V_i = t_i$ to the guard of the rule. The built-in $\neq /2$ for syntactic equivalence must be provided by CT. A transition represents a rule application according to the following transition relation of $\omega_{va}$:

$$\text{Apply}$$

$$(H_k \land H_r \land C) \Rightarrow_r (H_k \land C \land B \land G)$$

if there is an instance with new local variables $\pi$ of a rule named $r$ in $P$. 

$$r \in H_k \setminus H_r \Leftrightarrow G \land B \quad \text{and} \quad CT \models \exists \pi \exists G$$

The upper-case letters $H_k, H_r, G, B$ and $C$ represent conjunctions of constraints that can be empty. If $H_k$ and $H_r$ are present in the constraint store and $G$ holds, the rule is applicable and the CHR constraints $H_k$ are kept while the CHR constraints $H_r$ are removed. The resulting state additionally consists of the guard $G$ and the body $B$ [3, p. 56].

This transition system is nondeterministic, because if several rules are applicable one is chosen nondeterministically and this choice cannot be undone [3, p. 56].

Since a rule is always applicable if the head constraints are present and the guard is satisfied an applicable propagation rule stays applicable after any number of applications. This causes the aforementioned trivial non-termination.

### 2.3 Operational Semantics with Persistent Constraints $\omega_!$

The operational semantics for $\omega_!$ is based on three basic ideas:

1. Propagation rules in $\omega_{va}$ cause trivial non-termination, since given the corresponding head constraints are present in the constraint store the body can be generated any number of times. To avoid this kind of trivial non-termination a second constraint store is introduced in which those body constraints are
added. Constraints in this store are a finite representation of a very large, though unspecified number of identical constraints, so called persistent constraints. To differentiate between persistent and non-persistent constraints, non-persistent constraints are called linear constraints [1].

2. If the removed head of a rule in $\omega_{va}$ consists entirely of constraints that can be generated any number of time, the body of such a rule can also be generated any number of times given the constraints of the kept head are also present. To account for those indirect consequences of propagation rules, a rule’s body is introduced as persistent constraints, if its removed head is completely matched with persistent constraints [1].

3. Several occurrences of a persistent constraint are considered idempotent, since a persistent constraint represents an arbitrary number of identical constraints. For the execution model transitions are only supposed to happen if the post-transition state is not equivalent to the pre-transition state. This irreflexible transition system avoids trivial non-termination [1].

Definition 2 gives the definition for $\omega_{!}$ states [1].

**Definition 2.** ($\omega_{!}$-State).

A $\omega_{!}$-state is a tuple of the form $⟨L, P, B, V⟩$, where $L$ and $P$ are multisets of CHR constraints called the linear (CHR) store and the persistent (CHR) store, respectively. $B$ is a conjunction of built-in constraints and $V$ is a set of variables. The first state in a program execution is called initial state and can be any valid $\omega_{!}$ state.

Definition 3 defines the notion of local and strictly local variables, which is needed in definition 4 [1].

**Definition 3.** (Local and strictly local variables).

Let $σ = ⟨L, P, B, V⟩$ be an $\omega_{!}$ state. Then the variables occurring in $B$ or in $L$ or in $P$ but not in $V$ are called the local variables of $σ$. While the variables occurring in $B$ but not in $L$, $P$ and $V$ are called the strictly local variables of $σ$.

Definition 4 presents the definition of state equivalence and is based on the definition of state equivalence for $\omega_{va}$ given in [8] which has been extended by condition 5 to handle idempotence of persistent constraints [1].

**Definition 4 (State Equivalence).** Equivalence between $\omega_{!}$ states is the smallest equivalence relation $≡_{!}$ over $\omega_{!}$ states that satisfies the following conditions [1]:

1. (Equality as Substitution)

   Let $X$ be a variable, $t$ be a term and $\doteq$ the syntactical equality relation.

   $⟨L, P, X \doteq t \wedge B, V⟩ \equiv_{!} ⟨L[X/t], P[X/t], X \doteq t \wedge B, V⟩$

2. (Transformation of the Constraint Store)

   If $CT \models \exists \mathcal{F}.B \leftrightarrow \exists \mathcal{F}'.B'$ where $\mathcal{F}, \mathcal{F}'$ are the strictly local variables of $B, B'$ respectively, then:

   $⟨L, P, B, V⟩ \equiv_{!} ⟨L, P, B', V⟩$
3. (Omission of Non-Occurring Global Variables)
   If \( X \) is a variable that does not occur in \( \mathbb{L}, \mathbb{P} \) or \( \mathbb{B} \) then:
   \[
   \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \{X\} \cup \mathbb{V} \rangle \equiv_1 \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle
   \]

4. (Equivalence of Failed States)
   \[
   \langle \mathbb{L}, \mathbb{P}, \bot, \mathbb{V} \rangle \equiv_1 \langle \mathbb{L}', \mathbb{P}', \bot, \mathbb{V} \rangle
   \]

5. (Contraction of Persistent Constraints)
   \[
   \langle \mathbb{L}, \mathbb{P} \cup \mathbb{P} \cup \mathbb{B}, \mathbb{V} \rangle \equiv_1 \langle \mathbb{L}, \mathbb{P} \cup \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle
   \]

Based on the state equivalence definition a rewrite system is defined over equivalence classes of states with \([G] := \{G' \mid G \equiv_1 G''\}\). The fact that body constraints can be introduced as either persistent or linear constraints leads to two distinct transition rules. The post-transition state \( \tau \) needs to be different from the pre-transition state \( \sigma \). This means the transition relation is irreflexive. This definition is only valid for so called range restricted programs. Those are programs where no rule introduces free variables in the guard or body that are not also present in the head of the rule [1][2].

**Definition 5. (\(\omega\)-Transitions)**

For a range restricted CHR program \( \mathbb{P} \), the state transition system \((\Sigma_r/ \equiv_1, \mapsto_\tau)\) is defined as follows.

**ApplyLinear:**

\[
\begin{align*}
\mapsto_\tau^r &: (H_1^p \cup H_2^p) \setminus (H_1^p \cup H_2^p) \not\equiv G \mid B_c, B_b \quad H_2^p \neq \emptyset \quad [\sigma] \neq [\tau] \quad \\
\sigma &= [(H_1^l \cup H_2^l \cup \mathbb{L}, H_1^p \cup H_2^p \cup \mathbb{P}, G \land \mathbb{B}, \mathbb{V})] \\
\mapsto_\tau^r & [(H_1^l \cup B_c \cup \mathbb{L}, H_1^p \cup H_2^p \cup \mathbb{P}, G \land \mathbb{B} \land B_b, \mathbb{V})] = \tau
\end{align*}
\]

**ApplyPersistent:**

\[
\begin{align*}
\mapsto_\tau^r &: (H_1^p \cup H_2^p) \setminus H_2^p \not\equiv G \mid B_c, B_b \quad [\sigma] \neq [\tau] \quad \\
\sigma &= [(H_1^l \cup \mathbb{L}, H_1^p \cup H_2^p \cup \mathbb{P}, G \land \mathbb{B}, \mathbb{V})] \\
\mapsto_\tau^r & [(H_1^l \cup \mathbb{L}, H_1^p \cup H_2^p \cup B_c \cup \mathbb{P}, G \land \mathbb{B} \land B_b, \mathbb{V})] = \tau
\end{align*}
\]

where \(B_c\) are the CHR constraints and \(B_b\) are the built-in constraints of the body of a rule.

In cases where \(r\) is clear from the context or not important \(\mapsto_1\) is used instead of \(\mapsto_\tau^r\). With \(\mapsto_1^r\) the reflexive-transitive closure of \(\mapsto_1\) is denoted [2].

**Example 1. (Transitive Hull)[1]**

Consider the following CHR program for computing the transitive hull of a graph represented by edge constraints \(e/2\):

\[
\begin{align*}
t & @ e(X,Y), e(Y,Z) \Rightarrow e(X,Z)
\end{align*}
\]

Called with \(e(1,2), e(2,1)\) only four transitions are applied where \(e(1,1), e(1,2), e(2,2)\) and \(e(2,1)\) are added to the persistent store. No further transitions are possible since all resulting states would be equivalent.

For \(\omega\), this program terminates for all possible inputs [1].
3 Extended State Equivalence Definition for ω!

In definition 4 state equivalence for ω! is presented like it is introduced in [1]. In this definition the occurrence of linear constraints that are also present as persistent constraints has influence on state equivalence.

Example 2. Consider the following program:

\[
\begin{align*}
a &\leftrightarrow c. & a &\leftrightarrow d. \\
 b &\Rightarrow c. & b &\Rightarrow d.
\end{align*}
\]

In ωva this program does not terminate if called with \(a, b\) due to trivial non-termination. The constraint \(a\) can fire one of the two simplification rules which then lead to two different states. These states can fire propagation rules in a way that the resulting states are equivalent.

In ω! the resulting final states of an execution with the initial state \(\langle \{a, b\}, \emptyset, \top, \emptyset \rangle\) is \(\langle \{b, c\}, \{c, d\}, \top, \emptyset \rangle\) or \(\langle \{b, d\}, \{c, d\}, \top, \emptyset \rangle\). Those two states are not equivalent, even so no rule can be constructed where the head and guard can only be matched by only one of the two states.

Our work introduces definition 6 as extension to definition 4. It adds one condition, which is based on the idea that if any number of a constraint is present adding more does not make a difference. This captures the nature of persistent constraints more accurately than the original definition. This behavior is not represented in the state equivalence definition of ω! so far.

Definition 6. Definition 4 is extended by the following axiom:

(Contraction of Linear and Persistent Constraints)

\[6. \ \langle P \cup L, P \cup P, B, V \rangle \equiv! \langle L, P \cup P, B, V \rangle\]

Definition 6 gives an axiomatic definition for \(\equiv!\). It is difficult to show that something is not equal with an axiomatic definition. Definition 7 presents the ▽◁ relation [7] that is needed for Theorem 1 which gives a decidable criterion for \(\equiv!\). It is based on the criterion in [7] but takes definition 6 into account.

Definition 7 (▽◁). The relation ▽◁ over multisets of constraints is defined as

\[G ▽◁ G' \text{ if and only if } (\forall c \in G. \exists c' \in G'. c = c') \land (\forall c' \in G'. \exists c \in G. c = c')\]

Theorem 1 (Criterion for \(\equiv!\)). Let \(σ = \langle L, P, B, V \rangle, σ' = \langle L', P', B', V \rangle\) be ω! states with local variables \(\bar{y}, \bar{y}'\) that have been renamed apart. \(σ \equiv! σ'\) iff

\[
CT \models ▽(B) \Rightarrow ▽(\Delta L \cup P) \land (P ▽ P') \land B') \land ▽(B') \Rightarrow ▽(\Delta L' \cup P) \land (P ▽ P') \land B)
\]

where \(\Delta\) is the symmetric difference.

Proof. '⇐': Let \(σ\) and \(σ'\) be two ω! states with \(σ = \langle L, P, B, V \rangle, σ' = \langle L', P', B', V \rangle\) with local variables \(\bar{y}, \bar{y}'\) that have been renamed apart and
\[ CT \models \forall (B \mapsto \exists y'(\((((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \land B')) \land \\
\forall (B' \mapsto \exists y.(((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \land B) \] 

If \( CT \models \neg \exists ((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \), then \( CT \models B = B' = \bot \) so that definition 4 condition 4 proves \( \sigma \equiv \sigma' \).

If a matching for \( ((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \) does exist it follows from 
\[ \forall (B \mapsto \exists y'.(((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \land B') \land \text{by definition 4 condition 2} \]
that: 
\[ \sigma = (L, P, (((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \land B \land B', V) \]

Definition 4 condition 1 and 6 lead to:
\[ \sigma = (L \cap L', P, (((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \land B \land B', V) \] 

Definition 4 condition 1 and 5 lead to:
\[ \sigma = (L, P', (((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \land B \land B', V) \]

where \( P'' \) equals \( P' \) modulo multiplicities. Definition 4 condition 5 and definition 6 now lead to: 
\[ \sigma = (L', P', (((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \land B \land B', V) \]

From \( \forall (B' \mapsto \exists y.(((L \triangle L') \cup P) \bowtie P) \land (P \bowtie P') \land B) \) follows by definition 4 that: 
\[ \sigma = (L', P', B', V) = \sigma' \]

To prove the forward direction the compliance of the conditions 1 to 5 from definition 4 and the condition form definition 6 need to be shown. For condition 1 to 4 of definition 4, compliance is analogous to [7][p.39f] and for condition 5 of definition 4 compliance is analogous to [7][p.47] hence only Definition 6 is considered: Let \( \sigma = (P \bowtie L, P \bowtie P, B, V), \sigma' = (L, P \bowtie P, B, V) \in \Sigma \) with local variables \( \bar{y}, \bar{y}' \). As \( (((P \bowtie L) \triangle L) \cup P) \bowtie (P \bowtie P) = ((P \bowtie P) \bowtie (P \bowtie P)) \), the following is a tautology:

\[ CT \models \forall (B \mapsto \exists y',(((P \bowtie L) \triangle L) \cup P) \bowtie (P \bowtie P) \land ((P \bowtie P) \bowtie (P \bowtie P) \land B) \land \\
\forall (B \mapsto \exists y.(((P \bowtie L) \triangle L) \cup P) \bowtie (P \bowtie P) \land ((P \bowtie P) \bowtie (P \bowtie P) \land B)) \]

Definition 8 introduces the merge operator \( \circ \) for merging \( \omega_1 \) states [7]. It is a technical definition needed for Lemma 3.

**Definition 8 (Merge Operator \( \circ \)).** Let \( \sigma_1 = (L_1, P_1, B_1, V_1) \) and \( \sigma_2 = (L_2, P_2, B_2, V_2) \) such that local variables of one state are disjunct from all variables in the other state. Then for a set \( V \) of variables 
\[ \sigma_1 \circ_V \sigma_2 := (L_1 \cup L_2, P_1 \cup P_2, B_1 \land B_2, (V_1 \cup V_2) \backslash V) \]

This definition is further lifted to equivalence classes. In that case, the merge operation assumes that two representants with accordingly disjunct variables are selected: 
\[ [\sigma_1] \circ_V [\sigma_2] := [\sigma_1 \circ_V \sigma_2] \]

For \( V = \emptyset \), \( \sigma_1 \circ \sigma_2 \) and \( [\sigma_1] \circ [\sigma_2] \) is written, respectively.

Lemma 1 states that equivalence is maintained by the merge operator. The proof is similar to [7][p.50f].
Lemma 1 (\(\Diamond V\) maintains Equivalence). Let \(\sigma_1 \equiv! \sigma_2\), then \((\sigma_1 \Diamond V \tau) \equiv! (\sigma_2 \Diamond V \tau)\) for all \(V\).

Proof. W.l.o.g. let \(\sigma_i = \langle L_i, P_i, B_i, \mathcal{V}' \rangle\) for \(i = 1, 2\) and let \(\tau = \langle L, P, B, \mathcal{V}'' \rangle\) such that the variables are disjunct according to Definition 8. Let \(\bar{y}_1, \bar{y}_2\) be the local variables of \(\sigma_1\) and \(\sigma_2\) respectively. According to Theorem 1:

\[
\begin{align*}
CT &\models \forall (B_1 \rightarrow \exists \bar{y}_2 . (((L_1 \triangle L_2) \uplus P_1) \bowtie P_2) \land (P_1 \bowtie P_2) \land B_2) \\
\forall (B_2 \rightarrow \exists \bar{y}_1 . (((L_1 \triangle L_2) \uplus P_1) \bowtie P_2) \land (P_1 \bowtie P_2) \land B_1))
\end{align*}
\]

Let \(\bar{x} = (V' \cap V)\), then

\[
\begin{align*}
CT &\models \forall (B_1 \rightarrow \exists \bar{y}_2 \exists \bar{x}. (((L_1 \triangle L_2) \uplus P_1) \bowtie P_2) \land (P_1 \bowtie P_2) \land B_2) \\
\forall (B_2 \rightarrow \exists \bar{y}_1 \exists \bar{x}. (((L_1 \triangle L_2) \uplus P_1) \bowtie P_2) \land (P_1 \bowtie P_2) \land B_1))
\end{align*}
\]

As \((L \triangle L)\) is always empty and \((P \bowtie P)\) is a tautology, \(((L_1 \triangle L_2) \uplus P_1) \bowtie P_2)\) can be extended to \(((L_1 \uplus L_2) \triangle (L_2 \cap L)) \uplus (P_1 \uplus P) \bowtie (P_1 \uplus P)\) and \((P_1 \bowtie P_2)\) to \((P_1 \bowtie P) \bowtie (P_2 \bowtie P)\). Similarly, \(B \rightarrow B\) is a tautology, and therefore we have for \(\bar{z}\) being the local variables of \(\tau\) combined with \(V'' \setminus V\):

\[
\begin{align*}
CT &\models \forall (B_1 \land B \rightarrow \exists \bar{y}_2 \exists \bar{z} . (((L_1 \uplus L) \triangle (L_2 \cap L)) \uplus (P_1 \uplus P) \bowtie (P_1 \uplus P) \bowtie (P_2 \uplus P) \land B_2) \\
\forall (B_2 \rightarrow \exists \bar{y}_1 \exists \bar{z} . (((L_1 \uplus L) \triangle (L_2 \cap L)) \uplus (P_1 \uplus P) \bowtie (P_1 \uplus P) \bowtie (P_2 \uplus P) \land B_1))
\end{align*}
\]

As the local variables of \(\sigma_1 \Diamond V \tau\) are \(\bar{x} \cup \bar{y}_1 \cup \bar{z}\), and analogously for \(\sigma_2 \Diamond V \tau\), it can be concluded by Theorem 1

\[
\begin{align*}
\sigma_1 \Diamond V \tau &= \langle L_1 \uplus L, P_1 \uplus P, B_1 \land B, (V' \cup V'') \setminus V \rangle \equiv! \\
\langle L_2 \uplus L, P_2 \uplus P, B_2 \land B, (V' \cup V'') \setminus V \rangle &= \sigma_2 \Diamond V \tau
\end{align*}
\]

\(\square\)

4 Confluence Test

This section describes a confluence test for \(\omega_1\). The confluence test works in a similar way as it is described in [3] for \(\omega va\). It is shown that the test works and how the persistent constraint store influences it. This is not trivial, since \(\omega_1\) breaks with monotonicity which is used in the proof for \(\omega va\).

Definition 9 defines joinability of two states. This is needed for the definition of confluence itself [3, p. 102]. Definition 10 and 11 are general CHR definitions and not explicitly only for \(\omega_1\).

**Definition 9 (Joinability).** Two states \(\sigma_1\) and \(\sigma_2\) are joinable if there exists a state \(\sigma'\) such that \(\sigma_1 \rightarrow^* \sigma'\) and \(\sigma_2 \rightarrow^* \sigma'\).

Definition 10 defines confluence formally, but is not useful for actual confluence tests since in general there exists an infinite number of states [3, p. 102].
Definition 10 (Confluence). A CHR program is confluent if for all states \( S, S_1, S_2 \) if \( S \rightarrow^* S_1, S \rightarrow^* S_2 \) then \( S_1 \) and \( S_2 \) are joinable.

Definition 11 (Local Confluence). A CHR program is locally confluent if for all states \( S, S_1, S_2 \) if \( S \rightarrow^* S_1, S \rightarrow^* S_2 \) then \( S_1 \) and \( S_2 \) are joinable.

Definition 11 defines local confluence which is later used in the confluence test for terminating programs [3, p. 104].

Lemma 2 (Newman’s Lemma). A terminating reduction system is confluent if it is locally confluent.

Since the transition rules of \( \omega_1 \) are only applied if the resulting state is not equivalent to the original state \( \omega_1 \) does not have the monotonicity property that is needed in the confluence criterion for \( \omega_{eq} \). Lemma 3 is a weaker property that can be sufficient in many cases where proofs for other semantics use the monotonicity property. This lemma is needed for the proofs further on.

Lemma 3. If \( [\sigma] \rightarrow^*_1 [\tau] \) then \( [\sigma] \circ [\sigma'] \rightarrow^*_1 [\tau] \circ [\sigma'] \)

Proof. by induction:
In the following \( r_{seq} \) is a sequence of rules.
Basis: Let \( r_{seq} \) consist of 0 rules.
\[ [\sigma] \rightarrow^*_{r_{seq}} [\tau] \]
\[ [\sigma] \circ [\sigma'] \rightarrow^*_1 [\tau] \circ [\sigma'] \] is correct since \( \sigma \) and \( \tau \) are equivalent.

Induction hypothesis: if \( r_{seq} \) consists of \( n \) rules and \( [\sigma] \rightarrow^*_{r_{seq}} [\tau] \) then \( [\sigma] \circ [\sigma'] \rightarrow^*_1 [\tau] \circ [\sigma'] \) is true.

Inductive step: Let \( r_{seq} \) consist of \( n + 1 \) rules, \( r'_{seq} \) be the sequence of the first \( n \) rules of \( r_{seq} \) and \( r_{n+1} \) be the last rule of \( r_{seq} \). This means according to the induction hypothesis
\[ [\sigma] \rightarrow^*_{r_{seq}} [\tau], [\sigma] \circ [\sigma'] \rightarrow^*_1 [\tau] \circ [\sigma'] \] and \( [\tau] \rightarrow^*_{r_{n+1}} [\tau'] \)
Now \( [\tau] \circ [\sigma'] \rightarrow^*_1 [\tau'] \circ [\sigma'] \) needs to be shown.
If \( r_{n+1} \) is applicable to \( [\tau] \circ [\sigma'] \) then means that \( [\tau] \circ [\sigma'] \) is correct since the head and guard of \( r_{n+1} \) are satisfied by \( [\tau] \) and so the only way to prevent rule application in \( \omega_1 \) is if the resulting state is equivalent.

With lemma 2 there is still an infinite number of states to be tested in general. Definition 12 gives the definition for critical pairs which is used to reduce the number of states that need to be tested to a finite number of states [3, p. 103][7].
Definition 12 (Critical Ancestor State, Critical Pair). For any two (not necessarily different) rules of a CHR program with renamed apart variables that are of the form

\[ r_1 \not\in H_1 \setminus H_2 \Leftrightarrow G | B_c, B_b \]
\[ r_2 \not\in H'_1 \setminus H'_2 \Leftrightarrow G' | B'_c, B'_b \]

let \( O_1 \subseteq H_1, O_2 \subseteq H_2, O'_1 \subseteq H'_1, O'_2 \subseteq H'_2 \) such that for \( B := ((O_1 \cup O_2) = (O'_1 \cup O'_2)) \land G \land G' \) it holds that \( CT \models \exists \text{Band}(O_2 \cup O'_2) \neq \emptyset \), then all states of the form

\[ \sigma = (L; P; B; \forall) \]

where \( L \cup P = K \cup K' \cup R \cup R' \cup O_1 \cup O_2, \forall \) is the set of all variables occurring in heads and guards of both rules and \( K := H_1 \setminus O_1, K' := H'_1 \setminus O'_1, R := H_2 \setminus O_2, R' := H'_2 \setminus O'_2 \) are called critical ancestor states. The rules \( r_1 \) and \( r_2 \) are called overlapping rules. The pair of states \( (\sigma_1, \sigma_2) \) with

\[ \sigma_1 := \begin{cases} (((K \cup K' \cup R \cup R' \cup O_1) \setminus P); P \cup B_c; B \land B_b; \forall) & \text{if } H_2 \subseteq P \\ (((K \cup K' \cup R \cup R' \cup O_1) \setminus P); P \cup B_c; B \land B_b; \forall) & \text{else} \end{cases} \]
\[ \sigma_2 := \begin{cases} (((K \cup K' \cup R \cup R' \cup O_1) \setminus P); P \cup B'_c; B \land B'_b; \forall) & \text{if } H'_2 \subseteq P \\ (((K \cup K' \cup R \cup R' \cup O_1) \setminus P); P \cup B'_c; B \land B'_b; \forall) & \text{else} \end{cases} \]

is called a critical pair of the critical ancestor state \( \sigma \).

Since \( \omega_1 \) states have two constraint stores each overlap leads to up to \( 2^2 \) critical ancestor states.

With the definition of critical pairs the actual confluence test is presented in theorem 2.

Theorem 2. A terminating \( \omega_1 \) program is confluent iff all its critical pairs are joinable.

Proof. Because of Newman’s Lemma 2 it is sufficient to prove local confluence.

To show the if direction, let \( \sigma \) be an \( \omega_1 \) state where at least two transitions are possible.

\[ \sigma \rightarrow_1 \sigma_1 \text{ and } S \rightarrow_1 \sigma_2 \]

If the two rules apply to different parts of the state \( \sigma \) then \( \sigma_1 \) and \( \sigma_2 \) must be joinable due to lemma 3.

Else the two rules overlap. There must exist critical pair \( (\sigma'_1, \sigma'_2) \) with the same overlap and a state \( \sigma_{\text{rest}} \) so that \( \sigma'_1 \circ \sigma_{\text{rest}} \equiv_! \sigma_1 \) and \( \sigma'_2 \circ \sigma_{\text{rest}} \equiv_! \sigma_2 \), as a consequence of lemma 3 this means that if \( (\sigma'_1, \sigma'_2) \) is joinable then \( \sigma_1 \) and \( \sigma_2 \) are also joinable.

The only if direction can be shown by contradiction. Let \( P \) be a program that is locally confluent in \( \omega_1 \) and has a critical pair that is not joinable. The critical ancestor state of this critical pair can be constructed as initial state. So there exists a state that leads to the nonjoinable critical pair, but since \( P \) is locally confluent, the states must be joinable. This results in a contradiction. \( \Box \)
Theorem 2 gives a decidable test for confluence in \( \omega! \). The number of test cases is exponentially increased in comparison to the confluence test for \( \omega_{va} \).

Example 3. Consider the following program for transitive closure [3][p.190]:

\[
\begin{align*}
    dp @ p(X,Y) \setminus p(X,Y) &\leftrightarrow true. \\
p1 @ e(X,Y) &\Rightarrow p(X,Y). \\
p2 @ p(X,Y), p(Y,Z) &\Rightarrow p(X,Z).
\end{align*}
\]

This program is written for semantics that rely on a deterministic order in which rules are executed. The rule \( dp \) removes duplicates of \( p/2 \) constraints. This needs to happen before \( p2 \) is executed otherwise this program would not be guaranteed to terminate. In \( \omega! \) however it terminates for all possible inputs.

This program has several critical pairs that result from the overlap of \( dp \) and \( p2 \). Here are two of the 12 critical ancestor states:

\[
\begin{align*}
    \sigma_1 &= \langle \{p(X,X), p(X,X)\}, \emptyset, \emptyset, \{X\} \rangle \\
    \sigma_2 &= \langle \{p(X,X), p(X,X), p(X,X)\}, \emptyset, \emptyset, \{X\} \rangle
\end{align*}
\]

Each critical ancestor state leads to two equivalent critical pairs, so only two critical pairs need to be tested. For better readability states are represented by their logical reading and persistent constraints are marked with the index \( p \).

Critical pair resulting from \( \langle \{p(X,X), p(X,X)\}, \emptyset, \emptyset, \{X\} \rangle \):

\[
\sigma_1 \rightarrow^{dp} \tau_1 p(X,X) \rightarrow^{p2} \tau_2 p(X,X), p_p(X,X)
\]

Critical pair resulting from \( \langle \{p(X,X), p(X,X), p(X,X)\}, \emptyset, \emptyset, \{X\} \rangle \):

\[
\begin{align*}
    &\sigma_2 \rightarrow^{dp} \tau_1 p(X,X), p(X,X) \rightarrow^{p2} \tau_2 p(X,X), p_p(X,X) \rightarrow^{dp} \tau_1 p(X,X), p_p(X,X) \\
    &\rightarrow^{dp} \tau_1 p(X,X), p_p(X,X)
\end{align*}
\]

Both critical pairs are joinable. The rest of the critical ancestor states are analog and also lead to joinable critical pairs, so the program is confluent for \( \omega! \). It can be noted that if the input consists only of \( e/2 \) constraints, then the \( dp \) rule is unnecessary since all \( p/2 \) constraints are added to the persistent store and the semantics of \( \omega! \) would already prevent duplicates.

5 Confluence Checker

The theoretical results of the previous section are used to create a confluence checker for \( \omega! \). It can test syntactical correct programs for confluence. Those programs are not allowed to contain rules that recreate removed head constraints, so called pathological rules. It only supports the Prolog unification = /2 and
true as built-ins. It has the option to also support the built-ins $=</2$, $>/2$, $</2$, and $==/2$. These are implemented as constraint solvers with ask and entailed constraints to replace the built-ins in the guards and can be extended for further built-ins, as it is presented in [9]. For the joinability test it runs a source to source transformation that implements $\omega_1$ and checks if the final states are equivalent.

The application is based on the confluence checker for the abstract semantics that has been written in 2010 by Johannes Langbein [6].

5.1 Modifying the Confluence Checker

The idea behind the modification for $\omega_1$ is to use the source to source transformation of [2] to create a transformed program. The overlaps that are found in the untransformed program are then used to create critical pairs that can be tested with the transformed program in the joinability test.

The confluence checker starts by parsing all rules to search for overlaps. Since during the parsing process all of the necessary information for the source to source transformation is present, it is also used to create the transformed program according to Definition 13. This transformation is based on the implementation of $\omega_1$ as source to source transformation that is presented in [2]. It creates a transformed program that behaves like the original program would behave in $\omega_1$. This transformed program is executed in the so-called refined semantics [3] that is typically used in implementations. In this semantics the order of the rules has influence on the priority of their execution. To differentiate between linear and persistent constraints, each constraint gets an additional argument. This is either $l$ for linear constraints, $p$ for persistent constraints, $t$ for potentially added linear constraints or $c$ for potentially added persistent constraints.

Definition 13 (Source to Source Implementation). For every $n$-ary constraint $c/n$ in $P$ there exists a constraint $c/(n + 1)$ in $[P]$. In the following, for a multiset of user-defined $\omega_1$ constraints $M = \{c_1(\hat{t}_1),...c_n(\hat{t}_n)\}$ let

1. For every rule $r \preceq H_1 \setminus H_2 \iff G \mid B_c, B_b$ in $P$, and all multisets $H_1^1, H_1^P, H_2^1, H_2^P$ s.t. $H_1^1 \cup H_1^P = H_1$ and $H_2^1 \cup H_2^P = H_2$ and $H_2^1 \neq \emptyset$, the following rule is added at the end of $[P]$:

   \[
   l(H_1^1) \cup p(H_1^P) \cup p(H_2^P) \setminus t(H_2^1) \iff G \mid t(B_c), B_b
   \]

2. For every rule $r \preceq H_1 \setminus H_2 \iff G \mid B_c, B_b$ in $P$, and all multisets $H_1^1, H_1^P, H_2^1, H_2^P$ s.t. $H_1^1 \cup H_1^P = H_1$ the following rule is added at the end of $[P]$:

   \[
   l(H_1^1) \cup p(H_1^P) \cup p(H_2) \Rightarrow G \mid c(B_c), B_b
   \]
3. For every rule \( \{c(p, \overline{t}), c(p, \overline{t}')\} \uplus H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b \) in \([P]\) with fresh variables where \( \overline{t} \) and \( \overline{t}' \) are unifiable, add also the following rule at the end of \([P]\) with \( \overline{t} = \overline{t}' \):

\[\{c(p, \overline{t})\} \uplus H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b\]

4. For every user-defined constraint \( c/n \) in \( P \), add the following rules at the top of \([P]\), where \( \overline{t} \) is a sequence of \( n \) different variables:

- \( c(p, \overline{t}) \Leftrightarrow c(c, \overline{t}) \Leftrightarrow T \)
- \( c(c, \overline{t}) \Leftrightarrow c(p, \overline{t}) \)
- \( c(p, \overline{t}) \Leftrightarrow c(p, \overline{t}) \Leftrightarrow T \)
- \( c(p, \overline{t}) \Leftrightarrow c(p, \overline{t}) \Leftrightarrow c(l, \overline{t}) \)

Unlike in [2] this transformation is not designed for the priority based semantics. Instead it uses the refined semantics [3] that is used for the CHR implementation in SWI Prolog. In this semantics propagation rules can only fire once with the same constellation of constraints and the priority for rule execution is from top to bottom. Additionally rule 3 now performs the equivalence check of arguments by matching the arguments of the head constraints. The original definition uses an equivalence check in the guard, which would not work in the intended way for free variables in SWI Prolog. Rule 4 is extended for the new equivalence definition and adds a cleanup rule for persistent constraints. This ensures that the final state has the smallest number of constraints possible for easy state equivalence checks.

The transformed program can be executed in SWI Prolog and behaves like the untransformed program would be behave in \( \omega_1 \) [2]. To differentiate between linear and persistent constraints, each constraint has an additional argument. So only one goal store for CHR constraints is needed in a state. An initial query and its final state of the execution in the transformed program can be represented as terms with the form \( \text{state}(G, B, V) \). This is the representation needed for the state equivalence checker. The states are always reduced to the equivalent states with the smallest number of constraints according to condition 5 of Definition 4 and Definition 6. This allows the equivalence checker to be used for final states. The parser can be used on its own as an implementation of \( \omega_1 \) as source to source transformation.

The confluence check is modified so that for every found overlap all possible critical pairs are created according to definition 12. Each constraint of a critical pair has the additional argument to determine if the constraint are added as linear or persistent.

Example 4. Calling the confluence checker with the following program.

```prolog
b <=> true.
a <=> true.
a ==> b.
```
This program is confluent in the abstract semantics, but not in $\omega$ due to the non-
removability of persistent constraints. The checker finds a non-joinable critical 
pair and gives the following output:

```
================================================================================

The following critical pair is not joinable:
state([], [true], []),
state([a(l), b(c)], [], []),

This critical pair stems from the critical ancestor state:
[a]
with the overlapping part:
[(a, a)]
of the following two rules:
a<>true
a==>b

================================================================================

'The CHR program in 'E:\examples\example-1.pl' is NOT confluent!
'1' non-joinable critical pair(s) found!'
```

5.2 Limitations

The source to source transformation that is used to simulate the behavior of $\omega$
cannot be used with programs that contain pathological rules. Programs that have pathological rules need to be rewritten by splitting the rule in several rules with appropriate guards, where constraints are added to the kept head instead of removing and adding them. Since this also removes potential trivial non-
termination in $\omega$, the resulting program is not operationally equivalent to the original program, but usually has the intended behavior.

The source to source transformation creates many rules for a single rule in the original program. The worst case of the blowup of rules for a single rule lies in $\mathcal{O}(2^n n!)$ where $n$ is the number of head constraints of the rule, while the best case still lies in $\mathcal{O}(2^n)$. There can be redundant rules or propagation rules with true as body. The later kind of rule causes the confluence checker to write compiler warnings to the console during the joinability tests. These problems could be fixed by a real implementation of $\omega$. If such an implementation is made, the source to source transformation can be removed from the confluence checker and it can be modified to use the real implementation for the confluence tests without changing the overall operation of the confluence checker.

The confluence checker only supports a limited number of built-ins. While more built-ins can be added by constraint solvers those can still be limited, especially when trying to add something like mathematical operations e.g. addition.

Since the joinability test only checks final states for equivalence some joinable critical pairs may be presented as non-joinable. If this happens the non-
joinability that leads to different final states is implied by another critical pair that is not joinable.
6 Related Work

In [1] the idea of persistent constraints is introduced, $\omega_1$ is defined and its termination behavior is analyzed. The equivalence definition in this work misses the idea that linear constraints can be implied by persistent constraints in the context of state equivalence. The implementation of $\omega_1$ as source to source transformation is presented in [2]. It has an insufficient definition for pathological rules and had to be adjusted for the new equivalence definition. In [7] a decidable criterion for equivalence in $\omega_1$ is introduced. This is for the original state equivalence definition and had to be adjusted for the new definition. Additionally [7] presents a lemma for monotonicity in $\omega_1$ which is incorrect. Our work introduced a weaker property to replace the incorrect monotonicity lemma. So far there were no analysis of confluence behavior and confluence tests for $\omega_1$.

In [6] the confluence checker for the abstract semantics is presented. The test for confluence that it uses is described by [3]. This is used as foundation for the confluence checker for $\omega_1$ and extended by built-ins that are not supported in the original implementation.

7 Conclusion and Future Work

A confluence test for the operational semantics $\omega_1$ for CHR with persistent constraints has been presented (c.f. theorem 2). For this purpose, the state equivalence definition of $\omega_1$ states has been adapted such that it better reflects the intuitive meaning of persistent constraints. Other definitions and results for the original definition have been shown to be compatible with the improved state equivalence definition (c.f. section 3).

The standard confluence test of CHR has been adapted to match $\omega_1$. As it heavily relies on monotonicity which is broken by $\omega_1$, a weaker property has been established to allow for the confluence test (c.f. lemma 3). The number of critical pairs increases exponentially compared to $\omega_{va}$ (c.f. definition 12).

The confluence test has been implemented based on the confluence checker for the abstract semantics of CHR [6] (c.f. section 5). Due to the close relation of the proposed confluence criterion to the criterion for $\omega_{va}$, the confluence checker for $\omega_1$ has less limitations than the confluence checker for the abstract semantics on which it is based. The confluence checker integrates support for built-in constraints other than $true$, $false$ and syntactic equality $=$. The general interface of this extension allows for integration of even more built-in constraints.

For the future it can be investigated if confluence in $\omega_1$ also implies the linear logical reading of programs and easy parallelization. Since most existing CHR programs are written with the abstract semantics with token store in mind, many programs have to be modified to work properly in $\omega_1$. Hence, it would be interesting to investigate programs that have been originally been written for $\omega_1$ and their behavior regarding parallelization. Furthermore, a direct implementation of $\omega_1$ that replaces the source to source transformation in the confluence checker would be useful as it would improve efficiency of the execution of the programs.
and the confluence checker. The results presented in this paper are independent of the actual implementation of $\omega!$ and could therefore be used without deviation together with such a direct implementation.

Further it would be interesting to investigate so-called completion methods for $\omega!$. Those are methods that add rules to a non confluent program in order to make it confluent, like it is investigated in [4] for CHR without persistent constraints.

To improve the efficiency of the confluence test for $\omega!$ it should be investigated if the number of critical pairs can be reduced.

Acknowledgments

We want to thank the reviewers for their detailed feedback which was a great help for improving this work.

References

1. BETZ, Hariolf ; RAISER, Frank ; FRÜHWIRTH, Thom: Persistent constraints in constraint handling rules. In: WLP 9 (2010), S. 155–166
2. BETZ, Hariolf ; RAISER, Frank ; FRÜHWIRTH, Thom: A complete and terminating execution model for constraint handling rules. In: Theory and Practice of Logic Programming 10 (2010), Nr. 4-6, S. 597–610
7. RAISER, Frank: Graph transformation systems in Constraint Handling Rules: improved methods for program analysis, Universität Ulm, Diss., 2010 http://dx.doi.org/10.18725/OPARU-1742
An Operational Semantics for Constraint-logic Imperative Programming

Jan C. Dageförde and Herbert Kuchen

ERCIS, University of Münster, Germany
{dagefoerde,kuchen}@uni-muenster.de

Abstract. Object-oriented (OO) languages such as Java are the dominating programming languages nowadays, among other reasons due to their ability to encapsulate data and operations working on them, as well as due to their support of inheritance. However, in contrast to constraint-logic languages, they are not particularly suited for solving search problems. During development of enterprise software, which occasionally requires some search, one option is to produce components in different languages and let them communicate. However, this can be clumsy.

As a remedy, we have developed the constraint-logic OO language Muli, which augments Java with logic variables and encapsulated search. Its implementation is based on a symbolic Java virtual machine that supports constraint solving and backtracking. In the present paper, we focus on the non-deterministic operational semantics of an imperative core language.

Keywords: Java, operational semantics, encapsulated search, programming paradigm integration.

1 Introduction

Contemporary software development is dominated by object-oriented (OO) programming. Its programming style benefits most industry applications by providing e.g. inheritance and encapsulation of structure and behaviour, since these concepts can positively contribute towards reusability and maintainability [13]. Nevertheless, some industry applications require search, for which constraint-logic programming is more suited than OO (or imperative) programming. However, developing applications that integrate both worlds, e.g. a Java application using a Prolog search component via Java Native Interface (JNI), is tedious and error-prone [10].

For that reason, we propose the Münster Logic-Imperative Programming Language (Muli), integrating constraint-logic programming with OO programming in a novel way. Based on Java, it adds logic variables and encapsulated search to the language, supported by constraint solvers and non-deterministic execution on a symbolic Java virtual machine (JVM). The symbolic JVM adapts concepts from the Warren Abstract Machine, such as choice points and trail [21]. Muli’s tight integration of both paradigms facilitates development of applications whose
business logic is implemented in Java, but which also require occasional search, such as operations research applications\cite{8}.

In this paper, we describe a reduction semantics for a core subset of Muli. In particular, the interaction of imperative statements, free variables, and non-determinism is of interest. For simplicity, we ignore inheritance, multi-threading, and reflection in this core language. The formulated semantics is helpful to get an understanding of the mechanics behind concepts that are novel to imperative and OO programming, and serves as a formal basis for implementing the symbolic JVM. It can also be used for reasoning about applications developed in Muli.

To that end, our paper is structured as follows. We provide an overview of the new language and its concepts in Section 2. Section 3 formalises the operational semantics of the core language. An example evaluation using this semantics is shown in Section 7. Section 5 presents a discussion of our concepts. Related work is outlined in Section 6. We then conclude in Section 7 and provide an outlook towards further research.

2 Language Concepts

The Muli language is derived from Java 8. We do not change existing concepts and features of Java, so that Muli also benefits from Java’s well-known and well-received features, such as OO and managed memory. Instead, the language is defined by its additions to Java, i.e. Muli is a superset of Java.

Muli adds the concept of free variables, i.e. variables that are declared and instantiated, but not to a particular value. Instead, they are treated symbolically and can be used in statements and expressions. Constraints on symbolic variables and expressions are imposed during symbolic execution of conditional statements. For example, an if statement with a condition that involves insufficiently constrained variables results in multiple branches that can be evaluated. Conceptually, we can non-deterministically choose a branch and evaluate it. Our implementation considers all these branches using backtracking and a (complete!) iterative deepening depth-first search strategy. This is supported by a specialised symbolic JVM that records choice points for each non-deterministic branch.

Furthermore, we enforce that non-determinism only takes place inside encapsulated search regions, whereas code outside encapsulation is executed deterministically. This ensures that non-determinism is not introduced by accident, intending not to harm the understanding of known Java concepts. Furthermore, this ensures that the overall application exits in a single state. In contrast, unencapsulated symbolic execution could result in multiple exit states, which could cause difficulties on the side of the caller. Encapsulation is expressed by using either of the \texttt{getAllSolutions} and \texttt{getOneSolution} operators. The logic of encapsulated search is described by search regions that are implemented as methods, e.g. as lambda abstractions, in order to defer their evaluation until encapsulation begins.

Solutions of encapsulated search are defined by values or expression returned from search regions. Due to non-determinism, multiple solutions can be returned
from search. Additionally, we introduce the special operation \texttt{fail;}\, whose evaluation results in immediate backtracking in the symbolic JVM without recording a solution for that branch.

From a syntactic perspective, these concepts extend Java only minimally. The resulting syntax of Muli can best be demonstrated using an example. Listing 1 exhibits a Muli method \texttt{log()} that searches for the logarithm of a number \(x\) to the base 2 using a free variable \(y\) and a method \texttt{pow} that calculates \(b^y\) imperatively, which is constrained to be equal to \(x\).

```java
int log(int x) {
    int y free;
    if (pow(2, y) == x) return y;
    else fail;
}
int pow(int b, int y) {
    int i; int r; i = 0; r = 1;
    while (i < y) {
        r = r * b; i = i + 1;
    }
    return r;
}
```

\textbf{Listing 1.} Non-deterministic computation of the logarithm of a number to the base 2 using (core) Muli.

Let us assume that the considered search region consists of a call to \texttt{log}, e.g. \texttt{log(4)}. When calling \texttt{log} with a given \(x\), the free variable \(y\) is created and then passed to \texttt{pow} that calculates the power \(b^y\) symbolically, as \(y\) is free. Therefore, its result \(x\) is actually a symbolic expression, accompanied by a set of accumulated constraints. Consequently, \texttt{log} computes the logarithm by defining a constraint system using another imperative method.

If the variables involved in a branching condition (of \texttt{if} or \texttt{while} in Listing 1) are not sufficiently constrained, one of the feasible branches is chosen non-deterministically. Actually, our symbolic JVM would try them one after the other based on a backtracking mechanism. When selecting a branch, the corresponding condition is added to the constraint store and consistency is checked. For example, as \(y\) is free, \texttt{while} \((i < y)\) can be either true or false. Therefore, one branch assumes the condition to be true and therefore adds the constraint \(i < y\) to the constraint store by imposing a conjunction of the existing store and the new constraint. In contrast, the second branch assumes it to be false and therefore adds the negated condition as a constraint. If an added constraint renders the store inconsistent, backtracking occurs, i.e. that branch is pruned and execution continues with a subsequent branch. Similarly, backtracking occurs when a solution is found so that the next branch can be evaluated to find further solutions. Muli’s encapsulated search operators use lazy streams to return collected solutions to the surrounding deterministic computation, such that the surrounding computation can decide how many solutions it wants to obtain.
3 A Non-deterministic Operational Semantics of Muli

Muli is an extension to Java and therefore intends to fully support all Java functionality. In fact, all Muli programs even compile to regular JVM bytecode that can be parsed and executed by a regular JVM (but incorrectly), and all Java programs can be executed correctly by Muli’s symbolic JVM.

For the purpose of describing an operational semantics of Muli, we focus on an imperative, procedural subset of Java (and Muli). This concise subset allows us to focus on the interaction between imperative and constraint-logic programming. It therefore abstracts from some features that are expected from Java that would not contribute to the discussion in the present paper, e.g., inheritance. Nevertheless, Muli’s symbolic JVM supports these features exactly according to the JVM specification [12] (but does not add interesting details w.r.t. non-determinism).

Let us first describe the syntax of our core language. We will use variables taken from a finite set
\[ Var = \{x_1, \ldots, x_m\} \]
for simplicity all of type integer (\(m \in \mathbb{N}\)).

Also let
\[ Op = AOp \cup BOp \cup ROp = \{+, -, *, /\} \cup \{&\&\}, ||\} \cup \{==, !=, <, >\} \]
be a finite set of arithmetic, boolean, and relational operation symbols, respectively. We focus on binary operation symbols. Furthermore, \(M\) is a finite set of methods.

The syntax of arithmetic expressions and boolean expressions as well as statements can be described by the following grammar. \(AExpr, BExpr\), and \(Stat\) denote the sets of all arithmetic expressions, boolean expressions, and statements, respectively, which can be constructed by the rules of this grammar.

\[
e ::= c \mid x \mid e_1 \oplus e_2 \mid m(e_1, \ldots, e_k)
\]
where \(c \in \mathbb{Z}\), \(x \in Var\), \(e_1, \ldots, e_k \in AExpr\), \(\oplus \in AOp\), \(m \in M\), \(k \in \mathbb{N}\),

\[
b ::= e_1 \otimes e_2 \mid b_1 \otimes b_2 \mid true \mid false
\]
where \(e_1, e_2 \in AExpr\), \(b_1, b_2 \in BExpr\), \(\otimes \in ROp\), \(\otimes \in BOp\),

\[
s ::= ; \mid int x; \mid int x free; \mid x = e; \mid e; \mid \{s\} \mid s_1 s_2 \mid
\]
\(\text{if } (b)s_1 \text{ else } s_2; \mid \text{while } (b)s \mid \text{return } e; \mid \text{fail;}
\)
where \(x \in Var\), \(e \in AExpr\), \(b \in BExpr\), \(s, s_1, s_2 \in Stat\).

Note, in particular, the possibility to create free logic variables by \(\text{int } x \text{ free} ;\).

After describing the syntax of the core language, let us now define its semantics. In the sequel, let \(A = \{a_0, \ldots, a_n\}\) be a finite set of memory addresses \((n \in \mathbb{N})\). Moreover, let

\[
Tree(A, Z) = A \cup Z \cup \{\oplus(t_1, t_2) \mid t_1, t_2 \in Tree(A, Z), \oplus \in Op\}
\]
be the set of all symbolic expression trees with addresses and integer constants as leaves and operation symbols as internal nodes.

We provide a reduction semantics, where the computations depend on an environment, a state, and a constraint store. Let \(Env = (Var \rightarrow A) \cup (M \rightarrow Var^* \times Stat)\) be the set of all environments, mapping each variable to an address.

\[^1\] In fact they are functions, since we ignore object-orientation in this presentation.
and each function to a representation \(((x_1, \ldots, x_k), s)\) that describes its parameters and code. We consider functions to be in global scope and define a special initial environment \(\rho_0 \in \text{Env}\) that maps functions to their respective parameters and code. Moreover, let \(\Sigma = \mathcal{A} \rightarrow \{\bot\} \cup \text{Tree}(\mathcal{A}, \mathcal{Z})\) be the set of all possible memory states. In \(\sigma \in \Sigma\), a special address \(\alpha_0\) with \(\sigma(\alpha_0) = \bot\) is reserved for holding return values of method invocations. Furthermore, \(CS = \{\text{true}\} \cup \text{Tree}(\mathcal{A}, \mathcal{Z})\) is the set of all possible constraint store states. Since constraints are specific boolean expressions, only conjunctions and relational operation symbols such as \(==\) and \(>\) will appear at the root of such a tree.

In the sequel, \(\rho \in \text{Env}, \sigma \in \Sigma, \gamma \in CS\); if needed, we will also add indices. We will use the notation \(a[x/d]\) when modifying a state or environment \(a\), meaning

\[
a[x/d](b) = \begin{cases} 
  d, & \text{if } b = x \\
  a(b), & \text{otherwise.}
\end{cases}
\]

A free variable is represented by a reference to its own location in memory. Consequently, \(\sigma(\rho(x)) = \rho(x)\) if \(x\) is a free variable. Initially, a constraint store \(\gamma\) is empty, i.e. it is initialised with \(\text{true}\). During execution of a program, constraints may incrementally be added to the store. This is done by imposing a conjunction of the existing constraints and a new constraint, thus replacing the constraint store by the new conjunction. As a result, the constraint store is typically described by a conjunction of atomic boolean expressions. We treat the constraint solver as a black box. In our implementation, we use the external constraint solver JaCoP [11] in its most recent version 4.4. In fact, the constraint solver is exchangeable and any system of constraint solvers fulfilling our requirements can be used.\(^2\)

Note that our definition of functions does not fully cover the concept of methods in object-oriented languages, since we abstract from classes and, therefore, inheritance. However, a function in our semantics can be compared to a static method, since a function in this semantics can access and modify its own arguments and variables, but not instance variables of an object. Static fields could be modelled as global variables, i.e. further entries in \(\rho_0\).

Since classes, inheritance, instance variables, and static variables have little influence on the interaction between imperative statements and free variables and non-determinism, object orientation can be considered (almost) orthogonal to our work.

### 3.1 Semantics of Expressions

Let us start with the semantics of expressions. The semantics of expressions is described by a relation \(\rightarrow \subseteq (\text{Expr} \times \text{Env} \times \Sigma \times CS) \times (\mathbb{B} \cup \text{Tree}(\mathcal{A}, \mathcal{Z})) \times (\Sigma \times CS)\), which we use in infix notation. Note that evaluating an expression can, in general, change state and constraint store as a side effect, although only few rules actively do so. We will point out expressions that make use of this, whereas

\(^2\) A very simple constraint solver could just take equality constraints into account. In this case, \(\gamma \models x == v\), if \(\gamma = b_1 \land \ldots \land b_k\) and for some \(j \in \{1, \ldots, k\}\) \(b_k = (x == v)\).
the others merely propagate changes (if any) resulting from the evaluation of
subexpressions.

The treatment of constants and variables is trivial.

\[ \langle c, \rho, \sigma, \gamma \rangle \rightarrow \langle c, \sigma, \gamma \rangle \], \text{ if } c \in \mathbb{Z} \cup \mathbb{B} \quad \text{(Con)}

\[ \langle x, \rho, \sigma, \gamma \rangle \rightarrow \langle \sigma(\rho(x)), \sigma, \gamma \rangle \] \quad \text{(Var)}

Nested arithmetic expressions without free variables are evaluated directly, whereas expressions comprising free variables result in a (deterministic) unevalu-
ated (!) symbolic expression (\( \in Tree(A, Z) \)).

\[ \langle e_1, \rho, \sigma, \gamma \rangle \rightarrow \langle v_1, \sigma_1, \gamma_1 \rangle, \quad \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow \langle v_2, \sigma_2, \gamma_2 \rangle, \quad v_1, v_2, v = v_1 \oplus v_2 \in \mathbb{Z} \] \quad \text{(AOp1)}

\[ \langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow \langle v, \sigma_2, \gamma_2 \rangle \]

\[ \langle e_1, \rho, \sigma, \gamma \rangle \rightarrow \langle v_1, \sigma_1, \gamma_1 \rangle, \quad \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow \langle v_2, \sigma_2, \gamma_2 \rangle, \quad \{v_1, v_2\} \notin Z \] \quad \text{(AOp2)}

A boolean expression of the form \( e_1 \oplus e_2 \) is evaluated analogously.

Coherent with Java, conjunctions of boolean expressions are evaluated non-
strictly. The rules for the non-strict boolean disjunction operator \(||\) are defined
analogously to the following rules for \& \& .

\[ \langle b_1, \rho, \sigma, \gamma \rangle \rightarrow \langle v_1, \sigma_1, \gamma_1 \rangle, \quad \gamma \models \neg v_1 \] \quad \text{(And1)}

\[ \langle b_1 \& b_2, \rho, \sigma, \gamma \rangle \rightarrow \langle \text{false}, \sigma_1, \gamma_1 \rangle \]

\[ \langle b_1 \& b_2, \rho, \sigma, \gamma \rangle \rightarrow \langle v_1, \sigma_1, \gamma_1 \rangle, \quad \gamma \models v_1, \quad \langle b_2, \sigma_1, \gamma_1 \rangle \rightarrow \langle v_2, \sigma_2, \gamma_2 \rangle \] \quad \text{(And2)}

We consider a function invocation to be an expression as well, as the caller can
use its result in a surrounding expression. Evaluation of the function is likely to
result in a state change as well as in additions to the constraint store. Invoking \( m \)
implies that its description \( \rho(m) \) is looked up and corresponding fresh addresses \( \alpha_1, \ldots, \alpha_k \), one for each of its \( k \) parameters, are created. The corresponding
memory locations are initialised by the caller. Note that the respective values
 contain free variables. \( \sigma_{k+1}(\alpha_0) \) will contain the return value from evaluating the 
return statement in the body, whose semantics will be defined later (cf. Ret rule).

As the compiler enforces the presence of a return statement, we can safely assume that \( \sigma_{k+1}(\alpha_0) \) holds a value after reducing \( s \). Invoke resets that
value to \( \bot \) for further executions in the caller. We use the shorthand notation \( \bar{a}_k = (a_1, \ldots, a_k) \) for vectors of \( k \) elements.

\[ \langle e_1, \rho, \sigma, \gamma \rangle \rightarrow \langle v_1, \sigma_1, \gamma_1 \rangle, \quad \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow \langle v_2, \sigma_2, \gamma_2 \rangle, \ldots, \]

\[ \langle e_k, \rho, \sigma_{k-1}, \gamma_{k-1} \rangle \rightarrow \langle v_k, \sigma_k, \gamma_k \rangle, \quad \rho(m) = \langle \bar{x}_k, s \rangle, \]

\[ \langle s, \rho_0[\bar{x}_k/\alpha_k], \sigma_k[\bar{a}_k/\varepsilon_k], \gamma_k \rangle \approx \langle \bar{p}_{k+1}, \sigma_{k+1}, \gamma_{k+1} \rangle, \quad \sigma_{k+1}(\alpha_0) = r \]

\[ \langle m(e_1, \ldots, e_k), \rho, \sigma, \gamma \rangle \rightarrow \langle r, \sigma_{k+1}[\alpha_0/\bot], \gamma_{k+1} \rangle \] \quad \text{(Invoke)}
3.2 Semantics of Statements

Next, we will describe the semantics of statements by a relation

\[ \rightsquigarrow \subset (\text{Stat} \times \text{Env} \times \Sigma \times \text{CS}) \times (\text{Env} \times \Sigma \times \text{CS}) \],

which we also use in infix notation.

A variable declaration changes the environment by reserving a fresh memory location \( \alpha \) for that variable. A free variable is represented by a reference to its own location. Enclosing declarations in a block ensures that changes of the environment stay local.

\[ \langle \text{int} \ x, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho[x/\alpha], \sigma, \gamma) \quad \text{(Decl)} \]

\[ \langle \text{int} \ x \ \text{free}, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho[x/\alpha], \sigma[\alpha/\alpha], \gamma) \quad \text{(Free)} \]

\[ \langle s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1) \quad \text{(Block)} \]

As a particularity of a constraint-logic OO language, an assignment \( x = e \) cannot just overwrite a location in memory corresponding to \( x \), since this might have an unwanted side effect on constraints involving \( x \) and referring to its old value. This side effect might turn these constraints unsatisfiable. We avoid this by assigning a new memory address \( \alpha_1 \) to the variable on the left-hand side. At the new address, we store the result from evaluating the right-hand side. Consequently, old constraints or expressions that involve the former value of \( x \) are deliberately left untouched by the assignment. Later uses of the variable refer to its new value.

\[ \langle e, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1) \quad \text{(Assign)} \]

Since the syntax does not enforce that no statements follow a \text{return} statement, we provide sequence rules that take into account that the state may hold a value in \( \alpha_0 \) (indicating a preceding \text{return}) or not (\( \bot \)). Further statements are executed iff the latter is the case. Otherwise, further statements are discarded as a preceding \text{return} has already provided a result in \( \alpha_0 \).

\[ \langle s_1, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1), \quad \sigma_1(\alpha_0) = \bot, \]

\[ \langle s_1 s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_2, \sigma_2, \gamma_2) \quad \text{(Seq)} \]

\[ \langle s_1, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1), \quad \sigma_1(\alpha_0) \neq \bot \]

\[ \langle s_1 s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1) \quad \text{(SeqFin)} \]

The two following rules for if-statements introduce non-determinism in case that the constraints neither entail the branching condition nor its negation. \(^3\)

\[ \langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \quad \gamma_1 \neq v, \quad \langle s_1, \rho, \sigma_1, \gamma_1 \land v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2) \]

\[ \langle \text{if} \ (b) \ s_1 \ \text{else} \ s_2, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2) \quad \text{(If)} \]

\(^3\) In the implementation, the applicability of these rules will depend on the constraint propagation abilities of the employed constraint solver. We discuss the implications in Section 5.
As with if, while can also be behave non-deterministically.

\[ \langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \gamma_1 \not\models v, \langle s_2, \rho, \sigma_1, \gamma_1 \land \neg v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2) \] (If)

\[ \langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \gamma_1 \not\models \neg v, \langle s, \rho, \sigma_1, \gamma_1 \land v \rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2), \langle \text{while } (b) \ s, \rho_1, \sigma_2, \gamma_2 \rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_3) \] (Wh)

\[ \langle b, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1), \gamma_1 \not\models v \]
\[ \langle \text{while } (b) \ s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho, \sigma_1, \gamma_1 \land \neg v) \] (Whf)

All branching rules If, If, Whf, and Wh could be accompanied by more efficient ones that deterministically choose a branch if its condition does not involve free variables, i.e. without having to consult the constraint store. We omit these rules in an effort to keep our definitions concise, as the provided ones can also handle these cases.

We assume that the code of a user-defined function is terminated by a return statement, i.e. its existence has to be ensured by the compiler. The corresponding return value is supplied to the caller by storing it in \( \alpha_0 \), causing remaining statements of the function to be skipped (cf. rule SeqFin), and letting the caller extract the result from \( \alpha_0 \) (cf. rule Invoke). The return statement is handled as follows:

\[ \langle e, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_1, \gamma_1) \]
\[ \langle \text{return } e, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho, \sigma_1[\alpha_0/v], \gamma_1) \] (Ret)

Furthermore, we do not define an evaluation rule involving a fail statement. This is intentional, as the evaluation of such a statement leads to a computation that fails immediately.

The following (optional) substitution rule allows to simplify expressions and results.

\[ \gamma \models x == v, \langle s, \rho, \sigma[x/v], \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1) \]
\[ \langle s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1) \] (Subst)

When variables are not sufficiently constrained to concrete values, labeling can be used to substitute variables for values that satisfy the imposed constraints [5]. This non-deterministic rule is applied with the least priority, i.e. it should only be used if no other rule can be applied. Otherwise, it would result in a lot of non-deterministic branching and prevent the constraint solver from an efficient reduction of the search space by constraint propagation.

\[ \gamma \not\models \alpha \neq v, \langle s, \rho, \sigma[\alpha/v], \gamma \land (\alpha == v) \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1) \]
\[ \langle s, \rho, \sigma, \gamma \rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1) \] (Label)
4 Example Evaluation

We demonstrate the use of the reduction rules defined in Section 3 by computing one possible result of the logarithm program from Listing 1 that will be invoked by an additional method \texttt{int main() \{} \texttt{return log(1); \}}. Other possible results can be computed analogously. We abbreviate the code of \texttt{log} and \texttt{pow} by \texttt{s1} and \texttt{s3}, respectively, to improve readability. The substatement \texttt{s2} is included in \texttt{s1}, while \texttt{s3} includes the substatements \texttt{s4}, \texttt{s5}, and \texttt{s6}. Moreover, we use the infix notation for nested expressions, e.g. we write \texttt{n \ log \ s} instead of \texttt{\log(n \ s)}.

Performing an entire evaluation with this example is interesting, but lengthy. We therefore moved details of the evaluation into the appendix and use the opportunity to highlight some interesting evaluations here.

Initially, let \( \rho_0 = \{ \texttt{main} \implies (\epsilon, \texttt{return \ log(1)\};, \ \texttt{log} \implies ((x), s_1), \ \texttt{pow} \implies ((b, y), s_3) \} \). Furthermore, let \( \gamma_1 = \texttt{true} \) and \( \sigma_0 = \{ \alpha_0 \implies \perp \} \). We begin in method \texttt{main()} (Lemma 1), which evaluates to

\[
\langle 1, \rho_0, \sigma_0, \gamma_1 \rangle \rightarrow \langle 1, \sigma_0, \gamma_1 \rangle \ (\text{Con}), \ \rho_0(\texttt{log}) = ((x), s_1), \ \sigma_0(\alpha_0) = 0 \ (\text{Invoke}) \ \frac{\langle \texttt{log}(1), \rho_0, \sigma_0, \gamma_1 \rangle \rightarrow \langle 0, \sigma_0[\alpha_0/\perp], \alpha_2 = 0 \rangle}{\langle \texttt{return \ log}(1), \rho_0, \sigma_0, \gamma_1 \rangle \rightsquigarrow \langle \rho_0, \sigma_0[\alpha_0/0], \alpha_2 = 0 \rangle} \ (\text{Ret})
\]

The corresponding detailed evaluation is provided in Lemma 9 in the appendix. In the final state, \( \sigma_6 = \sigma_0[\alpha_0/0, \alpha_1/1, \alpha_2/\alpha_2, \alpha_3/2, \alpha_4/\alpha_2, \alpha_7/0, \alpha_8/1] \).

The final result \( \sigma_6(\alpha_0) = 0 \) results from the constraint \( \alpha_2 \leq 0 \) obtained from evaluating \( \texttt{Wh}_f \) (Lemma 9 in the appendix provides context):

\[
\langle i, \rho_4, \sigma_3, \gamma_1 \rangle \rightarrow \langle 0, \sigma_3, \gamma_1 \rangle \ (\text{Var}), \ \frac{\langle y, \rho_4, \sigma_3, \gamma_1 \rangle \rightarrow \langle \alpha_2, \sigma_3, \gamma_1 \rangle \ (\text{Var})}{\langle \alpha \leq y, \rho_4, \sigma_3, \gamma_1 \rangle \rightarrow \langle 0 < \alpha_2, \sigma_3, \gamma_1 \rangle \ (\text{AOp2}), \ \gamma \not\|= (0 < \alpha_2)} \ (\text{Wh}_f)
\]

where \( \rho_4 = \rho_0[b/\alpha_3, y/\alpha_4, i/\alpha_7, r/\alpha_8] \) and \( \sigma_0[\alpha_1/1, \alpha_2/\alpha_2, \alpha_3/2, \alpha_4/\alpha_2, \alpha_7/0, \alpha_8/1] \). \( \alpha_2 \leq 0 \) is refined to \( \alpha_2 = 0 \) by the labeling rule in Lemma 2. If we had non-deterministically chosen rule \( \texttt{Wh}_f \) in Lemma 9, we would have performed an iteration of the \texttt{while} loop, leading to more computations that would not result in solutions, as they would be discarded as incorrect by the \texttt{fail} statement of the \texttt{log} method.

The labeling rule refines the constraint \( \alpha_2 \leq 0 \) in Lemma 2. The constraint store is used to deduce that \( \alpha_2 = 0 \) is consistent with the current constraint, \( \alpha_2 \leq 0 \). Therefore, labeling non-deterministically imposes the more restrictive constraint \( \alpha_0 = \texttt{true} \). Other branches may impose further constraints consistent with \( \alpha_2 \leq 0 \).
5 Discussion

The key aspect of the semantic rules for the presented core language is the interaction between constraint-logic programming and imperative programming. Some aspects of it offer themselves for thorough discussion.

The (potentially) non-deterministic evaluation of our rules If\(_f\), If\(_t\), Wh\(_f\), and Wh\(_t\) highly depends on the included constraint solver. Our definition allows to follow a branch if the negation of its condition is not entailed by the current constraint store \(\gamma\). When implementing this, a constraint solver will be used to check whether \(\gamma \not|= \neg v\) (analogously for \(\gamma \not|= v\)). If the constraint solver is not able to show that the constraints entail \(\neg v\), this may have three reasons: 1) \(\gamma |= v\), or 2) the current constraints neither entail \(v\) nor \(\neg v\), or 3) the constraint propagation abilities of the employed constraint solver are insufficient to show that \(\gamma |= \neg v\), but in fact \(\gamma |= \neg v\). In case 1), the system behaves deterministically and only one rule for if (or while) will be applied. In case 2), one of the two rules for if (or while) can be chosen non-deterministically. Only case 3) is problematic. In this case, a branch can be chosen that corresponds to inconsistent constraints. In practice, solvers do not achieve perfect constraint propagation and also no global consistency of the constraints. Consequently, results corresponding to inconsistent constraints may only be discovered later, e.g. during labeling. In the meantime, non-backtrackable statements (e.g. ones that result in input/output) of search regions may have been executed in branches that prove infeasible later. Thus, we suggest to avoid input / output in search regions.

Moreover, we see two options to handle the mentioned problem. The first option is to explicitly label variables sufficiently at every branch such that the constraint solver is able to either infer \(\gamma |= v\) or \(\gamma |= \neg v\). However, as explained in context of the rule Label, this also introduces a lot of non-deterministic branching by creating one branch per label. Therefore, the effectivity of constraint propagation is reduced and the overall effort for search is increased.

The second option is to wait until a solution of the encapsulated search has been found. In fact, this will merely be a potential solution, given that the corresponding constraints are also satisfiable. Thus, at this point we could perform (sufficient) labeling, until it is clear whether the constraints are actually satisfiable. Therefore, encapsulated search produces a stream of pairs, each of which comprises one potential solution and its corresponding set of constraints. The enclosing application can iterate over this stream and try to label each solution, thus ruling out infeasible solutions afterwards. The implementation of Muli provides an explicit label operation, which the application developer can use for this purpose. We decided not to do this implicitly in order to give the developer more flexibility. It is easy to wrap this functionality into a search operation which labels every found solution implicitly.

Both mentioned options are available to the developer. We recommend the second one, possibly in the wrapped version with implicit labeling. For search regions that involve only backtrackable statements, the result does not depend on the chosen option, but the second option is presumably more efficient as fewer branches have to be evaluated. For other search regions, only the first option
can avoid unwanted side effects of illegally accessed branches. However, search then becomes less efficient. Therefore, in case that non-backtrackable side effects have to be avoided, we recommend that the developer removes input/output operations from search regions and moves them behind encapsulation instead.

Formalising the operational semantics of Muli has also helped uncover some operations whose semantics are sufficiently clear in deterministic Java, but become ambiguous when non-determinism and symbolic execution are added. Consequently, some alternatives could be discussed on a conceptual level using this semantics, before deriving a corresponding implementation. This particularly involves the interpretation of symbolic variables (rules Invoke and Var) and assignments, as outlined subsequently.

By rule Assign, an assignment \( x = e \) creates a new memory address for the variable \( x \) and changes the environment accordingly. As a result, memory usage of a Muli program is increased with every assignment, instead of with every declaration of a variable as in imperative OO languages. Nevertheless, this behaviour is required in order to avoid unwanted side effects on previous constraints involving \( x \). The alternative, mutating \( \sigma(\rho(x)) \) directly, would result in assignments to \( x \) that could render constraints involving \( x \) unsatisfiable ex post, i.e., after branching has occurred that involves such a constraint.

As another consequence, rule Assign ensures that the interpretation of symbolic variables is equivalent to that of regular values. Consider the simple excerpt from a Java program given in Listing 2 as an example: After evaluating the last line, \( y \) is still expected to be 5, even though \( x \) now holds a different value. After all, although primitive variables can be directly mutated in Java, their previous interpretations cannot. Similarly, for symbolic values, rule Assign ensures that references before and after an assignment are treated distinctly, even though memory efficiency is adversely affected. Nevertheless, unreferenced former meanings of a variable may be destroyed by the garbage collector, thus reclaiming (some) memory.

```java
int x = 5; int y = x;
x = 3;
```

**Listing 2.** Minimal example demonstrating that variables may be mutated directly, in constrast to their uses: After evaluation, \( y \) is 5.

Implicitly, our rules Assign (or Invoke) and Var enable sharing of symbolic values. Assigning a free variable \( x \) to another free variable \( y \) means that the address \( \rho(x) \) of \( x \) is stored in the memory location corresponding to \( y \) by modifying state as \( \sigma[\rho(y)/\rho(x)] \). Consequently, subsequent constraints and expressions that involve either variable will actually reference the same variable. The sharing behaviour is exhibited in the example in Lemma 5 where a free variable is passed to the `pow` method as its second parameter. `pow` adds constraints to that variable that only come into effect when labeling is performed in its invoking context in `log` (Lemma 2).

Regarding backtracking, the implementation is only implicitly affected by the presented operational semantics. Here, the semantics defines the desired state of the overall VM that must be achieved before evaluation in terms of \( \rho \in Env \),
\( \sigma \in \Sigma, \gamma \in CS \). Considering the multitude of options for achieving the desired VM state that lend themselves for the implementation, we briefly outline the options without prescribing either. Firstly, “don’t care” non-determinism considers only one evaluation alternative and therefore does not require backtracking at all. Secondly, it would be possible to fork at statements that introduce non-determinism, thus evaluating all alternatives in parallel. This does not require backtracking either, however, consider that this generates a lot of overhead in terms of memory and computation, as the VM must be forked in its entirety to accommodate for any side effects, and as all forks must be joined in order to fall back to deterministic computation after a search region is fully processed. Thirdly, the alternatives can be evaluated sequentially. To achieve this, the VM must record changes to the data structures on a trail equivalent to that of Prolog in order to reconstruct a previous state during backtracking. Our implementation resorts to the latter option using a trail. Nevertheless, the remaining options would also be interesting to pursue.

## 6 Related Work

To the best of our knowledge, this paper is the first to present a formal semantics of an imperative language enhanced by features of constraint-logic programming. For sake of clarity we focused on a core language. A full formal semantics of Java alone may require an entire book as in the work by Stark et al. [19]. K-Java [2] is another approach to define a formal semantics of Java. However, in the cited paper the authors focus on selected aspects of the language. The official semantics of Java is extensively described in natural language (cf. [6, 12]).

Some existing core languages of Java such as Featherweight Java [9] are tailored to the investigation of the typing system and not meant to be executable. Hairry [7] investigates an object-oriented core language focussing on computational complexity.

The encapsulated search of Muli has been inspired and adapted from the corresponding feature of the functional-logic language Curry. An operational semantics of Curry can be found in [1]. It is simpler than our semantics, since Curry is purely declarative and does not have to bother with side effects.

Approaches for integrating object-oriented features into a (constrained) logic language are e.g. Oz [20], Visual Prolog [17], Prolog++ [15], and Concurrent Prolog [18]. However, these approaches maintain a declarative flavour and mainly provide syntactic sugar for object-orientation. They are unfamiliar for mainstream object-oriented programmers.

There are also approaches which add constrained-logic features to an imperative / object-oriented language. Typically, the integration is less seamless than in Muli and the language parts stemming from different paradigms can clearly be distinguished [3, 4]. LogicJava [14] is more restrictive than Muli and only allows class fields to be logic variables. Moreover, entire methods have to be declared as searching or non-searching. Finally, a project on the extension of the imperative language Rust by symbolic execution and constraint solving, aimed
7 Conclusions and Future Work

Our work formalises an operational reduction semantics for a core of the novel integrated constraint-logic object-oriented language Muli. Muli extends Java by logic variables, non-determinism, encapsulated search, and constraint solving. Muli is particularly suited for enterprise applications that involve both searching and non-searching business logic. Encapsulated search ensures that non-determinism is only introduced deliberately where needed, instead of spreading out over the whole program. Thus, the code outside of encapsulated search regions behaves just as ordinary Java code.

The presented operational semantics provides a basis for implementations of compiler, symbolic JVM, and tools for processing Muli. In particular, the formalisation has helped clarify possible ambiguities w. r. t. the semantics of certain statements under non-determinism, such as that of assignments to variables and uses of them. Furthermore, the semantics will facilitate reasoning about programs developed in Muli. We made the symbolic JVM that executes Muli programs available as free software on GitHub.4

As future work, we would like to extend our core language and its semantics by more features of Java, such as classes and inheritance. We expect these additions to be quite orthogonal to the presently supported concepts. However, when (non-deterministically) instantiating a free variable with an object type, we have to take the whole corresponding inheritance hierarchy into account.

References


Appendix: Full Example Evaluation

In addition to \( \rho_0, \sigma_0, \) and \( \gamma_1 \) defined in section Section 4, the following auxiliary definitions will be needed as intermediate results: \( \rho_1 = \rho_0[x/\alpha_1, y/\alpha_2], \rho_2 = \rho_0[y/\alpha_3, y/\alpha_4], \rho_3 = \rho_2[i/\alpha_5, r/\alpha_6], \rho_4 = \rho_3[i/\alpha_7, r/\alpha_8], \sigma_1 = \sigma_0[\alpha_1/1, \alpha_2/\alpha_2], \sigma_2 = \sigma_1[\alpha_3/2, \alpha_4/\alpha_2], \sigma_3 = \sigma_2[\alpha_7/0, \alpha_8/1], \sigma_4 = \sigma_3[\alpha_0/1], \sigma_5 = \sigma_4[\alpha_0/\perp], \sigma_6 = \sigma_5[\alpha_0/0], \gamma_2 = \gamma_1 \land \alpha_2 \leq 0, \) and \( \gamma_3 = \gamma_2 \land \alpha_2 == 0. \) To simplify the understanding of the full computation provided in Section 4, we have decomposed it into a couple of lemmas. We present the computation in a top-down fashion. If you prefer a bottom-up fashion, just read the lemmas in reverse order. The names of the applied rules are given in each step.

\[
\langle \text{int y free} \rangle \Rightarrow (\rho_0[x/\alpha_1, y/\alpha_2], \sigma_0[\alpha_1/1, \alpha_2/\alpha_2], \gamma_1) \quad \text{(Free), (Lemma 2)}
\]

\[
\langle \text{int y free; s2, } \rho_0[x/\alpha_1], \sigma_0[\alpha_1/1], \gamma_1 \rangle \Rightarrow (\rho_1, \sigma_6, \gamma_3) \quad \text{(Seq)}
\]

\[
\langle \text{int y free; s2, } \rho_0[x/\alpha_1], \sigma_0[\alpha_1/1], \gamma_1 \rangle \Rightarrow (\rho_1, \sigma_6, \gamma_3)
\]

\[
\langle \text{int y free: s2, } \rho_0[x/\alpha_1], \sigma_0[\alpha_1/1], \gamma_1 \rangle \Rightarrow (\rho_1, \sigma_6, \gamma_3)
\]

\[
\langle \text{int y free} \rangle \Rightarrow (\rho_0[x/\alpha_1, y/\alpha_2], \sigma_0[\alpha_1/1, \alpha_2/\alpha_2], \gamma_1) \quad \text{(Free), (Lemma 2)}
\]

\[
\langle \text{int y free; s2, } \rho_0[x/\alpha_1], \sigma_0[\alpha_1/1], \gamma_1 \rangle \Rightarrow (\rho_1, \sigma_6, \gamma_3) \quad \text{(Seq)}
\]
\( \text{Lemma 3}, \ \gamma_2 \not\models \neg \text{true}, \ \gamma_2 \not\models \alpha_2 \neq 0, \ \text{(Lemma 4)} \)  
\( \langle \text{return } y; \rho_1, \sigma_5, \gamma_2 \rangle \rightsquigarrow (\rho_1, \sigma_6, \gamma_3) \)  
\( \text{(If)} \)  
\( \langle s_2, \rho_1, \sigma_1, \gamma_1 \rangle \rightsquigarrow (\rho_1, \sigma_5, \gamma_2) \)  
\( \text{(Lemma 2)} \)

\( \langle x, \rho_1, \sigma_5, \gamma_2 \rangle \rightarrow (1, \sigma_5, \gamma_2) \) \( \text{(Var)} \),  
\( 1 == 1 = \text{true} \)  
\( \langle \text{pow}(2, y) == x, \rho_1, \sigma_1, \gamma_1 \rangle \rightarrow (\text{true}, \sigma_5, \gamma_2) \)  
\( \text{(AOp1)} \)

\( \langle y, \rho_1, \sigma_5[\alpha_2/0], \gamma_2 \land \alpha_2 == 0 \rangle \rightarrow (0, \sigma_5, \gamma_3) \) \( \text{(Var)} \),  
\( \langle \text{return } y; \rho_1, \sigma_5, \gamma_2 \rangle \rightsquigarrow (\rho_1, \sigma_5[\alpha_0/0], \gamma_3) \)  
\( \text{(Ret)} \)

\( \langle y, \rho_1, \sigma_1, \gamma_1 \rangle \rightarrow (2, \sigma_1, \gamma_1) \) \( \text{(Con)} \),  
\( \langle y, \rho_1, \sigma_1, \gamma_1 \rangle \rightarrow (\alpha_2, \sigma_1, \gamma_1) \) \( \text{(Var)} \),  
\( \rho_1(\text{pow}) = ((b, y), s_3) \),  
\( \text{(Lemma 6)} \),  
\( \sigma_4(\alpha_0) = 1 \)  
\( \langle \text{pow}(2, y), \rho_1, \sigma_1, \gamma_1 \rangle \rightarrow (1, \sigma_4[\alpha_0/1], \gamma_2) \)  
\( \text{(Invoke)} \)

\( \langle \text{int } i; \rho_2, \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_2[i/\alpha_3], \sigma_2, \gamma_1) \) \( \text{(Decl)} \),  
\( \langle \text{int } r; \rho_2[i/\alpha_3], \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_2[i/\alpha_5, r/\alpha_6], \sigma_2, \gamma_1) \) \( \text{(Decl)} \),  
\( \langle \text{int } i; \text{int } r; i == 0; r == 1; s_4, \rho_2[i/\alpha_3], \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2) \)  
\( \text{(Seq)} \)

\( \langle i == 0; r == 1; s_4, \rho_2[i/\alpha_3], \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2) \)  
\( \text{(Seq)} \)

\( \langle i == 0; r == 1; s_4, \rho_3, \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2) \)  
\( \text{(Seq)} \)

\( \langle r == 1; s_4, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1) \)  
\( \text{(Lemma 8)} \)

\( \langle r == 1; s_4, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1) \)  
\( \text{(Assign)} \)

\( \langle i == 0; r == 1; s_4, \rho_3, \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2) \)  
\( \text{(Seq)} \)

\( \langle r == 1; s_4, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2) \)  
\( \text{(Seq)} \)

\( \langle i == 0; r == 1; s_4, \rho_3, \sigma_2, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2) \)  
\( \text{(Lemma 7)} \)

\( \langle r, \rho_1, \sigma_3, \gamma_2 \rangle \rightarrow (1, \sigma_3, \gamma_2) \) \( \text{(Var)} \),  
\( \langle \text{return } r; \rho_4, \sigma_3, \gamma_2 \rangle \rightsquigarrow (\rho_4, \sigma_3[\alpha_0/1], \gamma_2) \)  
\( \text{(Ret)} \)

\( \langle s_5; \text{return } r; \rho_4, \sigma_3, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_3[\alpha_0/1], \gamma_2) \)  
\( \text{(Seq)} \)

\( \langle i, \rho_4, \sigma_3, \gamma_1 \rangle \rightarrow (0, \sigma_3, \gamma_1) \) \( \text{(Var)} \),  
\( \langle y, \rho_4, \sigma_3, \gamma_1 \rangle \rightarrow (\alpha_2, \sigma_3, \gamma_1) \) \( \text{(Var)} \),  
\( \langle i < y, \rho_4, \sigma_3, \gamma_1 \rangle \rightarrow (0 < \alpha_2, \sigma_3, \gamma_1) \) \( \text{(AOp2)} \),  
\( \gamma \not\models (0 < \alpha_2) \)  
\( \langle \text{while } (i < y) \ s_6, \rho_4, \sigma_3, \gamma_1 \rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1 \land \neg(0 < \alpha_2)) \)  
\( \text{(Whf)} \)  
\( \text{(Lemma 9)} \)
Hypertree Decomposition: The First Step Towards Parallel Constraint Solving

Ke Liu, Sven Loeffler, Petra Hofstedt
Brandenburg University of Technology Cottbus-Senftenberg, Germany

Abstract. Parallel constraint solving is a promising way to enhance the performance of constraint programming. Yet, current solutions for parallel constraint solving ignore the importance of hypergraph decomposition when mapping constraints onto cores. This paper explains why and how the hypergraph decomposition can be employed to relatively evenly distribute workload in parallel constraint solving. We present our dedicated hypergraph decomposition method det-k-CP for parallel constraint solving. The result of det-k-CP, which conforms with four conditions of hypertree decomposition, can be used to allocate constraints of a given constraint network to cores for parallel constraint solving. Our benchmark evaluations have shown that det-k-CP can relatively evenly decompose a hypergraph for specific scale of constraint networks. Besides, we obtained competitive execution time as long as the hypergraphs are sufficiently simple.

Key Words: parallel constraint solving, hypertree decomposition

1 Introduction

Structural decomposition methods are one of research hot spots both in the area of relational databases and constraint programming. Many NP-complete and NP-hard problems can be solved in polynomial time if the corresponding hypergraph has the bounded hypertree-width, which indicates that the original intractable problem can be divided into a number of tractable subproblems [1]. In addition, the tree structure for a constraint network implies that each node of the tree decomposition can be solved simultaneously, which makes us naturally think of utilizing parallel computing to solve constraint satisfaction problem (CSP). In other words, the acyclic structure of constraint networks means that the given CSP problem is tractable and parallelizable [2–4]. Several decomposition methods have been developed to convert the cyclic constraint networks to acyclic ones although these methods apply to different types of graph for the given constraint network. For example, join-tree-clustering transforms the primal graph of the given constraint network into the equivalent acyclic network [5]. Cycle-cutset decomposition [6] also works on the primal graph by removing the vertexes that prevent the hypergraph to be acyclic. Some decomposition methods (e.g., hinge decomposition [7], hypertree decomposition [8, 1]), on the other hand, use the hypergraph as its input, and the output of these methods is at least in accord with the conditions for hypertree decomposition defined in [8].
Nevertheless, the decomposition methods, which have been proposed in the literature during the last decades, aim at obtaining as small hypertree width as possible for the hypergraph, because the smaller the width of a hypertree decomposition we obtained, the faster the original CSP problem can be solved [1]. Moreover, previous structural decomposition methods, such as *det-k-decomp* which is the most general decomposition method so far [9], cannot ensure a relatively even distribution of constraints based on our observation of results after running *det-k-decomp*. The algorithm *det-k-decomp* only guarantees the greatest node width of the decomposition tree is *k*, and fairly often, the width of most nodes is far less than *k*. This characteristic of *det-k-decomp* impedes its application in parallel constraint solving.

This paper intends to present a new decomposition *det-k-CP* method with stochastic search procedure for parallel constraint solving. The goal is to provide a mapping algorithm for parallel constraint solving. The idea behind *det-k-CP* is to utilize the property of dual graph that a redundant arc can be removed between two nodes of the graph if there is an alternative path that ensures two nodes still connected. We regard it as the first step towards parallel constraint solving due to the benefits that come from hypertree structure.

The rest of this article is organized as follows. The basic definitions used in this study are presented in Section 2. Section 3 describes the new method in detail, and then analyze the time complexity of *det-k-CP*. In Section 4, we present our experimental results. Finally, we conclude in Section 5.

## 2 Preliminaries

A *constraint network* \( R \) is a triple \((X, D, C)\), which consists of:

- a finite set of variables \( X = \{x_1, \ldots, x_n\} \),
- a set of respective finite domains \( D = \{D_1, \ldots, D_n\} \), where \( D_i \) is the domain of the variable \( x_i \), and
- a set of constraints \( C = \{c_1, \ldots, c_t\} \), where a constraint \( c_j \) is a relation \( R_j \) defined on a subset of variables \( S_j, S_j \subseteq X \).

Any constraint network can be graphically represented by a *hypergraph*. A hypergraph \( \mathcal{H} \) is a tuple \((V, E)\), where \( V \) is a set of vertexes and \( E \) is a set of *hyperedges*. A hyperedge of a hypergraph is composed of an arbitrary number of vertexes, which makes hyperedges fundamentally different from normal edges in a graph. Any constraint in a given constraint network corresponds to a hyperedge in a hypergraph, and the variables of a constraint can be seen as vertexes of a hyperedge.

A *hypertree* of a hypergraph \( \mathcal{H} \) is a triple \((T, \chi, \lambda)\), where \( T = (V_T, E_T) \) is a tree, \( \chi \) and \( \lambda \) are labeling functions. We denote a set of variables for a given node \((\text{node}_i)\) in a hypertree by \( v_i \). Therefore, \( v_i = \chi(\text{node}_i) \) and \( v_i \subseteq 2^{\text{vertices}(\mathcal{H})} \), where \( \text{vertices}(\mathcal{H}) \) are vertexes of hypergraph \( \mathcal{H} \). Similarly, we denote a set of edges of \( \text{node}_i \) by \( e_i \). Therefore, \( e_i = \lambda(\text{node}_i) \) and \( e_i \subseteq 2^{\text{edges}(\mathcal{H})} \),
where \( \text{edges}(\mathcal{H}) \) are the hyperedges of hypergraph \( \mathcal{H} \). By \( \text{root}(T) \) we denote the root of a tree \( T \), for every \( p \in V_T \), let \( T_p \) denote the subtree of \( T \) with root \( p \).

The **width of a hypertree** is the maximum number of hyperedges among the nodes of it, which is given by \( \text{hw}(T) = \max | \lambda(\text{node}_i) | \). **Hypertree decomposition** is a procedure that converts a hypergraph into a hypertree. In order to demonstrate hypertree decomposition on a given constraint network, assume we have a simple problem over a set of variables \( \{x_1, \ldots, x_{10}\} \subseteq X \) modeled by the following constraints

\[
\begin{align*}
\text{allDifferent1}(x_3, x_4, x_5, x_7) & \quad \text{allDifferent2}(x_1, x_4, x_6, x_9) \\
\text{table1}(x_5, x_8, x_{10}) & \quad \text{table2}(x_7, x_8, x_9) \\
\text{atLeastNvalues}(x_1, x_2, x_3) & \quad \text{arithm1}(x_5, x_6) \\
\text{arithm2}(x_6, x_8) & \quad \text{arithm2}(x_6, x_8)
\end{align*}
\]

![Fig. 1. The hypergraph for the constraint network. The example is based on [11]](image)

The hypergraph for this constraint network is depicted in Figure 1, where the variables \( x_i, i \in \{1, \ldots, 10\} \) are the vertices, while the edges are represented by the enclosing ellipses. Figure 2 shows a possible hypertree decomposition of this hypergraph. Note that the hypertree decomposition (Figure 2) can also be

\footnote{The name of the constraints is consistent with the name of constraints used in the Choco Solver [10].}
viewed as a dual graph for the hypergraph (Figure 1). The nodes of a dual graph consist of a set of hyperedges of the corresponding hypergraph, and an edge of the dual graph is due to existing shared variables between two nodes of the dual graph. Formally, $H_{dual}$ for a given $H$ can be represented as a tuple $(S,E)$ in which $S = \{s_1, ..., s_i, ..., s_j\} \subseteq \text{edges}(H)$ and $\forall e \in E = \text{edges}(H_{dual}): s_i \cap s_j = e \Leftrightarrow \text{var}(s_i) \cap \text{var}(s_j) \neq \emptyset$.

$$\{\text{allDifferent1, allDifferent2}\}, \{x_1, x_4, x_6, x_9, x_5, x_7\} \quad \{\text{atLeastNValues}\}, \{x_1, x_2, x_3\} \quad \{\text{table, arithm1}\}, \{x_5, x_6, x_7, x_8, x_9\}$$

$$\{\text{arithm2, table1}\}, \{x_5, x_6, x_8, x_10\}$$

Fig. 2. Hypertree decomposition for the hypergraph of Fig. 1. The example is based on [11]

3 The algorithm det-k-CP

In this section, we present our new algorithm det-k-CP which is designed to decompose a hypergraph for parallel constraint solving. The $k$ of det-k-CP means the number of nodes in the decomposition tree. Roughly speaking, the target of det-k-CP is to decompose a given constraint network $N$ to a degenerate tree in which each internal node has exactly one child, so that the solutions of $N$ can be found in time linear after each node is solved independently. Because an acyclic constraint network can be solved efficiently [5]. For example, figure 3 depicts a target degenerate decomposition tree with eight nodes decomposed by det-k-CP for a multi-core processor with eight cores.

Fig. 3. A constraint network is partitioned into eight parts. An edge between two nodes is due to the shared variables.

For a given ordering of nodes of a degenerate decomposition tree $T$ generated by det-k-CP, there is an edge between two nodes because there exists shared variables between two nodes. Additionally, we only keep the edges between two
adjacent nodes and eliminate the edges between any pair of non-adjacent nodes. The mechanism of elimination is based on the property of which any edge on a circuit formed by common shared variables of a dual graph can be removed without changing the set of all solutions for the constraint network [5]. For instance, in Figure 2, the edge between the root node and the right leaf node caused by shared variables \((x_5, x_6)\) is inexistent because there are shared variables \((x_5, x_6)\) between root node and its child node, as well as the right leaf node and its parent node respectively.

The positional relation between two nodes in \(T\) is either adjacent or non-adjacent; thus a pair of nodes in \(T\) can be denoted as \((N_i, N_j)\) for non-adjacent or \((N_p, N_{p+1})\) for adjacent respectively, where \(|i - j| \geq 2\), \(0 \leq i < j \leq k\) and \(p \in \{i, ..., j - 1\}\). Mathematically, a decomposed graph after decomposition by \(det-k-CP\) must meet the following two conditions:

\[
\forall p \in \{i, ..., j - 1\}: \quad \chi(N_i) \cap \chi(N_j) \subseteq \chi(N_p) \cap \chi(N_{p+1}) \quad (1)
\]

\[
\forall N_i \in T: \quad \cup_{i=0}^{k-1} \chi(N_i) = \chi(T), \text{ and } \cup_{i=0}^{k-1} \lambda(N_i) = \lambda(T) \quad (2)
\]

where the first condition means that the shared variables between any pair of nodes \((N_p, N_{p+1})\) in the interval \([i..j]\) must contain the shared variables between \((N_i, N_j)\), whereas the second condition guarantees that the decomposition method \(det-k-CP\) does not lose any constraint or variable.

Having these two conditions we can easily validate if a given decomposition tree is successfully decomposed by \(det-k-CP\). Besides, if a decomposition tree satisfies the conditions for \(det-k-CP\), it must meet the four conditions defined in [8] for hypertree decomposition.

**Algorithm 1** We are now going to explain the \(det-k-CP\) in more detail. In order to relatively evenly distribute constraints, line 1 of Algorithm 1 sorts the constraints based on its computation requirement (weight), which depends on many factors, such as, the time complexity of constraint propagator used by the constraint solver, the number of variables of the constraint, and the size of intervals of these variables. Then, after sorting procedure the constraints are inserted into an array \((array\_Nodes)\) with length \(k\) in turn. Algorithm 1 runs into the loop in line 7-13 until a qualified solution is found. The heuristic for swap procedure in line 12 of Algorithm 1 has many choices, for example, random exchange, switching nodes that has the fewest and most number of constraints, or changing the permutation in lexicographic order of the indexes of \(array\_Nodes\) in order. For instance, if the length of \(array\_Nodes\) is 4, we might first use the permutation \((0,1,2,3)\), then \((0,1,3,2)\) and so on.
Algorithm 1: \textit{det-k-CP(N,k)}

\textbf{Input:} A Constraint Network \textit{N}, and the desired number of nodes \textit{k}.
\textbf{Output:} A degenerate hypertree with \textit{k} nodes.

1. Set \textit{list\_LN} = a list which contains sorted constraints of \textit{N} based on weight ;
2. Set \textit{i\_size} = the size of \textit{list\_LN} ;
3. Initialize an array \textit{array\_Nodes} with \textit{k} nodes ;
4. for \textit{i} ← 1 \textbf{to} \textit{i\_size} \textbf{do}
5. \hspace{1em} add \textit{list\_LN}[\textit{i}] into \textit{array\_Nodes}[\textit{i}\%\textit{k}] ;
6. \textbf{end}
7. \textbf{while} \textit{true} \textbf{do}
8. \hspace{1em} getPotentialSolution(\textit{array\_Nodes}) ;
9. \hspace{1em} \textbf{if} \textit{array\_Nodes} pass test conditions (1) and (2) \textbf{then}
10. \hspace{2em} \textbf{break} ;
11. \hspace{1em} \textbf{end}
12. \hspace{1em} Swap nodes in \textit{array\_Nodes} ;
13. \textbf{end}
14. \textbf{return} \textit{array\_Nodes} ;

Algorithm 2 Let us now consider the function \textit{getPotentialSolution} defined by Algorithm 2, which exhaustively covers all the edges between non-adjacent nodes, and removes it if necessary. In order to do so, the starting point of the non-adjacent edge (\textit{i\_start}) is assigned to the third from the last index in line 2 of Algorithm 2, which is the first node that should not connect to the last node (with index \textit{i\_len} - 1). Then, it does decrement at each iteration until it reaches the zero that is the first index of the array in the outer loop; and the ending point of the non-adjacent edge is initialized as \textit{i\_end} = \textit{i\_start} + 2, then the inner loop continues to iterate to the end of the the array (\textit{i\_len} − 1).

Algorithm 2: \textit{getPotentialSolution(array\_Nodes)}

\textbf{Input:} \textit{array\_Nodes}
\textbf{Output:} A potential solution

1. Set \textit{i\_len} = the length of \textit{array\_Nodes} ;
2. Set \textit{i\_start} = \textit{i\_len}-3 ;
3. \textbf{while} \textit{i\_start} ≥ 0 \textbf{do}
4. \hspace{1em} for \textit{i\_end} ← \textit{i\_start} + 2 \textbf{to} \textit{i\_len} − 1 \textbf{do}
5. \hspace{2em} eliminateEdge(\textit{i\_start}, \textit{i\_end}, \textit{array\_Nodes}) ;
6. \hspace{1em} \textbf{end}
7. \hspace{1em} Set \textit{i\_start} = \textit{i\_start} − 1 ;
8. \textbf{end}
Algorithm 3 The function `eliminateEdge` plays an important role in `det-k-CP`. In line 2 of Algorithm 3, `list_2BeEliminated` is assigned to all shared variables between non-adjacent nodes `$N_{i_{start}}$` and `$N_{i_{end}}$`. For each edge between `$N_{i_{start}}$` and `$N_{i_{end}}$`, `eliminateEdge` checks whether or not every edge between adjacent nodes `$N_i$` and `$N_j$`, where $i$ is in the interval `{`$i_{start}$`, ..., $i_{end}$ − 1`}`, and $j = i + 1$, contains all variables that are also included in the input edge between nodes `$N_{i_{start}}$` and `$N_{i_{end}}$`, as shown in line 5 of Algorithm 3.

If an edge between `$N_i$` and `$N_j$` contains all shared variables that the edge between `$N_{i_{start}}$` and `$N_{i_{end}}$` has (in line 5), then the `for` loop runs into the next iteration for the next edge between `$N_{i+1}$` and `$N_{i+1}$`; otherwise, line 6 removes all shared variables of `$N_i$` and `$N_j$` from `list_2BeEliminated`.

The function `getMinimumSetConstraints4Shared`, which is not presented in this paper, returns the smallest set of constraints in terms of cardinality, which covers all variables in the list `list_2BeEliminated`. In line 8-12, the code snippet loops through all constraints obtained by `getMinimumSetConstraints4Shared`, if the constraint is not contained in `$N_j$`, the constraint will be added into `$N_j$`.

By doing so, the edges between non-adjacent nodes `$N_{i_{start}}$` and `$N_{i_{end}}$` can be eliminated. Because the shared variables between `$N_{i_{start}}$` and `$N_{i_{end}}$` are now covered by all edges between adjacent nodes `$N_i$` and `$N_j$` with `{`$i_{start}$`, ..., $i_{end}$ − 1`}`, and $j = i + 1$.

Algorithm 3: `eliminateEdge(i_{start}, i_{end}, array_Nodes)`

```
Input: i_{start}, i_{end}, array_Nodes
1   for i ← i_{start} to i_{end} − 1 do
2       Set list_2BeEliminated = getSharedVariables(i_{start}, i_{end}, array_Nodes);
3       Set j = i + 1;
4       Set list_SharedOnMainPath = getSharedVariables(i, j, array_Nodes);
5       if list_SharedOnMainPath.notContainsAll(list_2BeEliminate) then
6           list_2BeEliminated.removeAll(list_SharedOnMainPath);
7           Set list_2beAddedConstraints = getMinimumSetConstraints4Shared(i, array_Nodes, list_2BeEliminated);
8           foreach constraint cs ∈ list_2beAddedConstraints do
9               if array_Nodes[j] notContained cs then
10                  add cs into array_Nodes[j];
11           end
12       end
13   end
```

So far we have discussed the detailed process of `det-k-CP`. The reason why we introduce randomness into the search process to `Swap` method in line 12
of Algorithm 1 is because a large amount of loops would be incurred if we backtracked to edges eliminated before but the new added constraint makes the edges appear again. For instance, in Figure 3, we added some constraints onto Node 4 due to the elimination process for edge between Node 3 and 5, then the edge between node 4 and 7, which had been removed before, happened to occur again, and this edge forced us to add new constraints onto Node 5 and 6, after that the edge between Node 3 and 5 occurred again, consequently, we would fall into repeated elimination process until the worst case happened in which each node filled up with all the constraints of the given constraint network.

Now, we would like to analyze the time complexity of \textit{det-k-CP}. if we combine the loops in Algorithm 2 with the loop in Algorithm 3 to form a triple-nested loop, the total number of iterations for the triple-nested loop can be calculated by the following recurrence relation:

\begin{align*}
  n_3 &= 2 \\
  n_4 &= 3 + 2 + n_3 \\
  n_5 &= 4 + 3 + 2 + n_4 \\
  &\vdots \\
  n_k &= (k - 1) + (k - 2) + \cdots + 2 + n_{k-1} = \frac{(k^2 - k - 2)}{2} + n_{k-1}
\end{align*}

Where \( n_k \) denotes the number of iterations for the triple-nested loop for \( k \) number of nodes of the target decomposition tree. A recurrence relation for \( \{n_k\} \) can be obtained by considering whenever one node is added to a tree which has \( k - 1 \) nodes; therefore, new non-adjacent nodes are generated, consequently we have to eliminate these edges, where the number of these edges can be summed by \((k - 1) + (k - 2) + \cdots + 2\). For instance, as can be seen in Figure 4, when Node 0 is added to the original tree, the number of times of elimination process is increased by 5. To obtain the explicit formula for this recurrence relation, we solve it with the initial conditions \( n_3 = 2, n_2 = 0 \) and \( n_1 = 0 \). The solution of the recurrence relation is \( n_k = \frac{k^3 - 7k + 6}{6} \), which means the number of loops of the triple-nested loop is exactly \( \frac{k^3 - 7k + 6}{6} \).

At each iteration of the triple-nested loop, from line 2 to 13 of Algorithm 3, the number of executions can be bounded by the number of constraints \((N_c)\) plus the complexity of method \textit{getMinimumSetConstraints4Shared}, denoted by \( \mathcal{O}(N_c) + \mathcal{O}(\text{getMinimumSetConstraints4Shared}) \). Our implementation of \textit{getMinimumSetConstraints4Shared} that is not shown in this paper is bounded by \( \mathcal{O}(N_c) \), where \( N_v \) is the number of variables. Thus, Algorithm 2 is bounded by \( \mathcal{O}\left(\frac{k^3 - 7k + 6}{6} \cdot (N_c + N_v)\right) \). Algorithm 1, which can be viewed as the outermost loop to the entire algorithm, can be bounded by \( \mathcal{O}(k \cdot (N_c - \frac{N_c}{k})) \). That is because the loop of Algorithm 1 eventually stops running when each node is added to contain the whole constraint network. The overall time complexity is, therefore, \( \mathcal{O}(k \cdot (N_c - \frac{N_c}{k}) \cdot \frac{k^3 - 7k + 6}{6} \cdot (N_v + N_c)) \). Therefore, the asymptotic time complexity is \( \mathcal{O}(k^4 \cdot N_c \cdot (N_c + N_v)) \). Note that the runtime may be significantly
smaller in practice since we take into account the worst cases for Algorithm 1 and 3.

\[ \text{Node0} \rightarrow \text{Node1} \rightarrow \text{Node2} \rightarrow \text{Node3} \]

\[ 4-1=3 \]

\[ 3-1=2 \]

**Fig. 4.** A new node is prepended to a degenerated tree with 3 nodes.

# 4 Experimental Results

In this section, we present our experimental results of the algorithm \textit{det-k-CP} when applied to the benchmark suite provided by Gottlob et al. used in [1]. Note that we do not compare \textit{det-k-CP} with \textit{det-k-decomp} because these two algorithms aim at two different decomposition targets as mentioned before. However, this should not place an obstacle for us since the hypergraphs in the benchmarks are extracted from practical industrial constraint satisfaction problems. All the experiments are set up on an iMac computer having an Intel i7-3770 CPU, 3.40GHz, with 8 GB 1600 MHz DDR3 and running under macOS Sierra version 10.12.5. The algorithms are implemented in Java under JDK version 1.8.0.131.

Since the running platform of hardware for \textit{det-k-CP} will be multi-core processors, we choose 4, 8, and 16 for \( k \) as the number of nodes for the target degenerated decomposition tree. Because all the hypergraphs in the benchmark suite are decomposable by \textit{det-k-CP}, the experiment data to which we pay attention include execution time, the number of constraints on each node and the variance of the number of constraints of all nodes, and as well as the impact of different stochastic strategies on the experiment results. At current stage, we regard the variance of the number of constraints is an important target for parallel constraint solving. However, it will be easy to switch to variance of the weight measured by the amount of computation of each node if we can estimate the amount of computation of a constraint in a given constraint network in the future.

Table 1 shows experimental results for \textit{det-k-CP} of the benchmark suite from [1]. The symbols ○, △, and ■ denote the benchmark packages Daimler-Chrysler, Grid2D, and ISCAS89 as used in [1], respectively. The number of vertices and edges of each original instance is left out here due to lack of space. For the details on the benchmarks we refer to [1], Section 5.
Table 1: Experimental results for $det$-$k$-$CP$ of the benchmark suite from [1].

<table>
<thead>
<tr>
<th>Instance</th>
<th>$k = 4$</th>
<th></th>
<th></th>
<th></th>
<th>$k = 8$</th>
<th></th>
<th></th>
<th></th>
<th>$k = 16$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T$</td>
<td>Min</td>
<td>Max</td>
<td>$\sigma^2$</td>
<td>$T$</td>
<td>Min</td>
<td>Max</td>
<td>$\sigma^2$</td>
<td>$T$</td>
<td>Min</td>
</tr>
<tr>
<td><code>adder_75</code></td>
<td>75</td>
<td>94</td>
<td>201</td>
<td>1561</td>
<td>309</td>
<td>47</td>
<td>180</td>
<td>1832</td>
<td>3.7s</td>
<td>24</td>
</tr>
<tr>
<td><code>adder_99</code></td>
<td>9</td>
<td>94</td>
<td>173</td>
<td>1653</td>
<td>134</td>
<td>47</td>
<td>140</td>
<td>1734</td>
<td>3.9s</td>
<td>23</td>
</tr>
<tr>
<td><code>adder_75</code></td>
<td>10</td>
<td>94</td>
<td>208</td>
<td>1815</td>
<td>192</td>
<td>47</td>
<td>207</td>
<td>3201</td>
<td>3.6s</td>
<td>24</td>
</tr>
<tr>
<td><code>adder_99</code></td>
<td>75</td>
<td>124</td>
<td>262</td>
<td>2341</td>
<td>406</td>
<td>62</td>
<td>236</td>
<td>3062</td>
<td>5s</td>
<td>31</td>
</tr>
<tr>
<td><code>adder_99</code></td>
<td>10</td>
<td>124</td>
<td>261</td>
<td>4430</td>
<td>242</td>
<td>62</td>
<td>204</td>
<td>2098</td>
<td>12s</td>
<td>31</td>
</tr>
<tr>
<td><code>adder_99</code></td>
<td>14</td>
<td>124</td>
<td>272</td>
<td>3050</td>
<td>263</td>
<td>62</td>
<td>264</td>
<td>5008</td>
<td>5.9s</td>
<td>31</td>
</tr>
<tr>
<td><code>bridge_50</code></td>
<td>59</td>
<td>113</td>
<td>211</td>
<td>1323</td>
<td>277</td>
<td>57</td>
<td>269</td>
<td>2517</td>
<td>3s</td>
<td>29</td>
</tr>
<tr>
<td><code>bridge_50</code></td>
<td>7</td>
<td>113</td>
<td>221</td>
<td>2760</td>
<td>165</td>
<td>56</td>
<td>207</td>
<td>3160</td>
<td>3s</td>
<td>28</td>
</tr>
<tr>
<td><code>bridge_50</code></td>
<td>9</td>
<td>113</td>
<td>227</td>
<td>1830</td>
<td>207</td>
<td>57</td>
<td>219</td>
<td>2841</td>
<td>3.3s</td>
<td>29</td>
</tr>
<tr>
<td><code>bridge_75</code></td>
<td>82</td>
<td>170</td>
<td>314</td>
<td>2934</td>
<td>511</td>
<td>85</td>
<td>294</td>
<td>4068</td>
<td>7.8s</td>
<td>43</td>
</tr>
<tr>
<td><code>bridge_75</code></td>
<td>14</td>
<td>169</td>
<td>303</td>
<td>2729</td>
<td>639</td>
<td>84</td>
<td>347</td>
<td>7046</td>
<td>9s</td>
<td>42</td>
</tr>
<tr>
<td><code>bridge_75</code></td>
<td>17</td>
<td>170</td>
<td>336</td>
<td>4065</td>
<td>402</td>
<td>85</td>
<td>321</td>
<td>6796</td>
<td>8s</td>
<td>43</td>
</tr>
<tr>
<td><code>bridge_99</code></td>
<td>172</td>
<td>224</td>
<td>420</td>
<td>5168</td>
<td>13</td>
<td>112</td>
<td>376</td>
<td>7484</td>
<td>16.7s</td>
<td>56</td>
</tr>
<tr>
<td><code>bridge_99</code></td>
<td>38</td>
<td>223</td>
<td>435</td>
<td>6166</td>
<td>782</td>
<td>111</td>
<td>430</td>
<td>16265</td>
<td>14s</td>
<td>55</td>
</tr>
<tr>
<td><code>bridge_99</code></td>
<td>33</td>
<td>224</td>
<td>442</td>
<td>6727</td>
<td>737</td>
<td>112</td>
<td>424</td>
<td>11258</td>
<td>17s</td>
<td>56</td>
</tr>
<tr>
<td><code>NewSystem2</code></td>
<td>54</td>
<td>50</td>
<td>102</td>
<td>341</td>
<td>159</td>
<td>25</td>
<td>88</td>
<td>387</td>
<td>1.1s</td>
<td>53</td>
</tr>
<tr>
<td><code>NewSystem2</code></td>
<td>5</td>
<td>50</td>
<td>91</td>
<td>249</td>
<td>81</td>
<td>25</td>
<td>116</td>
<td>723</td>
<td>1.5</td>
<td>12</td>
</tr>
<tr>
<td><code>NewSystem2</code></td>
<td>3</td>
<td>50</td>
<td>102</td>
<td>341</td>
<td>72</td>
<td>25</td>
<td>111</td>
<td>729</td>
<td>0.7s</td>
<td>13</td>
</tr>
<tr>
<td><code>NewSystem3</code></td>
<td>84</td>
<td>70</td>
<td>133</td>
<td>343</td>
<td>268</td>
<td>35</td>
<td>134</td>
<td>1046</td>
<td>1.8</td>
<td>18</td>
</tr>
<tr>
<td><code>NewSystem3</code></td>
<td>5</td>
<td>69</td>
<td>145</td>
<td>1370</td>
<td>150</td>
<td>34</td>
<td>141</td>
<td>1824</td>
<td>3.9s</td>
<td>12</td>
</tr>
<tr>
<td><code>NewSystem3</code></td>
<td>7</td>
<td>70</td>
<td>149</td>
<td>899</td>
<td>115</td>
<td>35</td>
<td>141</td>
<td>1240</td>
<td>2m</td>
<td>18</td>
</tr>
<tr>
<td><code>NewSystem4</code></td>
<td>71</td>
<td>105</td>
<td>211</td>
<td>1467</td>
<td>391</td>
<td>53</td>
<td>185</td>
<td>1610</td>
<td>4.5s</td>
<td>27</td>
</tr>
<tr>
<td><code>NewSystem4</code></td>
<td>11</td>
<td>104</td>
<td>211</td>
<td>1629</td>
<td>328</td>
<td>52</td>
<td>235</td>
<td>5516</td>
<td>7.5s</td>
<td>26</td>
</tr>
<tr>
<td><code>NewSystem4</code></td>
<td>10</td>
<td>105</td>
<td>211</td>
<td>1467</td>
<td>286</td>
<td>53</td>
<td>219</td>
<td>3102</td>
<td>5s</td>
<td>27</td>
</tr>
<tr>
<td><code>grid2D_40</code></td>
<td>97</td>
<td>200</td>
<td>332</td>
<td>2630</td>
<td>777</td>
<td>100</td>
<td>311</td>
<td>4949</td>
<td>13s</td>
<td>50</td>
</tr>
<tr>
<td><code>grid2D_40</code></td>
<td>28</td>
<td>200</td>
<td>335</td>
<td>2755</td>
<td>695</td>
<td>100</td>
<td>320</td>
<td>8010</td>
<td>15s</td>
<td>50</td>
</tr>
<tr>
<td><code>grid2D_40</code></td>
<td>31</td>
<td>200</td>
<td>338</td>
<td>3079</td>
<td>592</td>
<td>100</td>
<td>318</td>
<td>5297</td>
<td>14s</td>
<td>50</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>486</td>
<td>703</td>
<td>1149</td>
<td>29669</td>
<td>7.8s</td>
<td>352</td>
<td>1093</td>
<td>62397</td>
<td>3.7m</td>
<td>176</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>277</td>
<td>703</td>
<td>1192</td>
<td>35509</td>
<td>9.4s</td>
<td>351</td>
<td>1149</td>
<td>66461</td>
<td>3.7m</td>
<td>175</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>298</td>
<td>703</td>
<td>1186</td>
<td>35874</td>
<td>7.9s</td>
<td>352</td>
<td>1131</td>
<td>68936</td>
<td>3.5m</td>
<td>176</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>593</td>
<td>710</td>
<td>204</td>
<td>1299</td>
<td>313</td>
<td>53</td>
<td>195</td>
<td>2286</td>
<td>3s</td>
<td>27</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>8</td>
<td>106</td>
<td>211</td>
<td>2756</td>
<td>225</td>
<td>53</td>
<td>217</td>
<td>2776</td>
<td>3.2s</td>
<td>26</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>6.9</td>
<td>106</td>
<td>204</td>
<td>1298</td>
<td>170</td>
<td>53</td>
<td>205</td>
<td>2724</td>
<td>3.5s</td>
<td>27</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>1494</td>
<td>78</td>
<td>164</td>
<td>335</td>
<td>3859</td>
<td>523</td>
<td>82</td>
<td>291</td>
<td>4489</td>
<td>6.6s</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>1494</td>
<td>15</td>
<td>163</td>
<td>350</td>
<td>8079</td>
<td>380</td>
<td>81</td>
<td>345</td>
<td>11411</td>
<td>7.6s</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>1494</td>
<td>17.5</td>
<td>164</td>
<td>348</td>
<td>4832</td>
<td>340</td>
<td>80</td>
<td>332</td>
<td>7630</td>
<td>7.6s</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>5378</td>
<td>507</td>
<td>740</td>
<td>1461</td>
<td>70147</td>
<td>5.8s</td>
<td>370</td>
<td>1338</td>
<td>101431</td>
<td>3m</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>5378</td>
<td>269</td>
<td>739</td>
<td>1347</td>
<td>57433</td>
<td>13.2s</td>
<td>369</td>
<td>1642</td>
<td>153866</td>
<td>5.3m</td>
</tr>
<tr>
<td><code>grid2D_75</code></td>
<td>5378</td>
<td>183</td>
<td>740</td>
<td>1461</td>
<td>70148</td>
<td>6.8s</td>
<td>370</td>
<td>1614</td>
<td>176602</td>
<td>3.8m</td>
</tr>
</tbody>
</table>
In Table 1, T represents execution time in ms (unless other specified), and Min, Max stands for the minimal and maximal number of constraints among all nodes, respectively. $\sigma^2$ denotes the variance of the number of constraints of all nodes. There are three different background colours for each instance which indicate three types of heuristics for the swap procedure, which are permutation, random exchange, and switching nodes according to number of constraints in turn.

Though Table 1 omits some instances from the benchmark suite due to lack of space, all the instances are decomposable by $det-k$-$CP$. Besides, the execution time of instances are affordable even for the largest instance $s5378$, which has 2958 constraints and 2993 variables. Nevertheless, for large scale instances, such as $s5378$, $grid2d_75$ etc. the algorithm $det-k$-$CP$ does not achieve one principal goal of decomposition that each node has relativity balanced workload distribution. The maximal node contains 1220 constraints in overall 2958 constraints which is impossible to be solved by any single thread constraint solver for a practical problem based on our user experience of constraint solver.

Since it is not urgent to solve a small constraint network in parallel, which consist of less than 200 constraints, because these problems are usually solvable for mainstream constraint solvers, we do not include in small instances such as $adder_{15}, adder_{25}$ etc. from the benchmark suite.

The decomposition results for the medium scale instances, such as $adder_{75}, bridge_{75}, NewSystem3, NewSystem4, and S953$ with number of constraints ranging from around 400 to 700, might be suitable for parallel constraint solving. Because each node for these instances can be solved within a reasonable time since the maximum number of constraints of these instances is slightly greater than 200 such as 201 in $adder_{75}$ and 210 $NewSystem4$, and the total number of constraints for $adder_{75}$ and $NewSystem4$ are 677 and 418, respectively. In addition to the relatively small variance of these instance indicates the number of constraints for these instances is not spread out from their mean. As we mentioned before, whether or not a given constraint network can be solved within a reasonable time does not only depend on the number of constraints, but also on the time complexity of propagators employed by the constraint solver, the number of variables of the constraint network and the size of domain of the variables.

The results in Table 1 show that the method of heuristics has significant impact on the results of decomposition. In most of cases, the exchange of nodes by permutation order (first lines, i.e. white background color) gets smaller variance, but there are exceptions, such as $bridge_{75}, NewSystem4$.

In summary, we can conclude that $det-k$-$CP$ can decompose a given constraint network within a reasonable execution time except for large instance (e.g., $s5378$ with 2958 constraints and 2993 variables) and a big $k$ (e.g., $k = 16$). For the application of parallel constraint solving, the algorithm can be applied to medium scale constraints network with the number of constraints ranging from around 400 to 700.
5 Conclusion and Future Work

We have presented the new algorithm det-k-CP to construct a degenerate decomposition tree for parallel constraint solving, and we have also evaluated det-k-CP by a benchmark suite from previous research det-k-decomp. Our results have shown that det-k-CP is appropriate for evenly partitioning a constraint network with around 400 to 700 constraints for a distribution on a given number of parallel constraint solving cores. However, we believe that there is a lot potential to improve det-k-CP. For example, the algorithm should take into consideration an estimate of the amount/complexity of computation for the constraints so that we could choose eg., constraint with low computation requirement when adding constraints to another node in Algorithm 3. Furthermore, local search methods such as tabu search can be employed to replace the existing stochastic strategies in order to obtain more optimized decomposition result. Besides, for distributing workload more fairly, the constraints in which the variables of these constrains are covered by the variables of the node can be added into the node, so that the decomposition tree would not be destroyed. Finally, the key indicator of the value of this research depends on whether we can obtain speedup or even superlinear speedup when using det-k-CP for parallel constraint solving, which is to be researched.

Acknowledgments We would like to express our special thanks to Georg Gottlob and Wolfgang Fischl for their source code of det-k-decomp, especially the benchmark suite for hypertree decomposition.

References

Declarative Systems
An Approach for Representing Answer Sets in Natural Language

Min Fang and Hans Tompits
Institute of Information Systems 184/3, Technische Universität Wien, Favoritenstraße 9-11, 1040 Vienna, Austria
{fang,tompits}@kr.tuwien.ac.at

Abstract. In recent years, different methods for supporting the development of answer-set programming (ASP) code have been introduced. During such a development process, often it would be desirable to have a natural-language representation of answer sets, e.g., when dealing with domain experts unfamiliar with ASP. In this paper, we address this point and provide an approach for such a representation, defined in terms of a controlled natural language (CNL), which in turn relies on the annotation language LANA for the specification of meta-information for answer-set programs. Our approach has been implemented as an Eclipse plug-in for SeaLion, a dedicated IDE for ASP.

1 Introduction

In recent years, the question of providing methods and tools for supporting the development of answer-set programs has received increased attention in the literature [7,2]. A feature often desirable in developing answer-set programs is to have a natural-language representation of the output of such programs, i.e., of answer sets which represent the solutions of encoded problems. Such a representation would be useful, e.g., when dealing with domain experts who are unfamiliar with answer-set programming (ASP).

In this paper, we address this point and provide an approach for such a natural-language representation of answer sets. However, for realising a method like this, it should be clear that information other than the pure ASP code is required. Somehow, a mechanism which allows to describe what certain ASP predicates “mean” is required. Such a feature is provided by the annotation language LANA (“Language for ANnotating Answer-set programs”) [3] that augments answer-set programs with additional meta-information. LANA offers language constructs for declaring predicates, type information, input and output signatures, etc. These additional annotations can then be interpreted by independent tools to support the development process, e.g., to run specialised test cases, to automatically generate a program documentation, and so on. In fact, LANA and some of the tools using it have already been integrated in SeaLion [2], a dedicated integrated development environment (IDE) for ASP implemented as an Eclipse plug-in.

Our goal was to create a tool which uses the information provided by LANA and which generates a more human-readable interpretation of answer sets. For this purpose, we used the user-specified atom descriptions (given with the LANA @atom annotations)
and built natural-language-like interpretations for the answer sets according to the problem instance. In order to parse the atom descriptions properly and deduce syntactic information from them, our approach uses descriptions which are formulated in a \textit{controlled natural language} (CNL). Our tool, realised as an Eclipse plug-in alongside SeaLion, allows a knowledge engineer to make further ad-hoc changes to a generated interpretation and eventually export it, e.g., in PDF format. This document can then be easily forwarded to a domain expert for further consideration.

Following Kuhn [14], a CNL is a constructed language resembling a natural language but being more restrictive concerning lexicon, syntax and/or semantics while preserving most of its natural properties. In effect, it is essentially a formal language and can therefore usually be specified by known mechanisms of specification of formal languages, e.g., in terms of a grammar. The degree of resemblance of a CNL to a natural language will vary depending on the purpose of the CNL and the (technical) background of the designated users of this language. A CNL is often used as a bridge between a highly ambiguous natural language and a less human-readable formal language, like, e.g., predicate logic formulas or a programming language. This type of CNL is referred to as \textit{computer-oriented CNL} and is usually employed in applications where some sort of semi-automatic or automatic processing of user input is necessary in order to translate it into a more formal representation [20].

Recent instances of general-purpose (i.e., not restricted to the vocabulary of a specific domain), English-based CNLs are, e.g., Attempto Controlled English (ACE) [9,8] and PENG/PENG Light [18,23,19].

Our approach is not the first to employ CNL techniques in the context of answer-set programming. To wit, Erdem and Yeniterzi [6] developed BioQuery-CNL, a domain-specific CNL which is used to express biomedical queries over predefined ontologies. These queries are then in turn translated into answer-set programs, using the BioQuery-ASP system [4,5]. Likewise, Schwitter [21,22] as well as Guy and Schwitter [11] discuss methods to solve search problems by representing them into a CNL and processing these representations by translations to ASP.

The paper is organised as follows: We first provide some background on the ASP annotation language $\text{LANA}$ in Section 2. Then, in Section 3, we describe our proposed CNL for obtaining a natural-language representation of answer sets and Section 4 details the actual translation. Section 5 briefly discusses the implemented tool and the paper closes with some general remarks in Section 6.

2 Background

We assume the reader familiar with the basics of answer-set programming (ASP) [1]. Briefly, an \textit{answer-set program} consists of a set of rules of form

\[
a : - b_1, \ldots, b_n, \text{ not } c_1, \ldots, \text{ not } c_m,
\]

where $a, b_1, \ldots, b_n, c_1, \ldots, c_m$ are atoms over a first-order vocabulary and \text{not} denotes \textit{default negation}. The semantics of such a program $P$ is given in terms of \textit{answer sets}, which are defined as those models $I$ of $P$ which satisfy a fixed-point condition using
the reduct of $P$ relative to $I$ [10]. Prominent solvers for computing answer sets are, e.g., clasp [17] and DLV [15].

The annotation language LANA was introduced by De Vos et al. [3] with the purpose of defining a standardised apparatus for specifying meta-information for answer-set programs. The formalism of LANA is reminiscent of Java annotations, with the @ symbol preceding each keyword. Just as some Java annotations (e.g., annotations for Javadoc), annotations in LANA have the form of an ASP comment, thus not altering the semantics

Fig. 1. An illustration of LANA annotations in a program file.
of the program that it is documenting. If an ASP solver—in particular its grounding component—supports block-comments, these can be used instead of the single-line comment marker “%”.¹

LANA offers an array of different annotations, of which we only mention a few. E.g., @block can be used to group certain rules together, i.e., deriving a more finely-grained structure within a code file. Each block can then declare predicates using the @atom annotation, and define its input and output signature with @input and @output, respectively. Other annotations include @assert, @precon, @postcon, @always, and @never, which define logical conditions for the answer sets and can be used for testing purposes.

For our use case, we will mainly focus on two of the annotations: @atom and @output. Figure 1 depicts some annotations for illustrating the usage of these two annotations. As can be seen in this example, @atom annotations are made up of a predicate name, a comma-separated list of its arguments given as variables (i.e., an identifier starting with an uppercase character) wrapped in round brackets, and an (optional) sentence describing the semantics of the corresponding predicate. LANA itself does not define any restrictions for this short description as it is only considered a comment, i.e., it is used as an informal documentation of the associated predicate. For our purposes, however, it is necessary to impose a certain well-defined structure on them.

The @input and @output annotations list the predicates that correspond to the input and output of a problem instance, respectively. Input predicates are usually required to encode the problem instance, while output predicates are relevant for the solution of a particular problem instance. The predicates are given with their name and their respective arity since it is possible to have predicates with the same name, in which case they are uniquely identifiable by the combination of their name and arity.

While in LANA all these annotations are not mandatory, we presume for our purposes that they are specified by the user, otherwise it would be impossible to generate a textual, human-readable interpretation of answer sets that would reflect their intended meaning.

3 A Controlled Natural Language for LANA Atom Descriptions

We first define the controlled natural language that determines the form of atom descriptions associated with an @atom annotation. The CNL that we are envisaging for the LANA atom descriptions cannot be domain-specific since we are not restricting the domain of answer-set programs that can use the LANA annotations (e.g., it is possible to encode a combinatoric puzzle as much as a scheduling problem). Hence, it is impossible to predefine a set lexicon for our CNL as far as content words are concerned. We can, however, presume a fixed set of function words (e.g., prepositions) which we incorporate directly into the grammar.

¹ E.g., if one uses gringo as grounding component. An additional “*” is then added to the block-comment marker “%*” in order to distinguish LANA annotations from normal block comments. Hence, LANA annotations are wrapped in “%*%” and “*%” blocks. We will assume this syntax for the examples below.
3.1 Preliminary Considerations

Having an open, utterly unconstrained lexicon is unusual for both domain-specific and general-purpose CNLs. The former would normally have a relatively small lexicon tailored to its field of application. For the latter, a dynamic lexicon is more typical, i.e., it would have a compact predefined lexicon with the most frequently used words of the associated natural language, with the additional possibility of the user specifying new entries for the lexicon as they go along. Even though a dynamic lexicon could be used for our purposes, it would complicate the user experience unnecessarily since the user may have to specify further information about the new lexemes (e.g., identifying their part-of-speech).

The real restriction of our CNL therefore lies in the syntactic structure of its sentences. Instead of a hundred odd rules (e.g., ACE Codeco has 164 grammar rules [14]), we constrain our grammar to a very small number of rules. This is possible because our CNL is, after all, not a general-purpose CNL. We know its application context (stated below an @atom annotation describing the meaning of this atom) and its purpose (generating human-readable interpretations of answer sets) despite being uninformed about the semantics of its individual words and the domain of the program. By imposing a rigid structure on the sentences we will gain enough structural information about these sentences and their constituents that we can use for the generation of new sentences, i.e., textual interpretations, according to the instantiated atoms of each answer set.

This highly syntactic approach is very much different from some of the more well-known CNLs, which usually map the sentences to some kind of formal logic representation (e.g., to discourse representation structures [9,23]). We are able to bypass this step since we are not so much interested in the exact semantics of the sentences but rather in the structural relations of its constituents. Indeed, we use these constituents as “boilerplates” for the generation of our own sentences, which make up the textual interpretation of answer sets.

Concerning the grammatical structure that we are defining for LANA atom descriptions, the first general restriction that we make is that each atom description is made up of one sentence only. This way we can avoid difficult problems such as anaphora resolution. The structure of these sentences must be basic and unambiguously parseable but at the same time flexible enough to allow the description of many possible predicates. We essentially allow three sentence types in our CNL, where the defining characteristics are linked to the main verb of the sentence and its argument structure, i.e., the number and the kind of arguments it combines with. In traditional grammar, this verb property is referred to as *valency* or *valence* of verbs (cf., e.g., Van Valin [25]).

3.2 Syntactical Structure of the CNL

We now detail the exact constituents each sentence type of our CNL allows and what each constituent is made up of. Instead of giving a general formal definition of the syntax of our CNL, which we omit here due to space reasons, we focus on providing illustrative explanations what valid phrases and sentences our language admits.

---

2 PENG [23] avoids this problem by disallowing personal pronouns, which are often contextually ambiguous, and using explicit variable references instead.
For our purposes it is sufficient that we assume a fixed set of (more or less) function words that our parser recognises and which help to structure the input. This set contains the following words: *is*, *are*, *be*, *has*, *have*, *do*, *does*, *must*, *can*, *cannot*, *an*, *a*, *A*, *The*, *There*, *maximally*, *minimally*, *at least*, *at most*, *more*, *or less*, *or fewer*, *to*, *with*, *of*, *for*, *as*, *by*, *at*, *in*, *on*, and *not*. The prepositions are especially important since they help the parser to unambiguously place prepositional phrases in the sentence (and thus identify where a noun phrase ends). In the examples below, instances of these words are underlined.

**Central Language Constituents.** We first list central language elements of our CNL.

**Variables.** We have predefined variables of a specific form: They must start with an upper-case character, followed by other upper-case characters or numbers 0 to 9. Note that this definition is more restricted than the usual ASP definition for variables. There is, however, one exception to this simple rule: The symbol “A” cannot be used to denote a variable since it is a reserved function word (referring to an indefinite article at the beginning of a sentence). In the subsequent examples variables are printed in bold.

**Noun Phrases (NPs).** First of all, we distinguish between noun phrases (NPs) which contain variables identical to those listed in the \texttt{@atom} signature and noun phrases which do not. \textit{Variable-free} NPs (NP\textsubscript{var}) are made up of an arbitrary number of words that are not in the set of function words (only the first one may be a definite or indefinite article). Since the words may be chosen at random by the user, they do not necessarily have to be real noun phrases, they can in fact be adjectival phrases (AdjPs) too (however, we will keep the label NP because the labelling has little importance). Because there is almost no restriction within a variable-free NP and also because they can be difficult to parse depending on the surrounding constituents, their availability in our grammar is very much restricted (details given further down). Typical NP\textsubscript{var} instances are the following:

1. a. a ship;
   b. battle ships;
   c. the best project leader.

\textit{Variable-containing} NPs (NP\textsubscript{var}) are in their most minimalistic form made up of the variable only. Optionally, they may then contain a definite or an indefinite article. In their most elaborate form they may additionally contain arbitrary words between the article and the variable (basically describing the semantics of the variable) and a post-variable modifier. Possible modifiers are \textit{maximally}, \textit{minimally}, \textit{at least}, \textit{at most}, \textit{more}, \textit{or less}, \textit{or fewer}. Hence, examples for variable-containing NPs typically look like the following:

2. a. \(E\);
   b. an employee \(E\);
   c. the project leader \(L\);
   d. a proficiency level \(V\) \textit{at least}.
However, sometimes the position of the variable and the describing noun may be reversed (we use \texttt{NP}_{\text{rev}} to refer to this special case and \texttt{NP}_{\text{com}} to the complementary case). This is usually the case if the type of the variable is integer. In these cases, the variable will be instantiated with integer constants in corresponding answer sets. A modifier is also possible in this type of NP after the last word. Note that there must be at least one word following the variable in order to determine the concept described or quantified by the variable. Examples of this type of NP are the following:

(3) a. N employees;
   b. N employees or more.

As an additional feature, we enable the user to specify the plural suffix of one-word nouns so that this information can also be automatically used in the interpretation generation process. For instance, consider the following example:

(4) a. N employee(s);
   b. project(s) P.

Hence, if desired, one can add the plural “-s” to a noun in brackets. This addition will especially make sense in \texttt{NP}_{\text{rev}} cases. It will then be used in generated sentences where it is clear that a plural form of the marked noun will be required. If this additional information is not given by the user, the generation process will simply stick to the form given.

\textit{Prepositional Phrases (PPs).} Prepositional phrases (PPs) are a combination of a preposition and an NP, where the NP can be variable-free or variable-containing. Hence, a PP is either \texttt{PP}_{\text{var}} or \texttt{PP}_{\text{com}} depending on the NP that they contain. Prepositions are comprised of the words \textit{to}, \textit{with}, \textit{of}, \textit{for}, \textit{as}, \textit{by}, \textit{at}, \textit{in}, and \textit{on}. Typical examples are the following:

(5) a. by E;
   b. on project P;
   c. for N employees at most;
   d. as the project leader.

\textbf{Sentence Types.} We next describe our three categories of sentences. Note that we use a classification of verbs into \textit{intransitive} (valency 1), \textit{transitive} (valency 2), and \textit{ditransitive} (valency 3) verbs.\footnote{Verbs with valency 0 do not have their own term since there is only a small number of them (predominantly weather verbs). We disregard them for our considerations.} Intransitive verbs come with a subject only, transitive verbs must have a subject and a direct object (DO), while ditransitive verbs combine a subject, a DO, and an indirect object (IO), which is normally realised as an oblique object (i.e., it is inside a preposition phrase) in English.
Sentences with an Intransitive Verb. The main characteristic of sentences with an intransitive verb is the fact that the verb only requires one argument, which is in subject position. Hence, we allow for this type of sentences only one NP, which furthermore must contain a variable. Consequently, a corresponding atom would have arity one or more. If it does indeed have a higher arity, the remaining variables must be encoded into PPs. Schematically, such sentences have the following form:\footnote{In what follows, we use superscripts to denote the valency of the associated verb and the symbols “?” and “*” refer to BNF syntax customs (i.e., standing for options and possible repetitions, respectively).}

\begin{equation}
\text{NP}\text{?}^\text{ Modal?} \ V^{1} \ \text{PP*}.
\end{equation}

As indicated here, we also allow the addition of a modal verb before the main verb, which is either can or must. Moreover, the negated version of an acceptable sentence is also acceptable (the accepted negation strategy depends on the verb configuration in the sentence). Some examples are the following:

\begin{enumerate}
\item [a.] [Employee \textbf{E}] works.
\item [b.] [Employee \textbf{E}] does not work.
\item [c.] [N employee(s)] cannot work.
\item [d.] [Employee \textbf{E1}] works [\textbf{with} employee \textbf{E2}] [\textbf{on} project \textbf{P}].
\item [e.] [N employees] must work [\textbf{on} project \textbf{P}].
\item [f.] [Employee \textbf{E}] must not work [\textbf{on} project \textbf{P}].
\end{enumerate}

Note again that the first constituent must be an NP containing a variable. A variable-free NP in this position will lead to a parsing error.

Sentences with a Transitive Verb. Since a transitive verb requires two arguments, one in subject position and one in object position, this sentence type requires two NPs, both of which have to contain a variable corresponding to the associated atom signature. For the remaining variables declared by the atom, PPs should be used, which would wrap variable-containing NPs. Schematically, this kind of sentences have the following form:

\begin{equation}
\text{NP}\text{?}^\text{ Modal?} \ V^{2/3} \ \text{NP}\text{?}^\text{ PP*}.
\end{equation}

Again, an additional modal verb just before the main verb is possible as well as the negated version of allowed sentences, like in the following examples:

\begin{enumerate}
\item [a.] [Employee \textbf{E}] heads [project \textbf{P}].
\item [b.] [Employee \textbf{E}] does not head [project \textbf{P}].
\item [c.] [Employee \textbf{E}] heads [project \textbf{P}] [\textbf{with} skill \textbf{S}].
\item [d.] [A ship \textbf{T}] must occupy [position \textbf{X}] [\textbf{on} day \textbf{D}].
\item [e.] [A ship \textbf{T}] must not occupy [position \textbf{X}].
\end{enumerate}
Sentences with a Copula. In linguistics, the term *copula* is used to refer to a certain kind of linking verb that connects the subject of a sentence to the subject complement, the so-called *predicative*. Informally, a copula can be understood to be similar to the mathematical “equals” sign, equating its left part to its right part. The main copula in English is the verb *to be*. Sentences with a copula allow the user more freedom with certain constituents. Contrary to the other types, it is possible to have both a variable-containing and a variable-free NP. In the first case, the copula can be followed by either an NP\textsubscript{var} or an NP\textsuperscript{var} and optional PPs. Schematically, such sentences look as follows:5

\[(10) \text{NP}\textsubscript{var} \text{Modal? V}^{\text{cop}} \text{NP PP*}.\]

In the latter case, however, we have to make sure that the sentence still contains at least one variable. Since the first NP does not contain one, there are only two other options: Either there is an NP\textsubscript{var} occurring immediately after the copula, in which case there are no further PPs necessary, or there is only an NP\textsuperscript{var} or no NP at all after the copula, in which case there must be at least one PP at the end of the sentence that contains an NP\textsubscript{var}. This is schematically summarised as follows:

\[(11) \begin{align*}
    & \text{a. NP}\textsuperscript{var} \text{Modal? V}^{\text{cop}} \text{NP\textsuperscript{var} PP*}; \\
    & \text{b. NP}\textsuperscript{var} \text{Modal? V}^{\text{cop}} \text{NP\textsuperscript{var} PP*}. 
\end{align*}\]

Let us illustrate schemata (10) and (11) with examples (12) and (13) below, respectively:

\[(12) \begin{align*}
    & \text{a. [Employee E] is [a project leader].} \\
    & \text{b. [Employee E] is [project leader] [for project P].} \\
    & \text{c. [Employee E] must not be [project leader] [for project P].} 
\end{align*}\]

\[(13) \begin{align*}
    & \text{a. [An employee] is [project leader L] [for project P].} \\
    & \text{b. [There] are [N project leader(s)] [on project P].} \\
    & \text{c. [The Planes] are [in airport X] [with identifier Y].} \\
    & \text{d. [Planes] cannot be [in airport X].} \\
    & \text{e. [There] must be [planes] [in airport X] [at day T].} \\
    & \text{f. [There] must not be [project leaders] [for project P].} 
\end{align*}\]

Observe that by allowing variable-free NPs in the subject position, existential constructions with there become available.

4 Interpreting Answer Sets

Based on the @atom annotations provided by the LANA language in an ASP code and the natural-language descriptions of the atoms conforming to the CNL of the previous section, we can generate adapted sentences that make answer sets more readable.

\footnote{By NP we denote the union of NP\textsubscript{var} and NP\textsuperscript{var}. Similarly, PP denotes the union of PP\textsubscript{var} and PP\textsuperscript{var}.}
Basic Generation. Let us revisit the example given in Figure 1. Suppose that this code forms part of a program that takes facts as input which encode information about employees and projects and is supposed to output answer sets that can be interpreted as possible project assignments. Following the ASP “guess and check” methodology, there would be a guessing part in the program generating candidate solutions and some rules and constraints which filter out those candidates which are not proper solutions of the problem instance (e.g., if certain employees are not available because they are on holidays).

A domain expert is usually only interested in those predicates which encode the problem solution. Hence, using the @output annotation of LANA, one can mark the relevant atoms that should be considered for the interpretation-generation process. If all predicates are relevant, this annotation may be left out since the default assumption is then that every predicate should be used.

Consider again the example from Figure 1 and the answer set of the program mentioned therein depicted in Figure 2, which has been stripped off of irrelevant atoms, i.e., atoms that were not marked as @output in the program code.

Using the information provided by the @atom annotation and the atom description directly below the annotation, we can generate interpretation sentences corresponding to this answer set by simply replacing each variable place holder in the sentence with the appropriate constant in the answer set atom. This basic method will generate a new sentence for each instantiated atom in the given answer set. The result of this naive procedure is as follows:

Employee boris works on project p1 with skill design. (1)
Employee boris works on project p1 with skill marketing. (2)
Employee boris works on project p2 with skill design. (3)
Employee boris works on project p2 with skill marketing. (4)
Employee boris works on project p2 with skill planning. (5)
Employee hans works on project p1 with skill modelling. (6)
Employee lisa works on project p2 with skill design. (7)
Employee lisa works on project p2 with skill modelling. (8)
Employee lisa works on project p2 with skill planning. (9)
Employee peter works on project p1 with skill coding. (10)
Employee peter works on project p1 with skill planning. (11)
“Contracting” the Sentences. Up to now we have not really employed the syntactic information provided by the parser for the atom descriptions. All we had used so far was the mapping between the variables in the sentences and the variables given by the @atom signatures.

Let us now consider sentences (1) to (5) of the above interpretation involving employee Boris. We will illustrate on them how one may make use of the syntactic information for “contracting” the sentences.

As is easily noticeable, these sentences offer redundant information. This property was to be expected since we only have one boilerplate sentence for each atom, which, however, usually has many instantiations in one answer set. Thus, placing these sentences without modification next to each other is most probably going to repeat already known information. This repetition is going to be especially severe if the number of instantiations of an atom is very large. Consequently, we have to look for procedures to systematically condense the information contained in the basic sentences so that they may expressed more concisely.

One Varying Element. The first approach is to focus on sentences which differ only in one variable. Consider sentences (1) and (2) on the one hand and (3) to (5) on the other hand. Within each respective group, the sentences differ only in the last variable, viz. concerning the variable denoting the skill which is assigned to an employee for a certain project. The best way to summarise the sentences of this kind of a group is to coordinate the varying elements, i.e., the differing skills, while maintaining the constituents that the sentences share, i.e., the employee and the project. In the first group, the varying constituents are with skill design and with skill marketing. Knowing that the constituents have the same PP shells, we can implement the rule that the NPs within the PPs can be coordinated, the noun phrase between the preposition and the term being pluralised if possible and only displayed in the first conjunct, and finally the preposition added in the front. This simple rule results in the contracted phrase with skills design and marketing. If we proceed as described, we are able to cut down the
number of sentences as follows:

Employee boris works on project p1 with skills design and marketing. \( (17) \)
Employee boris works on project p2 with skills design, marketing, and planning. \( (18) \)

Note that whilst the noun skill is used in its singular form in the original sentences (1) to (5), here the plural form is employed. Recall, as pointed out in Section 3.2, the information what the plural form looks like and on which word it has to appear can be specified in NP constituents in the atom description. In instances where the varying elements are grouped into one coordination phrase, it is usually quite safe to use the plural form (if available).

In general, this contraction step can be performed whenever there is a group of sentences where there is one element (i.e., an instantiated variable) varying across the board, while all remaining elements stay the same in all sentences of the group. The preliminary grouping process is crucial for the output: depending on how the sentences are grouped, the varying element may change. In the example above, the default grouping was applied so that the the varying variable was the last variable in the sentence. However, a different grouping may be possible, where the varying variable is, e.g., the first one in the sentence. In this case, we coordinate the subjects and end up with the verb morphologically agreeing with the coordinated subjects, i.e., it shows plural morphology. A possible sentence that would reflect this grouping is the following:

Employee hans and sarah work on project p1 with skill modelling.

Finally, if we take the complete answer set from Figure 2 and apply the described contracting process (with the default grouping/sorting order), we get the following condensed version:

Employee boris works on project p1 with skills design and marketing.
Employee boris works on project p2 with skills design, marketing, and planning.
Employee hans works on project p1 with skill modelling.
Employee lisa works on project p2 with skills design, modelling, and planning.
Employee peter works on project p1 with skills coding and planning.
Employee sarah works on project p1 with skills coding and modelling.
Employee sarah works on project p2 with skill modelling.
Employee sarah is project leader for project p1.
Employee lisa is project leader for project p2.

As we can see, taking this measure reduces the number of sentences of the original interpretation from 16 to 9.
One Identical Element. After the first contraction step, there may still be some redundancy left, though. In order to address this, we use the topic-rheme dichotomy (cf., e.g., Halliday [12]) to characterise the information structure of a certain kind of redundancy: While the topic (“what is being talked about”) stays the same throughout a group of sentences, the rheme (“what is being said about the topic”) changes in each sentence. Hence, in sentences (17) and (18) above, the topic is boris and the rheme is his job assignments on the two projects. In most English sentences the topic corresponds to the subject of the sentence.

In the described configuration, a second contraction step can be performed thus: The sentences in such a group can be coordinated, with the common topic functioning as a common subject for the coordination. Hence, if we condense sentences (17) and (18) accordingly, we obtain the following:

Employee boris works on project p1 with skills design and marketing and on project p2 with skills design, marketing, and planning.

In this example we have taken the constituents that make up the varying rheme and coordinated them. The coordinated phrases can then be added to the fixed topic and the associated verb. By applying this contraction to the whole answer set, we can lower the number of sentences to 7.

By using the structural information provided by the grammar and ultimately by the parse tree, we are able to identify the borders of constituents easily, which allows us to insert newly generated phrases at the appropriate spots. Also, without the syntactic tagging we would not be able to distinguish constituents from non-constituents. This very basic identification is required in order to specify rules on similar constituents, i.e., constituent boilerplates where only the term varies.

5 The Eclipse Plug-in

We have implemented our tool as an Eclipse View, called “LANA Interpretation View”, to be used in connection with the IDE SeaLion for ASP [2]. Using Eclipse’s built-in repositioning feature, one can have both the Interpretation View of SeaLion and the LANA Interpretation View side-by-side, as shown in Figure 3. This way, the user can choose an answer set in the Interpretation View and directly see the generated interpretation for it.

The LANA Interpretation View is divided into two main parts: the left part showing the instantiations of the atoms marked as output predicates, and the right part providing an editor with the generated textual interpretation, which can be edited and exported.

On the very left there is a list widget containing atoms, which gets refreshed according to the selection in SeaLion’s Interpretation View. This widget displays all atoms which are listed in the @output annotation, together with the name of the block that they belong to (annotated with @block in the code). The table of terms next to the list changes its values according to the selection in the list. This table simply displays all instantiations of the selected atom, allowing the user to get an overview of the terms and sort them according to their needs.
As already pointed out previously, the sorting order may change the generated interpretation since it determines the grouping algorithm, which in turn influences the output of the “contraction” steps. A particular sorting order is given in Figure 4.

As one can see, the values in the table are sorted by project first, then by skills. We have marked the groups with orange rectangles, in which one term varies while all other terms stay stable across the instantiated atom in the group. This is relevant for the “one varying element” approach as described in Section 4. Since the employee terms are the “least sorted” element and, thus, the one varying element, they will be coordinated to reduce the number of sentences.

The other half of the view shows a simple text editor that displays the generated textual interpretation and another text widget pointing to problems encountered by the parser (cf. Figure 5). The editor allows the user to make ad-hoc changes to the text if desired. The buttons on the right implement the export function.

The checkbox Condensed indicates whether the second contraction step has been applied. Checking the box will lead to a recomputation and the result is shown in the editor immediately afterwards. The Reparse Code button forces the system to reparse
the code files that were specified for this corresponding program. This is useful when the feedback text widget indicates parsing errors in particular atom descriptions. The user is advised to inspect the sentences and check whether they form an acceptable sentence in our CNL. Once all corrections are made, a reparse should be performed.

6 Conclusion

In this paper, we presented a controlled-natural language (CNL) approach for generating interpretations for answer sets in which the user can specify meta-information about the predicates used in an answer-set program. The descriptions in the LANA annotations are restricted accorded to the CNL we defined.

Complementary to our work are approaches which provide justifications for the inclusion or non-inclusion of ground atoms in answer sets [16] or giving explanations why an answer-set program has no answer sets at all [24]. On the other hand, the problem setting we studied is similar to work in natural language processing for generating text out of structured data [13]. Conversely, a platform for building systems that allow to translate text to different formal languages has been studied by Nguyen et al. [26].

Besides a natural-language interpretation of answer sets as discussed here, in ongoing work we currently investigate the question of translating answer-set programs themselves into natural language. However, this task requires a more dedicated syntactic and semantic analysis of user-specified sentences.

References

Techniques for Efficient Lazy-Grounding ASP Solving

Lorenz Leutgeb and Antonius Weinzierl

1 Knowledge-Based Systems Group
Institute of Information Systems, TU Wien
Vienna, Austria
2 Helsinki Institute for Information Technology HIIT
Department of Computer Science, Aalto University
Espoo, Finland
lorenz@leutgeb.xyz weinzierl@kr.tuwien.ac.at

Abstract. Answer-Set Programming (ASP) is a well-known and expressive logic programming paradigm based on efficient solvers. State-of-the-art ASP solvers require the ASP program to be variable-free, they thus ground the program upfront at the cost of a potential exponential explosion of the space required. Lazy-grounding, where solving and grounding are interleaved, circumvents this grounding bottleneck, but the resulting solvers lack many important search techniques and optimizations. The recently introduced ASP solver Alpha combines lazy-grounding with conflict-driven nogood learning (CDNL), a core technique of efficient ASP solving. This work presents how techniques for efficient propagation can be lifted to the lazy-grounding setting. The Alpha solver and its components are presented and detailed benchmarks comparing Alpha to other ASP solvers demonstrate the feasibility of this approach.

1 Introduction

Answer-Set Programming (ASP) is an expressive logic programming paradigm where non-monotonic rules are used to formalize problem descriptions. The semantics of such rules are given in terms of answer sets, which represent solutions to the specified problem (see [4] for a detailed introduction). For example the following rules encode that for nodes \( N \) of a graph a color \( C \) may be chosen.

\[
\begin{align*}
\text{chosenColor}(N,C) & \leftarrow \text{node}(N), \text{color}(C), \neg \text{notChosenColor}(N,C). \\
\text{notChosenColor}(N,C) & \leftarrow \text{node}(N), \text{color}(C), \neg \text{chosenColor}(N,C).
\end{align*}
\]

Rules allow to easily encode complex problems like graph coloring. Finding the answers to such a problem, however, is hard and requires advanced techniques.

ASP solvers are traditionally based on a two-phase computation. First, the variables are removed from the input program by grounding and second, the
The ground program is solved by highly optimized algorithms for propositional problems. Prominent such ground-and-solve systems are DLV [11] and Clingo [5]. The ground program, however, is in the worst case exponential in the size of the nonground program. This makes many real-world programs simply too big to fit in memory and therefore referred to as the *grounding bottleneck* of ASP.

Lazy-grounding on the other hand interleaves the grounding and solving phases and thus overcomes the grounding bottleneck (cf. GASP [13], Asperix [10], and Omiga [3]). Due to this interleaving, such solvers explore the (exponential) search space fundamentally different from CDNL-based solvers, making them very inefficient at solving problems that are trivial for ground-and-solve ASP systems. The Lazy-MX system [2] for the language of $FO(ID)$ follows a different approach and achieves lazy-grounding with efficient solving, but it is restricted to (some) subclass of ASP and requires manual translation.

The recently introduced ASP solver Alpha, however, combines CDNL-based search procedures with lazy-grounding to get the best of both worlds: fast search space exploration and avoidance of the grounding bottleneck at the same time.

**Example 1.** Consider the following program which selects from a domain exactly one element:

```
\text{dom}(1). \ldots \text{dom}(12). \quad \text{sel}(X) \leftarrow \text{dom}(X), \text{not nsel}(X).
\text{\textless} \quad \text{sel}(Y), \text{sel}(X), X \neq Y. \quad \text{nsel}(X) \leftarrow \text{dom}(X), \text{not sel}(X).
```

Adding to this program one rule that forms a large cross-product over selected elements, already exhibits the grounding bottleneck.

```
p(X_1, X_2, X_3, X_4, X_5, X_6) \leftarrow \text{sel}(X_1), \text{sel}(X_2), \text{sel}(X_3), \text{sel}(X_4), \text{sel}(X_5), \text{sel}(X_6).
```

For solvers like Clingo, the amount of required memory increases dramatically when domain elements are added to \text{dom}. A domain size of 20 already requires several gigabytes of memory to ground, while the same program can be solved by lazy-grounding almost immediately and without such memory consumptions.

Blending lazy-grounding and CDNL solving is challenging for a number of reasons. First, usual CDNL solvers guess truth assignments for atoms while lazy-grounding solvers guess whether rules satisfying certain conditions fire or not. Second, atoms may only become \textit{true} due to a rule that fires and must not become \textit{true} due to constraints, since e.g. the constraint \textless not \textit{a}. is no justification for \textit{a} being \textit{true}. Therefore unit-propagation on nogoods may not simply set atoms to \textit{true}. Introducing \textit{must-be-true} as a third truth value fixes this problem, but requires intricate adaptions on the data structures for unit-propagation. Specifically, the 2-watched-literals schema for nogoods is no longer sufficient. A solution to the first challenge is described in detail in [15]. Here, we provide an overview to that solution and are otherwise concerned with the second challenge.

The contributions (after preliminary Section 2) of this work are as follows:
presenting the novel architecture of the Alpha ASP solver (in Section 3) followed by an overview of the Alpha approach for blending lazy-grounding and CDNL-based search,

an enhancement of the two-watched literals schema to obtain efficient propagation performance in the presence of a third truth value and nogoods that are extended with heads (in Section 4), and

benchmarks of the resulting ASP solver Alpha (in Section 5), showing impressive improvements but also directions for future work (in Section 6).

2 Preliminaries

Let \( C \) be a finite set of constants, \( \mathcal{V} \) be a set of variables, and \( \mathcal{P} \) be a finite set of predicates with associated arities, i.e., elements of \( \mathcal{P} \) are of the form \( p/k \) where \( p \) is the predicate name and \( k \) its arity. We assume each predicate name occurs only with one arity. The set \( \mathcal{A} \) of (non-ground) atoms is then given by \( \{ p(t_1, \ldots, t_n) \mid p/n \in \mathcal{P}, t_1, \ldots, t_n \in \mathcal{C} \cup \mathcal{V} \} \). An atom \( a \in \mathcal{A} \) is ground if no variable occurs in \( a \) and the set of variables occurring in \( a \) is denoted by \( \text{vars}(a) \).

The set of all ground atoms is denoted by \( \mathcal{A}_{\text{grd}} \). A (normal) rule is of the form:

\[
\text{at}_0 \leftarrow \text{at}_1, \ldots, \text{at}_k, \text{not } \text{at}_{k+1}, \ldots, \text{not } \text{at}_n,
\]

where each \( \text{at}_i \in \mathcal{A} \) is an atom, for \( 0 \leq i \leq n \). For such a rule \( r \) the head, positive body, negative body, and body are defined as \( H(r) = \{ \text{at}_0 \} \), \( B^+(r) = \{ \text{at}_1, \ldots, \text{at}_k \} \), \( B^-(r) = \{ \text{at}_{k+1}, \ldots, \text{at}_n \} \), and \( B(r) = \{ \text{at}_1, \ldots, \text{at}_n \} \), respectively. A rule \( r \) is a constraint if \( H(r) = \emptyset \), a fact if \( B(r) = \emptyset \), and ground if each \( \text{at} \in B(r) \cup H(r) \) is ground. The variables occurring in \( r \) are given by \( \text{vars}(r) = \bigcup_{a \in h(r) \cup b(r)} \text{vars}(a) \). A literal \( l \) is positive if \( l \in \mathcal{A} \), otherwise it is negative. A rule \( r \) is safe if all variables occurring in \( r \) also occur in its positive body, i.e., \( \text{vars}(r) \subseteq \bigcup_{a \in B^+(r)} \text{vars}(a) \).

A program \( P \) is a finite set of safe rules. \( P \) is ground if each \( r \in P \) is.

A (Herbrand) interpretation \( I \) is a subset of the Herbrand base wrt. \( P \), i.e., \( I \subseteq \mathcal{A}_{\text{grd}} \). An interpretation \( I \) satisfies a literal \( l \), denoted \( I \models l \) if \( l \in I \) for positive \( l \) and \( l \notin I \) for negative \( l \). \( I \) satisfies a ground rule \( r \), denoted \( I \models r \) if \( B^+(r) \subseteq I \land B^-(r) \cap I = \emptyset \) implies \( H(r) \subseteq I \) and \( H(r) \neq \emptyset \). Given an interpretation \( I \) and a ground program \( P \), the FLP-reduct \( P^l \) of \( P \) wrt. \( I \) is the set of rules \( r \in P \) whose body is satisfied by \( I \), i.e., \( P^l = \{ r \in P \mid B^+(r) \subseteq I \land B^-(r) \cap I = \emptyset \} \). \( I \) is an answer-set of a ground program \( P \) if \( I \) is the subset-minimal model of \( P^l \); the set of all answer-sets of \( P \) is denoted by \( \text{AS}(P) \).

A substitution \( \sigma : \mathcal{V} \rightarrow \mathcal{C} \) is a mapping of variables to constants. Given an atom \( a \) at the result of applying a substitution \( \sigma \) to \( a \) is denoted by \( a\sigma \); this is extended in the usual way to rules \( r \), i.e., \( r\sigma \) for a rule of the above form is \( a_0\sigma \leftarrow a_1\sigma, \ldots, \text{not } a_n\sigma \). Then, the grounding of a rule is given by \( \text{grd}(r) = \{ v\sigma \mid \sigma \text{ is a substitution for all } v \in \text{vars}(r) \} \) and the grounding \( \text{grd}(P) \) of a program \( P \) is given by \( \text{grd}(P) = \bigcup_{r \in P} \text{grd}(r) \). Elements of \( \text{grd}(P) \) and \( \text{grd}(r) \) are called ground instances of \( P \) and \( r \), respectively. The answer-sets of a non-ground program \( P \) are given by the answer-sets of \( \text{grd}(P) \).
CDNL-based ASP solving takes a ground program, translates it into nogoods and then runs a SAT-inspired (i.e., a DPLL-style) model building algorithm to find a solution for the set of nogoods. Following established notation, a Boolean signed literal is of the form \( T \) at and \( F \) at for \( at \in A \). A nogood \( ng = \{ s_1, \ldots, s_n \} \) is a set of Boolean signed literals \( s_i, 1 \leq i \leq n \), which intuitively states that a solution cannot satisfy all literals \( s_1 \) to \( s_n \). For example, the nogood \( ng = \{ Ta, Fb \} \) states that it cannot be the case that \( a \) is true and \( b \) is false at the same time. Nogoods are interpreted over assignments, which are sets \( A \) of Boolean signed literals, i.e., an assignment is a (partial) interpretation where false atoms are represented explicitly. A solution for a set \( \Delta \) of nogoods then is an assignment \( A \) such that \( \{ at \mid T \ at \in A \} \cap \{ at \mid F \ at \in A \} = \emptyset \), \{ at \mid T \ at \in A \} \cup \{ at \mid F \ at \in A \} = A \), and no nogood \( ng \in \Delta \) is violated, i.e., \( ng \not\subseteq A \). A solution thus corresponds one-to-one to an interpretation that is a model of all nogoods. For more details and algorithms, see [5–7]. The complement of a Boolean signed literal \( s \), denoted \( \overline{s} \), is \( T \overline{s} = F \overline{s} \) and \( F \overline{s} = T \overline{s} \). Also note that CDNL-based solvers for ASP employ additional checks to ensure that the constructed model is supported and unfounded-free, but these checks are not necessary in the approach presented.

Lazy-grounding, also called grounding on-the-fly, is built on the idea of a computation, which is a sequence \( (A_0, \ldots, A_{\infty}) \) of assignments starting with the empty set and adding at each step heads of applicable rules (cf. [13, 8, 3]). A ground rule \( r \) is applicable in a step \( A_k \), if its positive body already has been derived and its negative body is not contradicted, i.e., \( B^+(r) \subseteq A_k \) and \( B^-(r) \cap A_k = \emptyset \). Observe that finding applicable ground rules, i.e., finding a non-ground rule \( r \) and a grounding substitution \( \sigma \) such that \( r\sigma \) is applicable, is the task of the (lazy) grounder. A computation \( (A_0, \ldots, A_{\infty}) \) then has to satisfy the following conditions besides \( A_0 = \emptyset \), given the usual immediate-consequences operator \( T_P \):

1. \( \forall i \geq 1 : A_i \subseteq T_P(A_{i-1}) \) (the computation contains only consequences),
2. \( \forall i \geq 1 : A_{i-1} \subseteq A_i \) (the computation is monotonic),
3. \( A_{\infty} = \bigcup_{i=0}^{\infty} A_i = T_P(A_{\infty}) \) (the computation converges), and
4. \( \forall i \geq 1 : \forall at \in A_i \setminus A_{i-1}, \exists r \in P \) such that \( H(r) = at \) and \( \forall j \geq i - 1 : B^+(
\subseteq A_j \wedge B^-(r) \cap A_j = \emptyset \) (applicability of rules is persistent through the computation).

It has been shown that \( A \) is an answer-set of a normal logic program \( P \) iff there is a computation \( (A_0, \ldots, A_{\infty}) \) for \( P \) such that \( A = A_{\infty} \) [9, 12]. Observe that \( A \) is finite, i.e., \( A_{\infty} = A_n \) for some \( n \in \mathbb{N} \), because \( C, \mathcal{P} \), and \( P \) are finite.

3 The Alpha Approach

Alpha is a combination of lazy-grounding and CDNL-search to obtain an ASP solver that avoids the grounding bottleneck and shows good search performance.

Architecture. On an abstract level, Alpha achieves this by utilizing a grounder component and a solver component, where the solver is a modified CDNL-search
Fig. 1. Architecture of the Alpha system. Data flow is indicated by arrows. Grounder (left) and CDNL-based solver (right) interact cyclically for lazy-grounding.

algorithm, but both components interact cyclically in the style of lazy-grounding ASP systems. The architecture of the Alpha solver is depicted in Figure 1. The grounder is composed of a parser and a semi-naive grounder that, given a partial assignment, computes all ground rules that potentially fire, transforms the ground rules into nogoods, and sends these to the solver. The solving component is a modified CDNL solver trying to find a satisfying assignment to the set of nogoods presented. It contains a nogood store for unit-propagation on nogoods, conflict resolution implementing conflict-driven nogood learning following the first-UIP schema to learn new nogoods, and a decision heuristic. The most important difference to an ordinary CDNL solver is that guessing is restricted to atoms representing applicable ground rules, i.e., rules whose positive body is satisfied in the current assignment. By that, Alpha prevents unfounded sets from becoming true, thus the assignments constructed by the solver are guaranteed to be unfounded-free. Another difference is that the partial assignments of Alpha contain truth values true, false, and must-be-true where the latter indicates that an atom must be true (e.g. due to a constraint) but no firing rule derives/justifies the atom yet.

Core Algorithm. The remainder of this section provides a summary of the Alpha algorithm and its fundamentals while full details can be found in [15]. The Alpha algorithm at a glance is given by Algorithm 1 which is similar to the main algorithm of CDNL solvers. There is one loop in which the search space is explored and each iteration begins with propagating from the already derived knowledge. If a conflict occurs, it is analyzed in line (a) and a new nogood is derived following the first-UIP schema for conflict-driven learning. In (b) the grounder is requested to derive new nogoods from the assignment derived so far. This is the lazy-grounding part and it is usually absent in CDNL solvers. In
Algorithm 1: The Alpha Algorithm (simplified).

**Input**: A (non-ground) program $P$.

**Output**: The answer-sets $\mathcal{AS}(P)$ of $P$.

Initialize $\mathcal{AS} = \emptyset$, assignment $\mathcal{A}$, and nogood storage $\Delta$.

Run lazy grounder, obtain initial nogoods $\Delta$ from facts.

while search space not exhausted do

Propagate on $\Delta$ extending $\mathcal{A}$.

if there exists conflicting nogood then

   Analyze conflict, learn new nogood, and backjump. (a)

else if propagation extended $\mathcal{A}$ then

   Run lazy grounder wrt. $\mathcal{A}$ and extend $\Delta$. (b)

else if exists an applicable rule then

   Guess as chosen by heuristic. (c)

else if exists an unassigned atom then

   Assign all unassigned atoms to false. (d)

else if no atom in $\mathcal{A}$ assigned to must-be-true then

   $\mathcal{AS} \leftarrow \mathcal{AS} \cup \{\mathcal{A}\}$

   Add enumeration nogood and backtrack. (e)

else

   Backtrack. (f)

return $\mathcal{AS}$

(c) a heuristic decides which atom to guess on. This way of guessing has been newly developed for Alpha and it ensures that the atom guessed on corresponds exactly to an applicable ground rule, i.e., the positive body of the ground rule is already in the assignment and the negative body is not (yet) contradicted by the assignment. When (d) is reached, the interplay of propagation, grounding, and guessing has reached a fixpoint: there are no more applicable ground rule instances and nothing can be derived by propagation or from further grounding. However, there may still be some atoms with unassigned truth value, because the guessing is restricted and does not guess on all atoms. Therefore in (d) all unassigned atoms are explicitly assigned to false and the propagation at the beginning of the following iteration ensures that no nogood is violated. Finally, in (e) the solver checks whether there is an atom assigned to must-be-true but could not be derived by some rule firing to become true. If there is no must-be-true, the current assignment corresponds to an answer-set and it is recorded as such. If the check fails, the current assignment is no answer-set and backtracking occurs in (f).

In order to represent rules using nogoods, Alpha introduces the notion of a nogood with head, that is, a nogood $ng = \{s_1, \ldots, s_n\}_i$ with one distinguished negative literal $s_i$, $1 \leq i \leq n$, such that $s_i = Fa$ for some $a \in \mathcal{A}$. The head of a nogood is denoted by $hd(ng) = s_i$. The head literal, intuitively, captures the idea of the head of a logic programming rule: if the nogood is unit on the head, it is assigned to true and not just must-be-true.
The full representation of a rule by nogoods is as follows: let \( r \) be a rule and \( \sigma \) be a substitution such that \( r\sigma \) is ground, let the positive body be \( B^+(r\sigma) = \{a_1, \ldots, a_k\} \) and the negative body be \( B^-(r\sigma) = \{a_{k+1}, \ldots, a_n\} \) while the head is \( H(r\sigma) = \{a_0\} \), then the nogood representation is given by the following nogoods:

\[
\{ F\beta(r, \sigma), T_a_1, \ldots, T_a_k, F_{a_{k+1}}, \ldots, F_{a_n} \}, \{ F_{a_0}, T\beta(r, \sigma) \}, \{ T\beta(r, \sigma), F_{a_1}, \ldots, \{ T\beta(r, \sigma), F_{a_k} \}, \{ T\beta(r, \sigma), T_{a_{k+1}}, \ldots, \{ T\beta(r, \sigma), T_{a_n} \}
\]

The new atom \( \beta(r, \sigma) \), intuitively, represents the body of the ground rule \( r\sigma \). Notice that the first and second nogood each has a head (as indicated by the subscript 1, the head is the first literal). Despite similarities, this nogood representation differs from the one used by Clingo: first, Alpha uses nogoods with heads and second, there are no nogoods establishing support of ground atoms, because that would require full grounding.

**Example 2.** Consider from Example 1 the rule \( r \) as follows:

\[
\text{sel}(X) : - \text{dom}(X), \text{not nsel}(X).
\]

From an assignment \( A \) where \( \text{dom}(3) \) holds, i.e., \( T\text{dom}(3) \in A \), the grounder generates the substitution \( \sigma : X \mapsto 3 \) for \( r \) and it introduces the new atom \( \beta(r, 3) \) representing the body of the ground rule \( r\sigma \). It then yields the following nogoods:

\[
\begin{align*}
n_1 & : \{ F\beta(r, 3), T\text{dom}(3), F\text{nsel}(3) \} \\
n_2 & : \{ T\beta(r, 3), F\text{dom}(3) \} \\
n_3 & : \{ T\beta(r, 3), T\text{dom}(3) \} \\
n_4 & : \{ F\text{sel}(3), T\beta(r, 3) \}
\end{align*}
\]

Nogoods \( n_1 \) to \( n_3 \) establish that \( \beta(r, 3) \) holds if and only if the body of the ground rule holds. Nogood \( n_4 \) ensures that the head atom is \text{true} whenever \( \beta(r, 3) \) holds. Observe that \( n_1 \) and \( n_4 \) have their first literal indicated as head, i.e., the solver will not set them to \text{must-be-true} but to \text{true} whenever the nogood is unit and all other positively occurring literals are \text{true}. This enables the nogoods to represent rules in the presence of two truth values, \text{must-be-true} and \text{true}.

### 4 Efficient Propagation: 3-Watched-Literals

This section provides details on efficient propagation realized in Alpha. Our approach extends the state-of-the-art propagation technique from SAT and CDNL-based ASP solving known as the 2-watched literals (2WL) schema (cf. [1]). A direct use of 2WL in lazy-grounding ASP solving, however, is not possible due to such solvers using \text{must-be-true} as a third truth value requiring special treatment. Since \text{must-be-true} allows propagation to \text{true}, but no other truth value may be changed once it is assigned, this requires a different propagation mechanism than 2WL, which is designed for propagation to \text{true} and \text{false} only.

Formally, propagation is the task of identifying nogoods that are unit, i.e., nogoods violated except for one yet unassigned literal whose truth value then is set in order to avoid violating the nogood, and subsequently assigning this unassigned literal. In Alpha, a nogood with head may propagate to the truth...
value true, false, and must-be-true while a nogood without head may only propagate to false and must-be-true. Subsequently, there are two notions of being unit: weakly-unit and strongly-unit. Formally, an assignment $A$ in Alpha is over truth values $T$, $F$, and $M$; the Boolean-projection $A^B$ maps $M$ to $T$, i.e., $A^B = \{Ta \mid Ta \in A \lor Ma \in A\} \cup \{Fa \mid Fa \in A\}$. Given a nogood $ng = \{s_1, \ldots, s_n\}$ and an assignment $A$: $ng$ is weakly-unit under $A$ for $s$ if $ng \setminus A^B = \{s\}$ and $s \notin A^B$; $ng$ is strongly-unit under $A$ for $s$ if $ng$ is a nogood with head, $ng \setminus A = \{s\}$, $s = hd(ng)$, and $s \notin A$. By this definition a nogood with head is strongly-unit only if all its positively occurring literals are assigned to true. Also note that only a nogood with head can be strongly-unit and if a nogood is strongly-unit, it also is weakly-unit.

Propagation is the least fixpoint of the immediate unit-propagation, i.e.,
\[
\text{propagate}(A) = \text{lfp}(\Gamma_\Delta(A)) \quad \text{s.t. for a set } \Delta \text{ of nogoods and an assignment } A:
\]
\[
\Gamma_\Delta(A) = A \cup \{Ta \mid \exists \delta \in \Delta, \delta \text{ is strongly-unit under } A \text{ for } s = Fa\}
\]
\[
\cup \{Ma \mid \exists \delta \in \Delta, \delta \text{ is weakly-unit under } A \text{ for } s = Fa\}
\]
\[
\cup \{Fa \mid \exists \delta \in \Delta, \delta \text{ is weakly-unit under } A \text{ for } s = Ta\}
\]

In order to compute the propagation efficiently, we extend the concept of two-watched literals to our setting where nogoods may have a head literal and a nogood can be unit in two different ways. Two-watched literals, intuitively is based on the following observations: if a nogood $\delta$ contains more than two literals $s_1, s_2, s_3 \in \delta$ that are unassigned in some assignment $A$ and one of these, say $s_3$, becomes assigned in $A' \supset A$, then $\delta$ is still not unit. Hence for as long as there are at least two unassigned literals, the nogood need not be checked for being unit. Therefore each nogood only requires two of its unassigned literals to be watched for being assigned in order to detect when the nogood is unit.

For our setting where a nogood may be weakly-unit or strongly-unit, intuitively, two-watched literals are required twice, 2WL for each type of being unit. Since the literal that will be propagated by a strongly-unit nogood always is the head literal of the nogood, it need not be watched explicitly. Therefore, three watches are sufficient. These watches are organized such that each atom is assigned one list per polarity and unit-type. Notice that each nogood requires only three watches but for each atom there are four types of watches.

**Definition 1.** A watch structure $W$ for an assignment $A$ and a set of nogoods $\Delta$ is a mapping $W : A \rightarrow \Delta^4$ of atoms to quadruples of lists (sets) of nogoods in $\Delta$. For a watch structure $W$, each atom $a \in A$ is associated a quadruple of lists
\[
W(a) = (\text{watch}^+(a), \text{watch}^-(a), \text{watch}_+^a(a), \text{watch}^-_a(a)).
\]

The list $\text{watch}^+(a)$ (resp. $\text{watch}^-(a)$) contains all nogoods $\delta$ where a watch is on a positive literal $Ta \in \delta$ (resp. negative literal $Fa \in \delta$) for detecting whether $\delta$ is weakly-unit. The list $\text{watch}_+^a(a)$, resp. $\text{watch}^-_a(a)$, contains all nogoods $\delta$ where a watch is on a positive literal $Ta \in \delta$, resp. negative literal $Fa \in \delta$, for detecting whether $\delta$ is strongly-unit.

A visualization of this data structure is given in Figure 2.
For convenience, in the following we denote for a signed literal $s = Xa$ by \( \text{watch}(s) \) the list $\text{watch}^+(a)$ if $X \in \{T, M\}$ and $\text{watch}^-(a)$ otherwise. Similarly, $\text{watch}_\alpha(s)$ denotes $\text{watch}^\alpha_+(a)$ if $a X \in \{T, M\}$ and $\text{watch}^\alpha_-(a)$ if $X = F$.

In order to obtain correctly watched literals also after backtracking and subsequent assignments (where some assigned atoms may become unassigned and subsequently being propagated), the watches for satisfied nogoods have to point at those literals that were assigned in the highest decision level.

Given an assignment $A$ and an atom $a$, we denote by $dl^w(A, a)$ the decision level on which $a$ is assigned to must-be-true or false in $A$. Similarly, $dl^s(A, a)$ denotes the decision level at which $a$ is assigned to true or false in $A$. Furthermore, for a signed literal $s = Xa$ with $X \in \{F, T, M\}$, we denote by $at(s)$ the atom of the literal, i.e., $at(s) = a$.

Intuitively, the watches of a nogood have to point at either (1) two unassigned literals, or (2) one of these literals atoms is assigned such that the nogood is satisfied and the other literal is either unassigned or assigned at an equal-or-higher decision level. The latter condition ensures that if backtracking removes the satisfying assignment then the second watched literal is guaranteed to be unassigned, i.e., even in case of backtracking the nogood is guaranteed to be either satisfied or contain two unassigned and watched literals.

**Definition 2.** Let $\delta$ be a nogood and $A$ be an assignment, then $s, s' \in \delta$ are potential watches if one of the following holds.

(i) $at(s)$ and $at(s')$ are both unassigned in $A$.

(ii) The atom of $s$ is complementary assigned, i.e., $s' \in A^B$, and either $s'$ is unassigned in $A$ or $dl^w(A, at(s')) \geq dl^w(A, at(s))$.

For a nogood with head there is only one watch, which is not the head itself, and it obeys a similar condition; the main difference being that an atom assigned to must-be-true is treated like it were unassigned.

**Definition 3.** Let $\delta$ be a nogood with head and $A$ be an assignment, then $s_\alpha \in \delta$ with $hd(\delta) \neq s_\alpha$ is a potential $\alpha$-watch if one of the following holds.

(i) $at(s_\alpha)$ is unassigned in $A$, assigned to must-be-true in $A$, or $s_\alpha$ is complementary assigned in $A$. 
\( (ii) \ dl^s(A, \text{at}(s_a)) \geq dl^s(A, \text{at}(hd(\delta))) \) and the head is true, i.e., \( \text{Tat}(hd(\delta)) \in A \).

Example 3. Consider the assignment \( A = \{MC, Fd\} \) with \( dl^w(A, c) \leq dl^w(A, d) \), i.e., \( MC \) was assigned at lower decision level than \( Fd \), and the nogoods \( \delta_1 = \{Fa, Tb, Tc, Fd, Fe\} \), \( \delta_2 = \{Fc, Fd\} \), and \( \delta_3 = \{Fa, Tc\} \), where \( \delta_1 \) and \( \delta_3 \) are nogoods with a head. For \( \delta_1 \) any two literals from \( \{Fa, Tb, Fe\} \) are potential watches since they are all unassigned and any literal in \( \{Tb, Tc, Fe\} \) is a potential \( \alpha \)-watch. The nogood \( \delta_2 \) has the potential watches \( Fc \) and \( Fd \) since \( A \) assigns \( c \) complementary to its occurrence in \( \delta_2 \) and \( d \) has higher decision level than \( c \). Since \( \delta_2 \) has no head, there is no potential \( \alpha \)-watch. For \( \delta_3 \) the literal \( Tc \) is a potential \( \alpha \)-watch since \( c \) is assigned must-be-true in \( A \), but \( \delta_3 \) has no potential watches, intuitively, because \( \delta_3 \) is weakly-unit under \( A \) and propagates \( Fa \).

Intuitively, a watch structure is consistent for an assignment and a set of nogoods, if each nogood is watched correctly.

**Definition 4.** A watch structure \( W \) for a set of nogoods \( \Delta \) is consistent with an assignment \( A \) if for each nogood \( \delta \in \Delta \) there exist potential watches \( s, s' \) and, for \( \delta \) being a nogood with head, a potential \( \alpha \)-watch \( s_a \) such that \( \delta \in \text{watch}(s) \), \( \delta \in \text{watch}(s') \), and \( \delta \in \text{watch}_a(s_a) \) all hold.

Example 4 (continued). Let \( A \) be the same as in Example 3 and let \( \Delta = \{\delta_1, \delta_2\} \).

One watch structure \( W \) consistent with \( \Delta \) and \( A \) as is as follows:

\[
\begin{align*}
W(a) &= (\emptyset, \{\delta_1\}, \emptyset, \emptyset) & W(b) &= (\{\delta_1\}, \emptyset, \{\delta_1\}, \emptyset) & W(c) &= (\emptyset, \{\delta_2\}, \emptyset, \emptyset) \\
W(d) &= (\emptyset, \{\delta_2\}, \emptyset, \emptyset) & W(e) &= (\emptyset, \emptyset, \emptyset, \emptyset)
\end{align*}
\]

Thus \( W \) watches \( \delta_1 \) on \( Fa, Tb \), and \( \alpha \)-watches it on \( Tb \). Furthermore, it watches \( \delta_2 \) on \( Fc \) and \( Fd \) while there exists no \( \alpha \)-watch for \( \delta_2 \) since it has no head. \( W \) is consistent because all watched literals in \( W \) are also potential \( (\alpha) \)-watches in \( A \). Note that for \( \Delta' = \{\delta_1, \delta_2, \delta_3\} \) and \( A \) there exists no consistent watch structure since \( \delta_3 \) has no potential watches (it is weakly-unit in \( A \)).

Computing \( \text{propagate}(A) \) is possible using Algorithm 2 where a watch structure \( W \) consistent with the current assignment \( A \) and set of nogoods \( \Delta \) is maintained. Notice that the algorithm receives as input a set \( \Sigma \) of new assignments, i.e., assignments done by Algorithm 1 outside of propagation (for example by guessing or backtracking). Intuitively, Algorithm 2 iterates over all new assignments (including those it derives itself during propagation) until all new assignments have been processed. For each new assignment the two lists of watched nogoods fitting to the polarity of the assignment are considered, e.g., if \( Fd \) is a new assignment then only nogoods \( \delta \) with \( Fd \in \delta \) are considered. Each of those lists is then checked whether one of its nogoods is violated, weakly-unit, or strongly-unit. If one of the latter two is the case, a new assignment is recorded. Afterwards, the watch structure is adapted such that consistency (with regard to the currently processed assignment) is restored.

The following properties can be shown:
Algorithm 2: propagate

Input: An assignment $A$, a set $\Sigma$ of new assignments, and a watch structure $W$ consistent with $A$ and $\Delta$.

Output: An (extended) assignment $A'$ or a pair of extended assignment $A'$ and a violated nogood $d$.

$A' \leftarrow A$

while $\Sigma \neq \emptyset$ do

$\Sigma \leftarrow \Sigma \setminus \{Xa\}$ for some $Xa \in \Sigma$. // Process each new assignment.

$(\Delta, \Delta_\alpha) \leftarrow \begin{cases} (\text{watch}^+(a), \text{watch}^+_\alpha(a)) & \text{if } X \in \{T, M\}, \\ (\text{watch}^-(a), \text{watch}^-\alpha(a)) & \text{otherwise.} \end{cases}$

foreach $\delta \in \Delta$ do // Propagation to M.F.

if $\delta$ is violated then

return $(A', \delta)$

else if $\delta$ is weakly-unit for $s$ then

Let $s' = Mb$ if $s = Fb$ and $s' = Fb$ otherwise.

$A' \leftarrow A \cup \{s'\}$

$\Sigma \leftarrow \Sigma \cup \{s'\}$

Remove $\delta$ from $\Delta$. // Update ordinary watches.

Let $s, s'$ be potential watches of $\delta$

$\text{watch}(at(s)) \leftarrow \text{watch}(at(s)) \cup \{\delta\}$

$\text{watch}(at(s')) \leftarrow \text{watch}(at(s')) \cup \{\delta\}$

foreach $\delta \in \Delta_\alpha$ do // Propagation to T.

if $\delta$ is strongly-unit then

$A' \leftarrow A \cup \{\text{At}(\text{hd}(\delta))\}$

$\Sigma \leftarrow \Sigma \cup \{\text{At}(\text{hd}(\delta))\}$

Remove $\delta$ from $\Delta_\alpha$. // Update alpha watch.

Let $s$ be a potential $\alpha$-watch of $\delta$

$\text{watch}_\alpha(at(s)) \leftarrow \text{watch}_\alpha(at(s)) \cup \{\delta\}$

return $A'$

---

Proposition 1. Let $W$ be a watch structure $W$ for a set of nogoods $\Delta$ that is consistent with an assignment $A$ and let $A' \supseteq A$ be a larger assignment with $\Sigma = A' \setminus A$. Then, Algorithm 2 running on $A, \Sigma,$ and $W$ returns either

1. a pair $(A'', \delta)$ such that $A''$ is a consequence of $A'$ and $\Delta$ and $\delta \in \Delta$ is violated by $A''$, or
2. an assignment $A'' = \text{propagate}(A')$ and the modified watch structure is consistent with $A''$ and $\Delta$.

5 Evaluation

We evaluated the Alpha solver on four benchmarks, that exercise different parts of a solver, comparing Alpha to the lazy-grounding solvers Omiga and AsPeRiX as well as to Clingo. All benchmarks were performed on a Linux machine with
two 12-core AMD Opteron 6176 SE CPUs and 128 GB RAM. The timeout for each run was 300 secs and the memory limit 8 GB. The HTCondor system\(^3\) was used for load distribution to minimize runtime variations for different runs. Since Java restricts itself to use only parts of the available system memory, the JVM was instructed that 8 GB of RAM are available and that it can use up to 3.5 GB for heap allocations, i.e., Java was called with the following command-line arguments: `-XX:MaxRAM=8000M -Xmx3500M`

We report the average runtimes in seconds on 10 randomly generated instances for each benchmark problem, except for one benchmark where only one instance per size exists. The compared solver versions were: Clingo version 5.2.0, AsPeRiX version 2.5, Omiga built from source using Git commit 037b3f9 and Alpha from source using Git commit a65421f.

*Ground Explosion.* This benchmark is the program of Example 1, i.e., given some domain, select exactly one element from the domain and derive a new atom containing the selected element six times. Table 1 shows the runtimes for domain sizes from 8 up to 1,000 where each solver is requested to compute 10 answer sets.

All lazy-grounding ASP solvers compute the answer sets within seconds for all instances, while Clingo runs out of 8GB memory with a domain of size 18 already. Comparing Alpha with Omiga and AsPeRiX one can observe that Alpha is slower than the other two. This is likely caused by Alpha having to maintain the data structures of a CDN solver (e.g., creation of nogoods, watch structures, etc.) while Omiga and AsPeRiX use a more direct representation of rules. One surprising result is that AsPeRiX takes more than 5 seconds for the instance with domain of size 8, which is much higher than for larger instances. A closer investigation revealed that AsPeRiX needs a lot of time to detect when no more answer sets exist for this particular problem. This only shows for this particular instance where there exist less than the requested 10 answer sets. Requesting 14 answer sets from AsPeRiX for the instance with domain size 12 already results in a timeout. Alpha, in contrast, does not exhibit the same problem.

*Cutedge benchmarks.* This problem was first introduced in [3] and is as follows: given a graph \(G = (V, E)\), choose one edge \(e \in E\) and compute reachability on the graph \(G'\) where \(e\) is cut, i.e., \(G' = (V, E \setminus \{e\})\). This problem is hard for ASP systems that are based on grounding the program upfront, while it is significantly easier for lazy-grounding ASP solvers.

We ran this problem on graphs with 100 to 500 vertices and 3,000 to 125,000 edges, instructing the solvers to compute 10 answer sets each. The results are given in Table 2. As expected, Clingo is only able to solve small instances and starting from graphs with 12,000 edges Clingo always hits the timeout of 300 seconds. Surprisingly, Clingo hits the timeout and does not run out of memory within 300 seconds. Closer inspection revealed, that Clingo indeed runs out of memory when given more time. It seems, however, that grounding in Clingo is

\(^3\) http://research.cs.wisc.edu/htcondor
not fast enough to run out of 8GB memory within 300 seconds time. Table 2 further shows that Alpha is comparable to AsPeRiX and both are slower than Omiga. This may be rooted in the fact that Omiga uses a Rete network for efficient grounding while Alpha uses a semi-naive grounding procedure similar to that of AsPeRiX.

**Graph Colorability.** This problem is inspired by the problem with the same name from the ASP competition. The task is to color a given graph with 5 available colors. This problem poses no grounding problem but requires efficient search procedures. The benchmark was run on randomly generated instances with 10 to 1,000 vertices and 40 to 4,000 edges. For each setting 10 random graphs were constructed. The average runtimes in seconds is reported in Table 3.

As expected, this benchmark is very easy for Clingo, while the lazy-grounding solvers Omiga and AsPeRiX struggle for all but the trivial instances. AsPeRiX performs better than Omiga, even solving instances with 100 vertices and 200 edges. These graphs are very sparse, however, and nearly each coloring yields an answer set. For less-trivial instances with more edges per vertex, like those with 30 vertices and 120 edges, Omiga and AsPeRiX time out on all of them. Alpha on the other hand, is able to solve also the harder instances where search is non-trivial. Comparing Alpha with Clingo we observe that there still is a significant gap in terms of search performance. This is rooted in the fact that Clingo employs numerous efficient search techniques (heuristics, nogood forgetting, nogood minimization, etc.) that are largely lacking in Alpha. There is some progress on implementing heuristics in Alpha (cf. [14]), but due to the specifics

![](image.png)

**Table 1.** Grounding explosion benchmark results. Instance size is the overall number of constants in the domain. Shown is runtime in seconds; out of memory is indicated by memout.
of lazy-grounding (restricted guessing, etc.) the techniques of Clingo cannot be adapted directly.

In order to more precisely compare the lazy-grounding solvers, Table 4 shows their runtimes on graphs with a fixed number of 50 vertices and an increasing number of edges. Omiga has timeouts even for 50 edges while AsPeRiX is able to handle 100 edges. With more than 100 edges Alpha is the only lazy-grounding solver that returned the requested answer sets in time.

Reachability. This benchmark is comprised of a simple positive program computing reachability in a large graph. The task is: given some start vertex of a graph, compute the set of all vertices reachable from the start vertex. The tests were run on 10 randomly generated graphs for each instance size, with 1,000 and 10,000 vertices and 4,000 to 80,000 edges. Since the resulting ASP program contains no negation, Clingo only uses its intelligent grounder while the solver

<table>
<thead>
<tr>
<th>Instance size</th>
<th>Alpha</th>
<th>Omiga</th>
<th>AsPeRiX</th>
<th>Clingo</th>
</tr>
</thead>
<tbody>
<tr>
<td>100/30</td>
<td>12.59(0)</td>
<td>4.25(0)</td>
<td>0.78(0)</td>
<td>27.64(0)</td>
</tr>
<tr>
<td>100/50</td>
<td>11.87(0)</td>
<td>6.22(0)</td>
<td>1.79(0)</td>
<td>79.50(0)</td>
</tr>
<tr>
<td>200/30</td>
<td>22.90(0)</td>
<td>13.46(0)</td>
<td>13.29(0)</td>
<td>300.00(10)</td>
</tr>
<tr>
<td>200/50</td>
<td>45.95(0)</td>
<td>24.20(0)</td>
<td>35.18(0)</td>
<td>300.00(10)</td>
</tr>
<tr>
<td>300/10</td>
<td>16.92(0)</td>
<td>10.08(0)</td>
<td>8.54(0)</td>
<td>291.35(4)</td>
</tr>
<tr>
<td>300/30</td>
<td>59.58(0)</td>
<td>32.36(0)</td>
<td>72.09(0)</td>
<td>300.00(10)</td>
</tr>
<tr>
<td>400/10</td>
<td>49.97(0)</td>
<td>20.72(0)</td>
<td>27.61(0)</td>
<td>300.00(10)</td>
</tr>
<tr>
<td>400/30</td>
<td>300.00(10)</td>
<td>84.73(0)</td>
<td>284.71(4)</td>
<td>300.00(10)</td>
</tr>
<tr>
<td>500/10</td>
<td>62.46(0)</td>
<td>32.01(0)</td>
<td>70.38(0)</td>
<td>300.00(10)</td>
</tr>
<tr>
<td>500/30</td>
<td>300.00(10)</td>
<td>122.16(0)</td>
<td>300.00(10)</td>
<td>300.00(10)</td>
</tr>
<tr>
<td>500/50</td>
<td>300.00(10)</td>
<td>215.01(0)</td>
<td>300.00(10)</td>
<td>300.00(10)</td>
</tr>
</tbody>
</table>

Table 2. Cutedge benchmark results. Instance is number of number of vertices / average percentage of edge being present. Shown is the average runtime in seconds over 10 instances with number of timeouts in parentheses.

<table>
<thead>
<tr>
<th>Instance size</th>
<th>Alpha</th>
<th>Omiga</th>
<th>AsPeRiX</th>
<th>Clingo</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/40</td>
<td>1.41(0)</td>
<td>14.33(0)</td>
<td>31.10(1)</td>
<td>0.02(0)</td>
</tr>
<tr>
<td>20/80</td>
<td>1.53(0)</td>
<td>234.93(6)</td>
<td>128.79(4)</td>
<td>0.02(0)</td>
</tr>
<tr>
<td>30/120</td>
<td>1.59(0)</td>
<td>300.00(10)</td>
<td>230.23(7)</td>
<td>0.03(0)</td>
</tr>
<tr>
<td>40/160</td>
<td>2.54(0)</td>
<td>300.00(10)</td>
<td>217.17(7)</td>
<td>0.04(0)</td>
</tr>
<tr>
<td>50/200</td>
<td>2.31(0)</td>
<td>300.00(10)</td>
<td>300.00(10)</td>
<td>0.04(0)</td>
</tr>
<tr>
<td>100/400</td>
<td>4.24(0)</td>
<td>300.00(10)</td>
<td>300.00(10)</td>
<td>0.06(0)</td>
</tr>
<tr>
<td>400/1600</td>
<td>22.54(0)</td>
<td>300.00(10)</td>
<td>300.00(10)</td>
<td>0.45(0)</td>
</tr>
<tr>
<td>500/2000</td>
<td>33.85(0)</td>
<td>300.00(10)</td>
<td>300.00(10)</td>
<td>0.68(0)</td>
</tr>
<tr>
<td>750/3000</td>
<td>67.22(0)</td>
<td>300.00(10)</td>
<td>300.00(10)</td>
<td>1.46(0)</td>
</tr>
<tr>
<td>1000/4000</td>
<td>119.94(0)</td>
<td>300.00(10)</td>
<td>300.00(10)</td>
<td>2.66(0)</td>
</tr>
</tbody>
</table>

Table 3. Graph 5-colorability benchmark results. Instance is number of vertices / number of edges. Shown is the average runtime in seconds over 10 instances with number of timeouts in parentheses.
Table 4. Graph 5-colorability benchmark on graphs with 50 vertices varying edges. Instance is number of vertices / number of edges. Shown is the average runtime in seconds over 10 instances with number of timeouts in parentheses.

Table 5. Reachability benchmark results. Instance size is number of vertices / multiple of edges of the random graph.

Summary. We observe that Alpha is comparable in speed to the other lazy-grounding solvers for problems where lazy-grounding avoids the grounding bottleneck. In addition to that, Alpha provides much better search performance, making search-intense problems solvable using lazy-grounding. There are, however, many efficient solving techniques not yet available for lazy-grounding ASP solving, making it slower than state-of-the-art ASP solvers on problems where grounding is not an issue. For ASP programs where grounding is problematic, however, Alpha is the best choice as it provides a good compromise between grounding performance and solving performance.

Conclusion

We presented the novel ASP solver Alpha which combines lazy-grounding and CDNL-search to obtain a system that is both, avoiding the grounding bottleneck...
and efficiently exploring the search space. An overview of Alpha and its architecture was given. To provide an efficient propagation the well-known 2-watched literals schema was enhanced to 3-watched literals in order to cope with nogoods being unit in two distinct ways.

Benchmarks showed that Alpha now is on-par with other lazy-grounding solvers on problems where grounding is an issue, while it provides a significant improvement for problems where search is dominating. Since Alpha is very recent, it lacks several important optimizations for search, making it noticeably slower than Clingo. Contrary to Clingo, however, Alpha does not suffer from the grounding bottleneck.

Topics for future work, among many others, are forgetting of learned nogoods, and using dependency information like strongly-connected-components for faster solving.

References

The Syllogistic Reasoning Task:
Reasoning Principles and Heuristic Strategies
in Modeling Human Clusters

Emmanuelle-Anna Dietz Saldanha, Steffen Hölldobler,\(^1\) and Richard Mörbitz\(^*\)

International Center for Computational Logic, TU Dresden, Germany,
\(^1\)and North-Caucasus Federal University, Stavropol, Russian Federation
{dietz,sh}@iccl.tu-dresden.de, richard.moerbitz@tu-dresden.de

Abstract. It seems widely accepted that human reasoning cannot be modeled by means of classical logic. Psychological experiments have repeatedly shown that participants’ answers systematically deviate from the classical logically correct answers. Recently, a new computational logic approach to modeling human syllogistic reasoning has been developed which seems to perform better than other state-of-the-art cognitive theories. We take this approach as starting point, yet instead of trying to model the human reasoner, we aim at identifying clusters of reasoners, which can be characterized by reasoning principles or by heuristic strategies.

1 Introduction

In recent years, a new cognitive theory based on the Weak Completion Semantics (WCS) has been developed. It has its roots in the ideas first expressed by Stenning and van Lambalgen \([13]\), but is mathematically sound \([5]\), and has been successfully applied to various human reasoning tasks. An overview can be found in \([4]\). Hence, it was natural to ask whether the WCS is competitive in syllogistic reasoning and how it performs with respect to the cognitive theories evaluated in the meta-analysis by Khemlani and Johnson-Laird \([7]\). Syllogisms are one of the oldest kinds of logical argument that date back to Aristotle. A syllogism consists of two premises and a conclusion. The syllogistic reasoning task is then: given the two premises, what conclusions are valid? Consider the following pair of syllogistic premises:

\[
\text{All } a \text{ are } b. \quad \text{Some } c \text{ are not } b. \quad (\text{AO3})
\]

The premises can be interpreted as quantified statements. In first-order logic (FOL), any \(c\) not \(a\) follows from these premises. However, according to \([7]\), the majority of participants in experimental studies either concluded some \(c\) are not \(a\) or answered that no valid conclusion follows. Yet, these two responses exclude each other, i.e. it is unlikely that the participants who answered no valid conclusion are the same ones who answered some \(c\) are not \(a\), and vice versa.

\(^*\) The authors are mentioned in alphabetical order.
The possible quantifiers and figures of the premises are shown in Tables 1 and Table 2: Each premise can have one of four quantifiers called *moods*. The entities can appear in four different orders called *figures*. Hence, a problem can be completely specified by the quantifiers of the first and second premise and the figure. The example discussed above is AO3.

In [8], reasoning principles under the WCS for modeling the logical form of the representation of quantified statements in human reasoning are identified. The approach achieved a match of 89% with respect to the conclusions given by the participants and based on the data reported in [7]. This result stands out, as the best of the twelve other state-of-the-art cognitive theories achieved only a match of 84%.

While reasoning with conditionals, humans seem to take certain assumptions for granted which, however, are not stated explicitly in the task description. As psychological experiments show, these assumptions seem not to be arbitrary but instead are systematic in the sense that they are repeatedly made by participants. Furthermore, some assumptions appear in various experiments, whereas other assumptions are only made in very few experiments or only by some participants. In order to identify and structure these assumptions, we view them as principles that are either applied or ignored by the participants who have to solve the task. As starting point, we take the syllogistic reasoning approach presented in [8]. However, a drawback of this approach is that only the matching with respect to the aggregated data is considered, i.e. the approach models the human reasoner. However, the above example and other examples such as cases of the Wason selection task reported in [10], serve as indication that the human reasoner does not exist, but instead we might better search for clusters of human reasoners. These clusters might be expressed by principles, i.e. some clusters might apply some principles that are not applied by other clusters. We also take into account the assumption that some humans do not reason at all to solve syllogistic reasoning tasks. We believe that they use heuristic strategies [15,14] and present a way to combine them with the WCS.

The paper is structured as follows: In Section 2 we present the principles for the representation of quantified statements, motivated by findings from cognitive science and linguistics. The WCS and the encoding of quantified statements within this approach are introduced in Section 3 and 4. In Section 5, the clusters and heuristics are discussed and an overall evaluation of the WCS is presented. In Section 6, we give an overview of our implementation of computing the conclusions that are drawn depending on the applied principles.

<table>
<thead>
<tr>
<th>mood</th>
<th>first-order logic</th>
<th>Short</th>
</tr>
</thead>
<tbody>
<tr>
<td>affirmative universal</td>
<td>$\forall X (a(X) \rightarrow b(X))$</td>
<td>Aab</td>
</tr>
<tr>
<td>affirmative existential</td>
<td>$\exists X (a(X) \land b(X))$</td>
<td>lab</td>
</tr>
<tr>
<td>negative universal</td>
<td>$\forall X (a(X) \rightarrow \neg b(X))$</td>
<td>Eab</td>
</tr>
<tr>
<td>negative existential</td>
<td>$\exists X (a(X) \land \neg b(X))$</td>
<td>Oab</td>
</tr>
</tbody>
</table>

*Table 1. The moods and their formalization.*

<table>
<thead>
<tr>
<th>premise1 premise2</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure1 a-b b-c</td>
</tr>
<tr>
<td>figure2 b-a c-b</td>
</tr>
<tr>
<td>figure3 a-b c-b</td>
</tr>
<tr>
<td>figure4 b-a b-c</td>
</tr>
</tbody>
</table>

*Table 2. The four figures.*
2 Principles about Quantified Statements

Eight principles for developing a logical form of quantified statements are presented. They originate from [8,1] except of the principles in Section 2.5 and 2.8.

2.1 Quantified Statements as Conditionals (conditionals)

Independent of the quantifiers mood, we formalize any relation between two objects of a quantified statement by means of a conditional such that the first object is the antecedent and the second object the conclusion in the conditional. For instance, the statement All a are b is expressed as \( \forall X (a(X) \rightarrow b(X)) \).

2.2 Licenses for Inferences (licenses)

Given the quantified statement all a are b, a license for this inference can then be expressed by all a that are not abnormal, are b [13]. Given the previous formalization of this statement as \( \forall X (a(X) \rightarrow b(X)) \), we extend this conditional by conjoining \( a(X) \) together with an abnormality predicate: \( \forall X (a(X) \land \neg ab(X) \rightarrow b(X)) \). Further, nothing is abnormal with respect to \( X \), i.e. \( \neg ab(X) \) is assumed.

2.3 Existential Import and Gricean Implicature (import)

Humans understand quantifiers differently due to a pragmatic understanding of the language. For instance, in natural language, we normally do not quantify over things that do not exist. Consequently, for all implies there exists. This appears to be in line with human reasoning and has been called the Gricean implicature [3]. This corresponds to what sometimes in literature is also called existential import. The statement some a are b often implies that some a are not b, which again is implied by the Gricean implicature [7].

2.4 Unknown Generalization (unknownGen)

Humans seem to distinguish between some y are z and some z are y, as the results reported by [7] show. Nevertheless, if we would represent some y are z by \( \exists X (y(X) \land z(X)) \) then this is semantically equivalent to \( \exists X (z(X) \land y(X)) \) because conjunction is commutative in FOL. Likewise, humans seem to distinguish between some y are z and all y are z, as we have already discussed in Section 2.3. Accordingly, if we only observe that an object \( o \) belongs to y and z then we do not want to conclude both, some y are z and all y are z. In order to distinguish between some y are z and all y are z, we introduce the following principle: If we know that some y are z, then there must not only be an object \( o_1 \), which belongs to y and z but there must be another object \( o_2 \), which belongs to y and for which it is unknown whether it belongs to z.
2.5 Deliberate Generalization (deliberateGen)

If all of the principles introduced so far are applied to an existential premise, the only object about which an inference can be made is the one resulting from the existential import principle. This is because the abnormality introduced by the licenses for inferences principle has to be false for inference, but due to the unknown generalization principle it is unknown for other objects. There is, however, evidence that some humans still draw conclusions in such circumstances [7]. We believe that they do not take into account abnormalities regarding objects that are not related to the premise.

2.6 Converse Premise (converse)

Although there seems to be some evidence that humans distinguish between some y are z and some z are y (see the results reported in [7]) we propose that premises of the form \( I_{ab} \) imply \( I_{ba} \) and vice versa. If there is an object which belongs to y and z, then there is also an object which belongs to z and y.

2.7 Search Alternative Conclusions to NVC (searchAlt)

Our hypothesis is that when participants are faced with a NVC conclusion (no valid conclusion), they might not want to accept this conclusion and proceed to check whether there exists unknown information that is relevant. This information may be explanations about the facts coming either from an existential import or from unknown generalization. We use only the first as source for observations, as they are used directly to infer new information.

2.8 Contraposition (contraposition)

In FOL, a conditional statement of the form \( \forall(X)(a(X) \rightarrow b(X)) \) is logically equivalent to its contrapositive \( \forall(X)(\neg b(X) \rightarrow \neg a(X)) \). This contraposition also holds for the syllogistic moods A and E. There is evidence in [7] that some of the participants make use of this equivalence when solving syllogistic reasoning tasks. We believe that when they encounter a premise with the mood A (e.g. all a are b), then they might reason with the contrapositive conditional as well.

3 Weak Completion Semantics

3.1 Contextual Logic Programs

Contextual logic programs are (data) logic programs extended by the truth-functional operator \( \text{ctxt} \), called context [2]. (Propositional) contextual logic program clauses are expressions of the forms \( A \leftarrow L_1 \land \ldots \land L_m \land \text{ctxt}(L_{m+1}) \land \).
\[
\begin{array}{cccccccc}
\neg F & \top & \land & \top & \lor & \top & \leftrightarrow & \top & L \ctxt(L) \\
F \top & \bot & \top & \top & \top & \top & \top & \top & \\
\bot & \top & \bot & \bot & \bot & \bot & \bot & \bot & \\
\bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \\
\end{array}
\]

Table 3. The truth tables for the connectives under the three-valued Lukasiewicz logic and for \(\ctxt(L)\). \(L\) is a literal, \(\top, \bot,\) and \(U\) denote true, false, and unknown, respectively.

\[\ldots \land \ctxt(L_{m+p})\] (called rules), \(A \leftarrow \top\) (called facts), \(A \leftarrow \bot\) (called negative assumptions)\(^1\) and \(A \leftarrow U\) (called unknown assumptions). \(A\) is an atom and the \(L_i\) with \(1 \leq i \leq m + p\) are literals. \(A\) is called head and \(L_1 \land \ldots \land L_m \land \ctxt(L_{m+1}) \land \ldots \land \ctxt(L_{m+p})\) as well as \(\top, \bot,\) and \(U\), standing for true, false and unknown respectively, are called body of the corresponding clauses. A contextual (logic) program is a finite set of rules, facts, and assumptions. \(g\mathcal{P}\) denotes the set of all ground instances of clauses occurring in \(\mathcal{P}\). \(A\) is defined in \(g\mathcal{P}\) iff \(g\mathcal{P}\) contains a rule or a fact with head \(A\). \(A\) is undefined in \(g\mathcal{P}\) iff \(A\) is not defined in \(g\mathcal{P}\). The set of all atoms that are undefined in \(g\mathcal{P}\) is denoted by \(\text{undef}(\mathcal{P})\). The definition of \(A\) in \(g\mathcal{P}\) is defined as \(\text{def}(A, \mathcal{P}) = \{A \leftarrow \text{Body} | A \leftarrow \text{Body}\}\) is a rule or a fact occurring in \(g\mathcal{P}\). \(\neg A\) is assumed in \(g\mathcal{P}\) if \(g\mathcal{P}\) contains a negative assumption with head \(A\), \(g\mathcal{P}\) does not contain an unknown assumption with head \(A\), and \(\text{def}(A, g\mathcal{P}) = \emptyset\). We omit the word contextual when we refer to programs, if not stated otherwise.

### 3.2 Three-Valued Lukasiewicz Logic Extended by \(\ctxt\) Connective

We consider the three-valued Lukasiewicz logic together with the \(\ctxt\) connective, for which the corresponding truth values are \(\top, \bot,\) and \(U\), meaning true, false and unknown, respectively. A three-valued interpretation \(I\) is a mapping from the set of logical formulas to the set of truth values \(\{\top, \bot, U\}\), represented as a pair \(I = (I^{\top}, I^{\bot})\) of two disjoint sets of atoms: \(I^{\top} = \{A | A\) is mapped to \(\top\) under \(I\}\) and \(I^{\bot} = \{A | A\) is mapped to \(\bot\) under \(I\}\). Atoms which do not occur in \(I^{\top} \cup I^{\bot}\) are mapped to \(U\). The truth value of a given formula under \(I\) is determined according to the truth tables in Table 3. \(I(F) = \top\) means that a formula \(F\) is mapped to true under \(I\). A three-valued model \(\mathcal{M}\) of \(\mathcal{P}\) is a three-valued interpretation such that \(\mathcal{M}(A \leftarrow \text{Body}) = \top\) for each \(A \leftarrow \text{Body} \in \mathcal{P}\). Let \(I = (I^{\top}, I^{\bot})\) and \(J = (J^{\top}, J^{\bot})\) be two interpretations. \(I \subseteq J\) iff \(I^{\top} \subseteq J^{\top}\) and \(I^{\bot} \subseteq J^{\bot}\). \(I\) is the least model of \(\mathcal{P}\) iff for any other model \(J\) of \(\mathcal{P}\) it holds that \(I \subseteq J\).

\(^1\) Under WCS, the negative assumption will become \(A \leftarrow \bot\) and, hence, \(A\) has to be false. This can, however, be overwritten by other rules and facts (defeating the assumption).
3.3 Integrity Constraints

A set of integrity constraints $IC$ consists of clauses of the form $U \leftarrow Body$, where $Body$ is a conjunction of literals and $U$ denotes the unknown. Hence, an interpretation maps an integrity constraint to $\top$ iff $Body$ is either mapped to $U$ or $\bot$. This understanding is similar to the definition of integrity constraints for the well-founded semantics in [9]. Given an interpretation $I$ and a set of integrity constraints $IC$, $I$ satisfies $IC$ iff all clauses in $IC$ are true under $I$.

3.4 Forward Reasoning: Least Fixed Point of $\Phi_P$

For a given $P$, consider the following transformation: 1. For each ground atom $A$ which is defined in $gP$, replace all clauses of the form $A \leftarrow Body_1, \ldots, A \leftarrow Body_m$ occurring in $gP$ by $A \leftarrow Body_1 \lor \ldots \lor Body_m$. 2. Replace all occurrences of $\leftarrow$ by $\leftrightarrow$. The obtained ground set of equivalences is called the weak completion of $P$ or $wcP$.

Consider the following semantic operator, which is due to Stenning and van Lambalgen [13]: Let $I = \langle I^T, I^\bot \rangle$ be an interpretation. $\Phi_P(I) = \langle J^T, J^\bot \rangle$, where

$$J^T = \{ A \mid A \leftarrow Body \in \text{def}(A, P) \text{ and } Body \text{ is true under } \langle I^T, I^\bot \rangle \}$$
$$J^\bot = \{ A \mid \text{def}(A, P) \neq \emptyset \text{ and } Body \text{ is false under } \langle I^T, I^\bot \rangle \text{ for all } A \leftarrow Body \in \text{def}(A, P) \}$$

[5] showed that the weak completion of non-contextual programs always has a least model under Lukasiewicz logic, which can be obtained as the least fixed point of $\Phi$. However, for programs with the $\text{ctxt}$ operator this property only holds if the programs do not contain cycles [2]. In this paper, let $M_P$ denote the least fixed point of $\Phi_P$. We define $P \models_{wcs} F$ iff $M_P(F) = \top$.

3.5 Backward Reasoning: Explanations by Means of Abduction

An abductive framework $\langle P, A, IC, \models_{wcs} \rangle$ consists of a program $P$, a set $A$ of abducibles, a set $IC$ of integrity constraints, and the entailment relation $\models_{wcs}$. The set of abducibles is $A = \{ A \leftarrow \top \mid A \in \text{undef}(P) \} \cup \{ A \leftarrow \bot \mid A \in \text{undef}(P) \} \cup \neg A$ is not assumed in $gP$. Let $\langle P, A, IC, \models_{wcs} \rangle$ be an abductive framework and $O$ be a set of literals (called observation). $O$ is explainable in $\langle P, A, IC, \models_{wcs} \rangle$ iff there exists an $E \subseteq A$ such that $P \cup E \models L$ for all $L \in O$ and $P \cup E$ satisfies $IC$. $E$ is then called explanation for $O$ given $P$ and $IC$. We restrict $E$ to be minimal, i.e. there does not exist any other explanation $E' \subseteq A$ for $O$ such that $E' \subseteq E$.

Among the minimal explanations, it is possible that some of them entail a certain formula $F$ while others do not. There exist two strategies to determine whether $F$ is a valid conclusion in such cases. $F$ follows credulously, if it is entailed by at least one explanation. $F$ follows skeptically, if it is entailed by all explanations. Due to previous results on modeling human reasoning [4], skeptical abduction is applied.
In this paper, abduction is only applied to observations occurring in \( \mathcal{O}_P = \{ A \mid A \text{ is the head of a fact and a rule occurring in } gP \} \). Usually, this set is further restricted by considering only facts that result from the application of certain principles. The idea is to find an explanation for each observation \( A \in \mathcal{O}_P \) after the fact \( A \leftarrow \top \) has been removed from \( gP \).

3.6 Encoding Quantified Statements in Logic Programs

Negation by Transformation (transformation) The logic programs we consider do not allow heads of clauses to be negative literals. A negative conclusion \( \neg p(X) \) is represented by introducing an auxiliary formula \( p'(X) \) together with the clause \( p(X) \leftarrow \neg p'(X) \) and the integrity constraint \( U \leftarrow p(X) \land p'(X) \). This is a widely used technique in Logic Programming. Applying the principle licenses introduced in Section 2.2, the first clause is extended to \( p(X) \leftarrow \neg p'(X) \land \neg ab_{npp}(X) \) and the assumption \( ab_{npp}(X) \leftarrow \bot \) is added.

No Derivation through Double Negation (doubleNeg) A positive conclusion can be derived from double negation using two conditionals under the WCS. It appears to be the case that humans do not reason in such a way (see [7]). Hence, we block them with the help of abnormalities.

4 Quantified Statements as Logic Programs

Based on the principles and encoding aspects presented in Sections 2 and 3.6, we encode the quantified statements into logic programs. The programs are specified using the predicates \( y \) and \( z \) and depend on the figures shown in Table 2, where \( yz \) can be replaced by \( ab, ba, cb \) or \( bc \). Here, all principles regarding a premise are applied. However, we will later assume different clusters of reasoners, some of which do not apply certain principles (see Section 5). The clauses associated with principles that are not applied are removed for such clusters.

4.1 All y are z (Ayz)

All y are z is represented by \( \mathcal{P}_{Ayz} \), which consists of the following clauses:

\[
\begin{align*}
z(X) & \leftarrow y(X) \land \neg ab_{yz}(X). & \text{(conditionals \\ licenses)} \\
ab_{yz}(X) & \leftarrow \bot. & \text{(licenses)} \\
y(o) & \leftarrow \top. & \text{(import)} \\
ab_{yz}(X) & \leftarrow ctxt(z'(X)). & \text{(contraposition \\ licenses \\ deliberateGen)} \\
y'(X) & \leftarrow \neg z(X) \land \neg ab_{xy}(X). & \text{(contraposition \\ conditionals \\ licenses)} \\
ab_{xy}(X) & \leftarrow \bot. & \text{(contraposition \\ licenses)} \\
y(X) & \leftarrow \neg y'(X) \land \neg ab_{nxy}(X). & \text{(contraposition \\ transformation \\ licenses)}
\end{align*}
\]

As contraposition has been applied, we have to add the integrity constraint \( U \leftarrow y(X) \land y'(X) \). We obtain \( \mathcal{M}_{P_{Ayz}} = \langle \{y(o), z(o)\}, \{ab_{yz}(o)\}\rangle \).
4.2 No y is z (Eyz)

No y is z is represented by $P_{Eyz}$, which consists of the following clauses:

$$
z'(X) \leftarrow y(X) \land \neg ab_{yz}(X).$$  \hspace{1cm} \text{(transformation & licenses)}

$$ab_{yz}(o_1) \leftarrow \bot.$$ \hspace{1cm} \text{(licenses)}

$$z(X) \leftarrow \neg z'(X) \land \neg ab_{nzz}(X).$$ \hspace{1cm} \text{(transformation & licenses)}

$$y(o_1) \leftarrow \top.$$ \hspace{1cm} \text{(import)}

$$ab_{nzz}(o_1) \leftarrow \bot.$$ \hspace{1cm} \text{(licenses & doubleNeg)}

$$y'(X) \leftarrow z(X) \land \neg ab_{zyy}(X).$$ \hspace{1cm} \text{(converse & transformation & licenses)}

$$ab_{zyy}(X) \leftarrow \bot.$$ \hspace{1cm} \text{(converse & licenses)}

$$y(X) \leftarrow \neg y'(X) \land \neg ab_{nzyy}(X).$$ \hspace{1cm} \text{(converse & transformation & licenses)}

$$z(o_2) \leftarrow \top.$$ \hspace{1cm} \text{(converse & import)}

$$ab_{nzyy}(o_2) \leftarrow \bot.$$ \hspace{1cm} \text{(converse & licenses & doubleNeg)}

The integrity constraints $U \leftarrow z(X) \land z'(X)$ and $U \leftarrow y(X) \land y'(X)$ must be added. Iterating $P_{Eyz}$ we obtain $M_{P_{Eyz}} = \langle \{y(o_1), z'(o_1), z(o_2), y'(o_2)\}, \{ab_{yz}(o_1), ab_{nzz}(o_1), z(o_1), ab_{zyy}(o_2), ab_{nzyy}(o_2), y(o_2)\}\rangle$.

4.3 Some y are z (Iyz)

Some y are z is represented by $P_{Iyz}$, which consists of the following clauses:

$$z(X) \leftarrow y(X) \land \neg ab_{yz}(X).$$ \hspace{1cm} \text{(conditionals & licenses)}

$$ab_{yz}(o_1) \leftarrow \bot.$$ \hspace{1cm} \text{(unknownGen & licenses)}

$$y(o_1) \leftarrow \top.$$ \hspace{1cm} \text{(import)}

$$y(o_2) \leftarrow \top.$$ \hspace{1cm} \text{(unknownGen)}

$$ab_{yz}(X) \leftarrow \text{ctxt}(z'(X)).$$ \hspace{1cm} \text{(licenses & deliberateGen)}

$$ab_{byy}(o_2) \leftarrow U.$$ \hspace{1cm} \text{(licenses & deliberateGen)}

$$y(X) \leftarrow z(X) \land \neg ab_{zy}(X).$$ \hspace{1cm} \text{(converse & conditionals & licenses)}

$$ab_{zy}(o_1) \leftarrow \bot.$$ \hspace{1cm} \text{(converse & licenses & unknownGen)}

$$z(o_1) \leftarrow \top.$$ \hspace{1cm} \text{(converse & import)}

$$z(o_2) \leftarrow \top.$$ \hspace{1cm} \text{(converse & unknownGen)}

$$ab_{zy}(X) \leftarrow \text{ctxt}(y'(X)).$$ \hspace{1cm} \text{(converse & licenses & deliberateGen)}

$$ab_{byy}(o_1) \leftarrow U.$$ \hspace{1cm} \text{(converse & licenses & deliberateGen)}

We obtain $M_{P_{Iyz}} = \langle \{y(o_1), y(o_2), z(o_1)\}, \{ab_{yz}(o_1)\}\rangle$. One should observe that $ab_{yz}(o_2)$ is an unknown assumption in $P_{Iyz}$ and, hence, $M_{P_{Iyz}}(z(o_2)) = U$.

4.4 Some y are not z (Qyz)

Some y are not z is represented by $P_{Oyz}$ which consists of the following clauses:

$$z'(X) \leftarrow y(X) \land \neg ab_{ynz}(X).$$ \hspace{1cm} \text{(conditionals & transformation & licenses)}

$$ab_{ynz}(o_1) \leftarrow \bot.$$ \hspace{1cm} \text{(unknownGen & licenses)}

$$z(X) \leftarrow \neg z'(X) \land \neg ab_{nazz}(X).$$ \hspace{1cm} \text{(transformation & licenses)}

$$y(o_1) \leftarrow \top.$$ \hspace{1cm} \text{(import)}

$$y(o_2) \leftarrow \top.$$ \hspace{1cm} \text{(unknownGen)}

$$ab_{nazz}(o_1) \leftarrow \bot.$$ \hspace{1cm} \text{(doubleNeg & licenses)}

$$ab_{nazz}(o_2) \leftarrow \bot.$$ \hspace{1cm} \text{(doubleNeg & licenses)}

We have to add the integrity constraint $U \leftarrow z(X) \land z'(X)$ and obtain $M_{P_{Oyz}} = \langle \{y(o_1), y(o_2), z'(o_1)\}, \{ab_{ynz}(o_1), ab_{nazz}(o_1), ab_{nazz}(o_2), z(o_1)\}\rangle$. 

4.5 Entailment of Conclusions from Pairs of Syllogistic Premises

Based on the applied principles of the previous section, we define when $M_P$ entails a conclusion, where $yz$ is to be replaced by $ac$ or $ca$.

**All (A)** $P \models Ayz$ iff there exists an object $o$ such that $P \models_{wcs} y(o)$ and for all objects $o$ we find that if $P \models_{wcs} y(o_1)$ then $P \models_{wcs} z(o)$. 

**No (E)** $P \models Eyz$ iff there exists an object $o_1$ such that $P \models_{wcs} y(o_1)$ and for all objects $o_1$ we find that if $P \models_{wcs} y(o_1)$ then $P \models_{wcs} \neg z(o_1)$ and there exists an object $o_2$ such that $P \models_{wcs} z(o_2)$ and for all objects $o_2$ we find that if $P \models_{wcs} z(o_2)$ then $P \models_{wcs} \neg y(o_2)$. 

**Some (I)** $P \models Iyz$ iff there exists an object $o_1$ such that $P \models_{wcs} y(o_1) \land z(o_1)$ and there exists an object $o_2$ such that $P \models_{wcs} y(o_2)$ and $P \not\models_{wcs} z(o_2)$ and there exists an object $o_3$ such that $P \models_{wcs} z(o_3) \land y(o_3)$ and there exists an object $o_4$ such that $P \models_{wcs} z(o_4)$ and $P \not\models_{wcs} y(o_4)$. 

**Some Are Not (O)** $P \models Oyz$ iff there exists an object $o_1$ such that $P \models_{wcs} y(o_1) \land \neg z(o_1)$ and there exists an object $o_2$ such that $P \models_{wcs} y(o_2)$ and $P \not\models_{wcs} \neg z(o_2)$. 

**NVC** When no previous conclusion can be derived, no valid conclusion holds.

4.6 Accuracy of Predictions

We have nine different answer possibilities for each of the 64 pairs of syllogistic premises: $Aac$, $Eac$, $Iac$, $Oac$, $Aca$, $Eca$, $Ica$, $Oca$ and $NVC$. For every pair of syllogistic premises, we define two lists of length nine for the predictions of the WCS and for the participants’ answers, where the first element represents $Aac$, the second element represents $Eac$, and so forth. When $Aac$ is predicted under the WCS (or the majority’s conclusions) for a given a pair of syllogistic premises, then the value of the first element of this list is a 1, otherwise it is a 0, and the same holds for the other eight elements in the list. Given

$$\text{comp}(i) = \begin{cases} 1 & \text{if both lists have the same value for the } i\text{th element} \\ 0 & \text{otherwise} \end{cases}$$

the matching percentage of this pair of syllogistic premises is then computed by $\sum_{i=1}^{9} \text{comp}(i)/9$.

5 Clusters and Heuristics

We understand clusters of human reasoners in terms of principles or heuristics. Each cluster is a group of humans that applies the same principles or heuristics. When identifying such clusters, e.g. among the participants of [7], the principles or heuristics used by a single cluster should lead to a significant answer for the pair of syllogistic premises in question. As the answers of all participants have been accumulated in the meta-analysis, the combined answers of all clusters should exactly correspond to the significant answers for that pair of syllogistic premises.
5.1 Basic Principles

Basic principles are assumed to be applied by all reasoners, regardless of any cluster. These are conditionals, licenses, import, and unknownGen. Note that they are not necessarily applicable to every pair of syllogistic premises: unknownGen may only be used for premises with an existential mood.

5.2 Advanced Principles and Clusters

Advanced principles are assumed to be used by not all humans, making them the starting point for clusters. Advanced principles considered in this paper are converse, deliberateGen, contraposition, and searchAlt, but there may exist more. When two individuals differ in the sense that one applies such a principle and the other one does not, we assume that they belong to different clusters.

As an example, consider AO3 introduced in Section 1. According to the encoding described in Section 4, \( P_{AO3,\text{basic}} \) represents the logic program for AO3, where only the basic principles are applied:

\[
\begin{align*}
  b(X) & \leftarrow a(X) \land \neg ab_{ab}(X). \\
  ab_{ab}(X) & \leftarrow \bot. \\
  a(o_1) & \leftarrow T. \\
  b'(X) & \leftarrow c(X) \land \neg ab_{cnb}(X). \\
  c(o_2) & \leftarrow T. \\
  ab_{nbb}(o_2) & \leftarrow \bot. \\
  ab_{nbb}(o_3) & \leftarrow \bot.
\end{align*}
\]

We obtain

\[
\mathcal{M}_{P_{AO3,\text{basic}}} = \langle \{ a(o_1), b(o_1), c(o_2), c(o_3), b'(o_2) \}, \{ ab_{ab}(o_1), ab_{ab}(o_2), ab_{ab}(o_3), ab_{cnb}(o_2), ab_{nbb}(o_2), ab_{nbb}(o_3) \} \rangle.
\]

The atoms that are relevant for conclusions are highlighted. NVC follows from this model. If additionally contraposition is used, we consider

\[
\mathcal{M}_{P_{AO3,\text{contraposition}}} = \langle \{ a'(X) \leftarrow \neg a(X) \land \neg ab_{ha}(X), \neg ab_{ha}(X) \leftarrow \bot, \neg a'(X) \land \neg ab_{nha}(X), ab_{ab}(X) \leftarrow \text{ctxt}(b'(X)) \} \rangle.
\]

instead and obtain

\[
\mathcal{M}_{P_{AO3,\text{contraposition}}} = \langle \{ a(o_1), ab_{ab}(o_2), b(o_1), c(o_2), c(o_3), a'(o_2), b'(o_2) \}, \{ a(o_2), ab_{ab}(o_1), ab_{ab}(o_3), ab_{cnb}(o_2), ab_{nha}(o_1), ab_{nha}(o_2), ab_{nha}(o_3), ab_{nbb}(o_2), ab_{nbb}(o_3), b(o_2), a'(o_1) \} \rangle.
\]

Again, the relevant atoms are highlighted. \( \mathcal{M}_{P_{AO3,\text{contraposition}}} \) entails the conclusion Oca. Assuming two clusters of people whose reasoning process differs in the application of the contraposition principle, we unite the conclusions predicted for the clusters and obtain \( \{ \text{Oca, NVC} \} \). These are exactly the significant answers reported in [7].

In order to represent the principles leading to a conclusions, multinomial processing trees (MPTs) [12] are used. They have been suggested for modeling cognitive theories because they represent cognitive processes as probabilistic...
procedures, thus being able to predict multiple answers and even their quantitative distribution [11]. We set the latent states (inner nodes) of the MPTs to the decisions whether to use certain principles and put the corresponding conclusions in the leaves. An MPT for AO3 based on the clustering described above is presented in Fig. 1. The parameter $p_{\text{contraposition}}$ models the probability that an individual applies the contraposition principle and, therefore, belongs to the corresponding cluster. It can be trained from experimental data with algorithms like expectation-maximization [6]. Note that the MPT in Figure 1 is not complete in the sense that it cannot predict all possible conclusions for AO3. This issue is addressed below.

5.3 Heuristic Strategies

Some theories suggest that some humans do not reason at all to solve syllogistic reasoning tasks, but rely on heuristics such as the atmosphere bias [15] or the matching bias [14]. Such heuristics are simple rules that state what conclusions are likely depending on certain features of the premises, e.g., mood or figure. Given the participants’ answers presented in [7], it seems that often answers are given by a small amount of people (less than 5%). Many of these answers, but also some significant ones, are not (yet) explainable by the WCS. A plausible explanation for that is that these people simply guess or use one of the heuristics mentioned below (educated guess).

A generative approach to model this behavior can be based on MPTs. An MPT for a random guess can lead to all nine conclusions. MPTs for a particular heuristic strategy only take into account the valid conclusions under the corresponding theory. For the atmosphere bias, universal and affirmative conclusions are excluded when one of the premises is existential or negative, respectively. In the case of identical moods, the conclusion must have this mood as well. For the matching bias, the following order from the most to the least conservative quantifier is defined on moods:

\[ \text{E} > \text{O} = \text{I} > \text{A}. \]

A conclusion may not be answered if it is less conservative than one of the premises with respect to that order. We have also observed biased conclusions
in the data of [7] that may be explained by the following heuristic strategy: for almost all pairs of syllogistic premises with figure 1, Xac is answered, while the answer Xca is not given at all. X is the least conservative mood from the premises that is still allowed under the matching strategy (O is preferred over I).

As an alternative to generating the answers given by a cluster of guessers using MPTs, the following inverse process can be considered: predictions of the WCS that are not in accordance with a particular heuristic strategy are not given by a cluster using that strategy. In the filtering approach, these conclusions are suppressed in the predictions. If no conclusion remains, NVC is answered instead. As it is likely that some participants do not use logic [14], such clusters must be modeled under the WCS by using the generative or the filtering approach. As a consequence, MPTs can construct a prediction for all answer possibilities.

5.4 A Clustering Approach

Based on the principles and heuristic strategies described above, the participants of [7] have been partitioned into three reasoning clusters and two clusters applying heuristic strategies:

1. Basic principles, searchAlt, and converse for I.
2. Basic principles, converse for I and deliberateGen.
3. Basic principles, converse for I, E, and contraposition for A.
5. Biased conclusions in figure 1.

Abduction was only used in one cluster because of the computational effort it requires. Although it would be interesting to model this principle for different clusters, the impact would be very small. This is because converse is the only advanced principle that adds existential imports, which we currently consider as atoms for observations. According to the results of [8], abduction has the same results independent of whether only the converse I mood or both the converse I and E mood are used. The matching strategy was implemented using the filtering approach. The biased conclusions in figure 1 heuristics was implemented using the generative approach such that its prediction overwrites the answers of other clusters, except NVC.

5.5 Evaluation

We evaluate the predictions of the WCS based on the clustering approach described in Section 5.4. The combined answers of all clusters are compared with the data of humans presented in [7] and evaluated as described in Section 4.6. The prediction for AO3 and the overall results are compared with other cognitive theories in Table 4. The WCS predicts the participants’ answers in [7] correctly for 32 out of the 64 pairs of syllogistic premises. For 20 cases there is one incorrect prediction, for 11 cases there are two and for one case there are three mismatches. The overall match between the predictions of the WCS and the answers of the participants is 92%.
Table 4. Comparison of the WCS with other cognitive theories. The participants’ answers are highlighted.

### 6 Implementation

The goal of our implementation is to automate the process of evaluating a certain clustering. This is crucial, because as stated above, the number of possible clusterings grows exponentially with the number of principles. We want to be able to evaluate new candidates for an optimal clustering as fast as possible.

For this, we have developed a modular, declarative implementation. It consists of two parts: An implementation of the $\Phi$ operator to compute the least fixed point of a given program $P$, and a framework to generate logic programs from an abstract representation of principles and evaluate the conclusions entailed by them.

#### 6.1 Computing the Least Fixed Point of $\Phi_P$

The least fixed point of $\Phi_P$ is computed in Prolog. The implementation receives a program $P$ – written in Prolog – as input and processes it in two phases. The output is an interpretation $\langle I^\top, I^\perp \rangle$ of $\text{wc} P$ represented as two lists corresponding to $I^\top$ and $I^\perp$.

The input program $P$ is first grounded to obtain $gP$ and, secondly, computes the least fixed point of $\Phi_P$ starting with the empty interpretation $\langle \emptyset, \emptyset \rangle$. Recall that $\Phi_P$ operates directly on $gP$.

The context operator is also implemented so that contextual logic programs can be handled. However, there is a problem: if a contextual logic program $P$ contains a cycle, then the least fixed point of $\Phi_P$ may not exist. Consider the following quantified statements:

\[
\text{All } a \text{ are } b. \quad \text{No } b \text{ is } c. \quad (\text{AE1})
\]

Assume that additionally to the basic principles we apply for each quantified statement the advanced principles converse, deliberateGen, and contraposition. The corresponding program consists of the following clauses:
Consider the highlighted atoms: Note the cycle $c' > b > b' > c > c'$, where $A > B$ if $A$ is an atom in the head of a rule and $B$ is an atom that occurs in the body of the same rule. Because $b'$ is an argument of the context operator and is part of the cycle, this program does not admit a least fixed point. This has to be taken into account when modeling clusters. It must be made sure that the logic program resulting from the applied principles do not contain such cycles. This is guaranteed for the clusters given in Section 5.4.

### 6.2 Computing the Predictions for a Cluster of Reasoners

The framework for evaluating a cluster is written in Haskell. A program run consists of four phases:

1. Generate program $P$ of the pair of syllogistic premises using the principles.
2. Call the Prolog implementation to compute the least fixed point of $\Phi_P$.
3. Extract the conclusions entailed by the least fixed point of $\Phi_P$.
4. Compare the conclusions with the participants’ answers and output score.

The Haskell program contains definitions of datatypes for all entities occurring in the programs, i.e., truth values, atoms, literals, and clauses. These entities are built recursively on each other and have functions for conversion into a Prolog representation.

Principles are implemented as functions that return their corresponding clause representation. Using this abstraction, clusters are written as lists of ‘principle functions’ and are thus valid Haskell source code by themselves. Of course, one consequence is that the user of our implementation has to be familiar with Haskell. However, there are two main advantages of using Haskell source code as a representation. Firstly, many principles are part of a certain subset of the pair of syllogistic premises (e.g. the unknown generalization principle is used for all premises with an existential mood). These connections can be modelled precisely and without redundancy in source code. Secondly, because Haskell is a compiled language, the representation of the pair of syllogistic premises itself is...
compiled. Therefore, a representation is automatically checked and the program
does not crash due to an error in it, which would not be the case if e.g. a string
representation was used.

The Prolog representation of the program results from a function converting
sets of clauses to a string and is written into a file. Then, the Prolog implementa-
tion of the previous program is called to compute the least fixed point of \( \Phi_P \),
which is again written to a file. After completion that file is parsed and the con-
clusions are extracted with respect to the definitions given in Section 4.5. Our
heuristic filters — implemented as post-processing functions — are then applied
to these conclusions. This complete process is done for all 64 pairs of syllogis-
tic premises and their conclusions are compared with the participants’ answers
(available in a file), so that in the end the score of the cluster is computed.

Until now, we have described only the evaluation of a single cluster, although
a clustering consists of multiple clusters whose answers are combined. For this, a
list of clusters can be specified. The program will then compute the predictions
for each cluster, combine them, and compare the results with the participants’
answers.

7 Conclusions

We have successfully extended the approach in [8,1] by introducing two new
principles and applying a clustering approach to model individual differences in
human reasoning. This also takes into account that some people may not reason
at all, but rather guess or apply heuristic strategies. The clustering presented in
Section 5.4 is only the currently known best clustering under WCS but we don’t
know whether it is already the optimal one. However, due to the combinatorial
explosion,\(^2\) it is difficult to find the global optimum. Furthermore, programs
based on certain principles considered for some moods, might not have a least
fixed point, as they contain cycles with respect to the \( \text{ctxt} \) operator. This mus be
taken into account when selecting the principles for a clustering. Finally, we have
applied multinomial processing trees to model that different principles lead to
different conclusions. This information is lost if the predictions for all clusters are
accumulated. This shows how much we depend on the way experimental results
are reported. If we would have more insight about the patterns participants
opted for, we could model single pair of syllogistic premises by MPTs instead of
fitting them to the overall results.

References

completion semantics. In Steffen Hölldobler, Andrey Malikov, and Christoph Wern-
hard, editors, Proceedings of the Young Scientist’s Second International Workshop

\(^2\) For \( n \) principles, there are up to \( 2^n \) possible clusters. Additionally, it is unknown if
the current set of principles is already complete.


In Praise of Impredicativity: A Contribution to the Formalisation of Meta-Programming

François Bry
Institute for Informatics, Ludwig-Maximilian University of Munich, Oettingenstr. 67, 80538 München, Germany
bry@lmu.de

Abstract. Processing programs as data is one of the successes of functional and logic programming. Higher-order functions, as program-processing programs are called in functional programming, and meta-programs, as they are called in logic programming, are widespread declarative programming techniques. In logic programming, there is a gap between the meta-programming practice and its theory: Meta-programming’s formalisations do not explicitly address meta-programming’s impredicativity and are cumbersome. This article aims at overcoming this unsatisfactory situation by discussing the relevance of impredicativity to meta-programming and by revisiting Ambivalent Logic’s syntax and model theory. The impredicative language and model theory proposed in this article are conservative extensions of the language and model theory of first-order logic.

Keywords: Logic Programming, Meta-Programming, Impredicativity, Ambivalent Logic, Model Theory, Barber Paradox, Russell’s Paradox

1 Introduction

Processing programs as data is one of the successes of functional and logic programming. Indeed, in most functional and logic languages, programs are standard data structures what releases programmers writing program-processing programs from explicitly coding or importing parsers. The following program, in which upper case characters are variables, specifies beliefs of Ann and Bill using the programming style called “amalgamation” in [4]:¹

\[
\begin{align*}
\text{believes}(\text{ann}, \text{itRains}) \\
\text{believes}(\text{ann}, \text{itIsWet} \leftarrow \text{itRains}) \\
\text{believes}(\text{bill}, X \leftarrow \text{believes}(\text{ann}, X))
\end{align*}
\]

This program’s intended meaning is that Ann believes that it rains, Ann believes that it is wet when it rains, and Bill believes everything Ann believes. This

¹ Examples referring to beliefs and trust are given because they are intuitive. This article does not address how to specify belief and trust systems but instead how to formalise meta-programming using which such systems can be specified.
program is a meta-program because its second fact, “believes(ann, itIsWet ← itRains)”, includes a clause, “itIsWet ← itRains”. This fact violates the syntaxes of most logics that require that a fact be formed from a predicate, like “believes”, and a list of terms like “ann” but unlike the clause “itIsWet ← itRains”. Indeed, in most logics a clause is a formula, not a term.

While most logics, especially classical predicate logic, prescribe a strict distinction between terms and formulas, meta-programming is based upon disregarding this distinction. Both Prolog and most formalisations of meta-programming pay a tribute to this dictate of classical logic: They require to code a clause like “itIsWet ← itRains” as a term like “cl(itIsWet, itRains)” when it occurs within a fact, expressing the second clause above like

\[
\text{believes(ann, cl(itIsWet, itRains))}
\]

Such a contortion is not necessary. Atomic and compound formulas can be treated as terms as Ambivalent Logic has shown [28, 29]. An expression such as

\[
\text{likes(ann, bill)}
\]

(with the intended meaning that Ann likes Bill) is built up from the three symbols “likes”, “ann” and “bill” that all three can be used for forming nested Ambivalent Logic expressions such as

\[
\text{likes(ann, likes(bill, ann))}
\]

(with the intended meaning that Ann likes that Bill likes her). As a consequence, the Wise Man Puzzle suggested in [37] as a benchmark for testing the expressive power and naturalness of knowledge representation formalisms can be expressed in Ambivalent Logic exactly as it is expressed in [30].

Ambivalent Logic also allows expressions such as

\[
(\text{loves} \land \text{trusts})(\text{ann, bill})
\]

that can be defined by “(loves \land trusts)(X, Y) ← loves(X, Y) \land trust(X, Y)” or even more generally by “(P1 \land P2)(X, Y) ← P1(X, Y) \land P2(X, Y)” and expressions such as

\[
\text{likes(ann, (bill \land charlie))}
\]

that can be defined by “P(X, (Y \land Z)) ← P(X, Y) \land P(X, Z)”. Even more general expressions like the following are possible:

\[
(\forall T \text{ trust}(T) \Rightarrow T)(\text{ann, bill})
\]

or, in a program syntax

\[
(T ← \text{trust}(T))(\text{ann, bill})
\]

with the intended meaning that Ann trusts Bill, expressed as “T(ann, bill)”, in all forms of trust specified by the meta-predicate “trust”. If there are finitely many forms of trust, that is, if “trust(T)” holds for finitely many values of T, then this intended meaning can be expressed by the following rule that relies on negation as failure:
Even though Prolog’s syntax does not allow compound predicate expressions such as “(loves ∧ trusts)” and “(T ← trust(T))” such expressions make sense because they reflect in simple forms a widespread practice of meta-programming in Prolog [11, 49, 41, 25, 24].

This article corrects a deficiency of the model theory of Ambivalent Logic [28, 29]. The model theory proposed below in Section 7 ensures that two expressions (with the intended meaning that Bill believes all what Ann believes) like

\[
\begin{align*}
\text{believes}(&\text{bill}, X \leftarrow \text{believes} (\text{ann}, X)) \\
\text{believes}(&\text{bill}, Y \leftarrow \text{believes} (\text{ann}, Y))
\end{align*}
\]

that differ only in the variables occurring in object-level expressions are identically interpreted. As pointed out in [28, 29], in a same Ambivalent Logic interpretation, the one of these two expressions can be true and the other false.

Furthermore, thanks to the paradigm “quantification makes variables”, the syntax and model theory proposed in the present article are simpler than those of Ambivalent Logic. From an expression like “p(a, b)” in which “a” and “b” are constants, one can construct the expression “∀ a p(a, b)” in which “a” is a variable and “b” is a constant. A price to pay for this feature is that, in contrast to the widespread logic programming practice, universal quantifications can no longer be kept implicit. Arguably, this is a low price to pay since explicit universal quantifications are beneficial to program readability and amount to variable declarations. Another consequence is that the logic has no open formulas. This is, however, not a restriction, since in classical logic open formulas have no expressivity in their own and serve only as components of closed formulas.

This article is structured as follows: Section 1 is this introduction. Section 2 reports on related work. Section 3 is a comparison of meta-programming and higher-order logics. Section 4 recalls why predicativity has been sought for and under which conditions impredicativity is nowadays considered acceptable. Section 5 defines an Ambivalent Logic syntax under the paradigm “quantification makes variables”. Section 6 discusses expressing the Barber and Russell’s Paradoxes in Ambivalent Logic. Section 7 defines a model theory for Ambivalent Logic that corrects a deficiency of the model theory given in [28, 29]. Section 8 is the conclusion.

The novel contributions of this article are as follows:

1. A discussion of how meta-programming relates to higher-order logics and impredicativity.
2. A simplification of the syntax and model theory of Ambivalent Logic.
3. A correction of the model theory of Ambivalent Logic.
4. An explanation why Ambivalent Logic’s impredicativity is acceptable.

2 Related Work

Meta-programming has been considered since the early days of logic programming. It is discussed in [31, 4]. Meta-programming in Prolog is addressed among
others in \cite{11,49,41}. However, the standard formalisation of logic programming \cite{36} does not cover meta-programming. Formalisations of meta-programming of three kinds have been proposed so far:

1. Formalisations relying on higher-order logics commonly called “higher-order logic programming”.
2. Formalisations representing formulas by terms.
3. Ambivalent Logic, a logic lifting the distinction between terms and formulas.

Higher-order logic programming languages, most prominently λ-Prolog \cite{39}, Elf \cite{42} and Twelf \cite{43}, are based on higher-order logics. They have been used for specifying deductive systems and program-processing programs like parsers, compilers, interpreters, and static type checkers \cite{42,44}.

Being based on higher-order logics, higher-order logic programming languages require each expression, term or formula, to belong to a single “order”, or type, and strictly distinguish between terms and formulas. Thus, higher-order logic programming languages preclude expressions like

\[
\text{believes}(\text{ann}, \text{itIsWet} \leftarrow \text{itRains})
\]

\[
\text{believes}(\text{bill}, X \leftarrow \text{believes}(\text{ann}, X))
\]

that have formulas in places where first and higher-order logics allow only terms. As a consequence, higher-order logic programming languages do not support the flavour of meta-programming called “amalgamation” \cite{4}.

Meta-programming and higher-order logics are discussed in more details below in Section 3.

The higher-order logic programming languages have been designed in the spirit of, or after, the Edinburgh Logical Framework LF \cite{22}, a predicative language for a uniform representation of the syntax, the inference rules, and the proofs of predicative logics. LF is based on intuitionistic logic and on the typed lambda calculus. As a consequence, higher-order logic programming languages have no model theories.

Formalisations representing formulas by terms. The formalisations of meta-programming Metalog \cite{5,6}, the language proposed by Barklund in \cite{3}, Reflexive Prolog \cite{13}, R-Prolog* \cite{50,51}, ‘LOG (spoken “quotelog”) \cite{9}, HiLog \cite{10}, Gödel \cite{24} and the language proposed by Higgins in \cite{23} rely on representing, or mirroring, in different yet conceptually similar manners a formula or a program by one or two first-order terms thus keeping with the imperative of first and higher-order logics that formulas and terms be distinct categories.

Metalog uses “metalevel names” for representing object programs as first-order terms. Reflexive Prolog distinguishes meta-variables from object variables and represents object programs using first-order “name constants”. R-Prolog* \cite{50,51} uses a quote notation à la Lisp \cite{47} for representing object programs as first-order terms. Barklund proposes in \cite{3} a “naming scheme” for representing object programs using terms built from reserved first-order function symbols and
first-order constants. 'LOG relies on a “naming scheme” associating “with every syntactic object of the language, from characters to programs”, a “constant name” and a structured ground term called the object’s “structural representation”. HiLog [10] treats every symbol except connectives and quantifiers as a predicate and maps it to

- an “infinite tuple of functions” over the universe, one function of arity \( n \) for each \( n \in \mathbb{N} \), and
- an “infinite tuple of relations” over the universe, one relation of arity \( n \) for each \( n \in \mathbb{N} \).

Gödel [25, 24] is equipped with a variable typing distinguishing object variables from meta-variables and “ground representations” of programs as first-order terms. Higgins proposes in [23] a language relying on a “naming scheme” similar to that of 'LOG associating with every syntactic object a first-order “primitive name” and a first-order “structured name”.

The articles [46, 33] give meta-programming foundations in the form of a Herbrand model theory that also rely on mirroring a formula or a program by a first-order terms in the meta-language of a “demo” or “solve” meta-predicate [31, 4].

Ambivalent Logic [28, 29] was the first formalisation of meta-programming giving up the distinction between terms and formulas thus making superfluous representations of formulas by terms the afore-mentioned languages and formalisms rely on.

It is surprising that this seminal and elegant proposal received little consideration in the logic programming community. The article [2] titled “Meta-Variables in Logic Programming, or in Praise of Ambivalent Syntax” is a noticeable exception. Relying on Ambivalent Logic’s treatment of formulas as terms, and therefore of meta-variable as first-order variables, it extends SLD-resolution [36] to logic programs with meta-variables and establishes the soundness and completeness of that extension.

The data modelling language RDF [14] provides a form of meta-programming by treating formulas of the form “subject properties object”, so-called triples, as terms: A triple can be for example the subject of another triple. However, RDF imposes syntactical restrictions on the use of properties in meta-level expressions: They cannot occur as triples’ subjects.

Xcerpt [7, 8, 45], a declarative XML query language inspired from logic programming, is related to the present work in the sense that it treats expressions and what amount to the terms of a logic language uniformly. Xcerpt however, is a query language and therefore amenable neither to full programming, nor to meta-programming.

More on meta-programming and on meta-reasoning can be found in the survey [12].
3 Meta-Programming and Higher-Order Logics

Prolog-style meta-programming \([11, 49, 41]\) called “amalgamation” in \([4]\) allows

1. variables to range over predicates and formulas,
2. predicates the arguments of which are predicates or formulas, and
3. reflection \([17]\) in the sense that every predicate can have any predicate, including itself, and formulas including any predicate as argument.

Prolog extremely permissive approach to meta-programming goes back to a fruitful disregard by Alain Colmerauer, Prolog’s designer, of the relationship between meta-programming, higher-order logic, and impredicativity and to a time at which the undecidability of unification in second-order logic \([34]\) and third-order logic \([27]\) as well as Damas-Hindley-Milner type systems \([26, 40, 16, 15]\) were unknown. Prolog permissive approach to meta-programming is very useful in practice, as the following examples demonstrate:

- A unary predicate ranging over all unary predicates (including itself) can be used for both static and dynamic type checking.
- Reflection in the sense of a predicate occurring in an argument of itself can be used for applying an optimisation to the very predicate specifying this optimisation.
- Formulas occurring in place where first and higher-order logics expect terms are useful for expressing believes and trust as shown in the articles \([4, 31, 30]\) and outlined in the introduction.

Can meta-programming be formalised in higher-order logics? In order to answer this question, let us recall higher-order logics main traits \([32, 18]\):

- First-order logic has (first-order) variables that are (first-order) terms and (first-order) predicates the arguments of which are (first-order) terms.
- Second-order logic extends first-order logic with second-order variables that are first-order predicates.
- Third-order extends second-order logic with (second-order) predicates the arguments of which are first-order predicates.
- Fourth-order logic extends third-order logic with fourth-order variables that are second-order predicates.
- Etc.

Disregarding function symbols, the model theories of higher-order logics are as follows:

- A first-order term is interpreted as an individual, that is, as an element of the universe of discourse.
- A first-order n-ary predicate is interpreted as sets of n-tuples of individuals.
- A second-order n-ary predicate is interpreted as a set of n-tuples of sets of individuals.
- A third-order n-ary predicate is interpreted as a set of n-tuples of sets of (tuples of) individuals.
Prolog-style meta-programming resembles higher-order logics since it has variables for predicates or formulas and predicates the arguments of which are predicates or formulas. However, in contrast to Prolog-style meta-programming, higher-order logics allow no confusion of orders like

- A unary predicate ranging over all unary predicates including itself.
- A predicate being, or occurring in, an argument of itself.

Such confusions are widespread in Prolog-style meta-programming [11, 4, 31, 49, 41]. Thus, Prolog-style meta-programming cannot be fully formalised in higher-order logics.

4 Predicativity and Impredicativity

Consider a property $P$ on the nodes of an undirected graph $G$ defined as follows: A node $n$ of $G$ has property $P$ if its immediate neighbours all have property $P$. This definition is not acceptable because it is ambiguous: It applies among others to the property holding of no nodes and to the property holding of all nodes.

At the beginning of the 20th century Henri Poincaré and Bertrand Russell have proposed the Vicious Circle Principle that forbids circular definitions, that is, definitions referring to the very concept they define [48]. This principle has been endorsed by many contemporary mathematicians. Russell called “predicative” definitions that adhere to the Vicious Circle Principle, “impredicative” definitions that violate it.

The Vicious Circle Principle principle, however, has a drawback: It forbids hereditary\(^2\) and, more generally, inductive definitions [1] (like the definition of the formulas of a logic or of the programs of a programming language). The definition sketched above makes sense as an inductive definition:

- Basis cases: A (possibly empty) set of nodes of $G$ is specified that have the property $P$.
- Induction case: If a node has the property $P$, then all its immediate neighbours have the property $P$.

An inductively defined property (or set) is the smallest property (or set) that fulfils the basis and induction cases of its definition. Thus, understood as an inductive definition, the definition given at the beginning of this section is that of the empty relation (that is, of the relation that holds of no nodes).

The Vicious Circular Principle also rejects definitions that quantify over domains the definitions of which refer to the entities defined. Such a definition

\(^2\) The definition of a property $P$ is hereditary if it states that whenever a natural number $n$ has property $P$, so does $n + 1$. 


is the core of Russell’s Paradox: A set of all sets that are not elements of themselves.3

The Vicious Circle Principle motivated Russell to develop the Ramified Theory of Types, that is, logics of various, first and higher, orders. By considering a strict hierarchy of types, or orders, as recalled in Section 3, quantifications over domains the definitions of which refer to the entities defined are excluded. Some definitions that quantify over domains the definitions of which refer to the entities defined are unambiguous, though:

- $y$ is the smallest element of an ordered set $S$ if and only if for all elements $x$ of $S$, $y$ is less than or equal to $x$, and $y$ is in $S$.
- $y$ is the greater lower bound of an ordered set $S$ if and only if for all elements $x$ of $S$, $y$ is less than or equal to $x$, and any $z$ less than or equal to all elements of $S$ is less than or equal to $y$.

Such examples led some mathematicians, most notably Kurt Gödel, to object that impredicative definitions are acceptable provided the entities they refer to are clearly apprehensible [21]. Nowadays, most logicians and mathematicians follow Gödel and accept impredicative definitions of the following kinds [1, 19]:

- Inductive definitions.
- Impredicative definitions that characterise elements (like the smallest number in a set) of clearly apprehensible sets (including inductively defined sets).

5 Ambivalent Logic’s Syntax Revisited

This section introduces “expressions” that amount to both the terms and the formulas of standard logic languages. Except for the use of the paradigm “quantification makes variables”, the syntax given below is that of Ambivalent Logic [28, 29].

Definition 1 (Expressions). An Ambivalent Logic language is defined by

- the logical symbols consisting of the connectives $\land, \lor, \Rightarrow$, and $\neg$, and of the quantifiers $\forall$ and $\exists$,
- at least one and at most finitely many non-logical symbols each of which is distinct from every logical symbol.4

The expressions of an Ambivalent Logic language and their outermost symbols are inductively defined as follows:

- A non-logical symbol $s$ is an expression the outermost symbol of which is $s$ itself.

3 The paradox is that such a set cannot exist because if it would be an element of itself, then by definition it would not be an element of itself, and if it would not be an element of itself, then by definition it would be an element of itself.

4 A countable infinite set of non-logical symbols could be considered but this is not necessary in programming since programs are finite.
If $E$ and $E_1,\ldots,E_n$ with $n \geq 1$ are expressions, then $E(E_1,\ldots,E_n)$ is an expression the outermost symbol of which is the outermost symbol of $E$.

If $E$ is an expression, then $(\neg E)$ is an expression the outermost symbol of which is $\neg$.

If $E_1$ and $E_2$ are expressions, then $(E_1 \land E_2)$, $(E_1 \lor E_2)$, $(E_1 \Rightarrow E_2)$ are expressions whose outermost symbols are $\land$, $\lor$, and $\Rightarrow$ respectively.

If $E_1$ is an expression the outermost symbol of which is a non-logical symbol and if $E_2$ is an expression, then $(\forall E_1 \ E_2)$ and $(\exists E_1 \ E_2)$ are expressions whose outermost symbols are $\forall$ and $\exists$ respectively.

The set of expressions of an Ambivalent Logic language is not empty since, by definition, the language has at least one non-logical symbol.

More parentheses are required by Definition 1 than in classical logic. This is necessary for distinguishing an expression such as $(\neg a)(b)$ from $(\neg a(b))$. Provided a few additional parentheses are added, first-order logic formulas are expressions in the sense of Definition 1, that is, the syntax given in this section is a conservative extension of the syntax of first-order logic.

A logical expression is an expression the outermost symbol of which is a logical symbol, that is, a negation, a connective, or a quantifier. Logical expressions correspond to first-order logic compound (that is, not atomic) and quantified formulas.

An atom is an expression the outermost symbol of which is a non-logical symbol. Thus, "believes(Ann, (itRains \land itIsWet))" is an atom while "(believes(Ann, itRains) \land believes(Ann, itIsWet))" is not an atom but instead a logical expression. Note that if $A_1$ and $A_2$ are atoms, then the (well-formed) expressions $(\forall A_1 A_2)$ and $(\exists A_1 A_2)$ are no atoms.

Skolemization can be specified as usual by adding additional non-logical symbols to the language.

For the sake of simplicity, the definition above assumes that every non-logical symbol has all arities. This reflects a widespread logic programming practice: Using p/2 in a Prolog program, for example, does not preclude using p/3 in the same program.\footnote{Assuming that non-logical symbols of an ambivalent language have all arities is a convenience, not a necessity.}

In contrast to the syntax of Ambivalent Logic given in [28, 29], the above definition does not distinguish between variables and constants. According to the above definition, quantifications make variables:

- "likes(ann, bill)" contains no variables.
- "\exists ann likes(ann, bill)" means that there is someone who likes Bill.
- "\forall bill \exists ann likes(ann, bill)" means that everyone is liked by someone.

A first advantage of the paradigm "quantifications make variables" is that every expression is closed. Indeed, a symbol which is not quantified such as "x" in "likes(x, bill)" is not a variable. This is not a restriction, since in logics with open formulas, open formulas serve only as components of closed formulas. The
paradigm “quantifications make variables” corresponds to the declarations of programming languages.6

Explicit quantifications as introduced in Definition 1 are not usual in logic programming. Combined with the paradigm “quantifications makes variables”, they are useful for meta-programming because they make it easy to transform expressions. The expression “likes(ann, bill)” for example can be abstracted into “(∃ likes likes(ann, bill))” (meaning that Ann and Bill are in some relationship) and generalised as “(∀ likes likes(ann, bill))” (meaning that Ann and Bill are in all possible relationships).

6 The Barber and Russell’s Paradoxes in Ambivalent Logic

Since Ambivalent Logic’s syntax is a conservative extension of the syntax of first-order logic, a formulation of the Barber Paradox in first-order logic like the following is also a formulation of that paradox in Ambivalent Logic:

\[
\begin{align*}
\text{man(barber)} \\
(\forall y \ (\text{man}(y) \Rightarrow (\text{shaves}(\text{barber}, y) \Leftrightarrow (\neg \text{shaves}(y, y)))))
\end{align*}
\]

where, extending Definition 1, \( (E_1 \Leftrightarrow E_2) \) is defined as a shorthand notation for \(((E_1 \land E_2) \lor ((\neg E_1) \land (\neg E_2)))\). The above sentences express that the barber is a man shaving all men that do not shave themselves. The Barber Paradox is, in spite of its name, not a paradox but an inconsistency: The barber cannot exist because he would have both to shave himself and not to shave himself. The self-contradictory formula

\[
(\text{shaves}(\text{barber}, \text{barber}) \Leftrightarrow (\neg \text{shaves}(\text{barber}, \text{barber})))
\]

follows in first-order logic from the above specification of the Barber Paradox. A formula expressing the Barber Paradox is inconsistent with respect to the model theory defined in the next section as it is in first-order logic.

The syntax of [28, 29] and of Section 5 that does not distinguish between formulas and terms gives rise to (well-formed) expressions that are not expressible in first-order logic, and that, like the Barber Paradox, are inconsistent. One such expression is the following:

\[
(*) \ (\forall x \ (e(x) \Leftrightarrow (\neg x(x))))
\]

Instantiating \( x \) with \( e \) in \( (*) \) yields the self-contradicting expression

\[
(e(e) \Leftrightarrow (\neg e(e)))
\]

6 The paradigm “quantification makes variables” is akin to lambda-abstraction. We give it an expressive denomination for avoiding referring to the lambda calculus our proposal does not build upon.
Expression (⋆) is inconsistent for the model theory given in the next section as it must be in every well-specified model theory because it is self-contradictory. Thus, Expression (⋆) is, like the above specification of the ill-named Barber Paradox, not the specification of a paradox but of a mere inconsistency.

But (⋆) is a formulation of Russel’s Paradox in Ambivalent Logic and Russel’s Paradox is known [35] to be a real paradox! The apparent dilemma is explained by the syntax of Ambivalent Logic being more expressive, or more permissive, than the syntax of first-order logic: Russell’s Paradox cannot be expressed by a first-order logic formula but it can be expressed by an Ambivalent Logic expression. While it is paradoxical to think of a set that cannot exist, an inconsistent expression is not paradoxical.

Russell’s paradox stems from the conception that every expression must define a set. Giving up this conception allows for “impredicative sets”7 provided their specifications in an appropriate logic, like Ambivalent Logic, are consistent and provided this logic has a well-defined semantics (for example, a model theory).

7 Ambivalent Logic’s Model Theory Revisited

Since atoms (that is, expressions whose outermost symbols are neither connectives nor quantifiers) like

\[ E = \text{believes(ann, } (\forall x (\text{believes(ann, } x) \Rightarrow \text{believes(bill, } x))) \]

(with the intended meaning that Ann believes that Bill believes all what she herself believes) of an Ambivalent Logic language corresponds to both ground atoms and ground terms of first-order logic languages, the set of all expressions of an Ambivalent Logic language corresponds to both the Herbrand universe and the Herbrand basis of a first-order logic language.

An expression like \( E \), however, can have variants, like

\[
\begin{align*}
\text{believes(ann, } (\forall y (\text{believes(ann, } y) \Rightarrow \text{believes(bill, } y))) \\
\text{believes(ann, } (\forall t(a) (\text{believes(ann, } t(a)) \Rightarrow \text{believes(bill, } t(a)))))
\end{align*}
\]

that should be given the same meaning as \( E \) even though they syntactically differ from \( E \). Thus, the Herbrand-style universe of an Ambivalent Logic language is the set of equivalence classes of the language’s atoms with respect to the variant relation.

This variance relation is defined similarly as in first-order logic. It can be informally defined as follows. Two expressions \( E_1 \) and \( E_2 \) of an Ambivalent Logic language are variants of each other, noted \( E_1 \sim E_2 \), if there is a one-to-one mapping \( \sigma \) of the occurrences of their quantified expressions (serving as variables) such that replacing each such occurrence \( E \) within its quantification scope in \( E_1 \) by \( \sigma(E) \) yields \( E_2 \).

The following atoms are variants of each other:

7 In the sense of sets that can be element of themselves [19].
The following atoms (with the intended meaning that for all possible forms b of belief specified by “belief(b)”, Ann b-believes that Bill b-believes all what she herself b-believes) are variants of each other:

\[(∀ b (\text{belief}(b) ⇒ b)(\text{ann}, (∀ x (\text{b(ann, x) ⇒ b(bill, x))))))) \]

\[(∀ B (\text{belief}(B) ⇒ B)(\text{ann}, (∀ y (\text{B(ann, y) ⇒ B(bill, y))))))) \]

The following expressions are variants of each other:

\[(∀ x (p(x) ⇒ (∃ x q(x)))) \]  \( (∀ x (p(x) ⇒ (∃ y q(y)))) \]

Definition 2 (Universe). Let \( A \) be the set of atoms of an Ambivalent Logic language \( L \) and \( ∼ \) the variant relation of \( L \). The Herbrand universe of \( L \) is \( A ∕ ∼ \), the set of equivalence classes of \( ∼ \).

If \( E \) is an expression and if \( A_1 \) and \( A_2 \) are atoms, let \( E[A_2/A_1] \) denote the expression obtained from \( E \) by simultaneously replacing all occurrences of \( A_1 \) in \( E \) by \( A_2 \).

Definition 3 (Interpretation and Model). An Herbrand interpretation \( I(S) \) of an Ambivalent Logic language \( L \) is specified as a subset \( S \) of the universe of \( L \) of \( L \).

Satisfaction in an Herbrand interpretation \( I(S) \) of \( L \) is defined as follows, where:

- \( E, E_1, \) and \( E_2 \) denote expressions.
- \( A, A_1 \) and \( A_2 \) denote atoms.

\[
I(S) \models A \quad \text{iff} \quad A \in C \in S
\]
\[
I(S) \models \neg E \quad \text{iff} \quad I(S) \not\models E
\]
\[
I(S) \models (E_1 \land E_2) \quad \text{iff} \quad I(S) \models E_1 \quad \text{and} \quad I(S) \models E_2
\]
\[
I(S) \models (E_1 \lor E_2) \quad \text{iff} \quad I(S) \models E_1 \quad \text{or} \quad I(S) \models E_2
\]
\[
I(S) \models (E_1 ⇒ E_2) \quad \text{iff} \quad \text{if} \ I(S) \not\models E_1, \ \text{then} \ I(S) \models E_2
\]
\[
I(S) \models ∃A_1 E \quad \text{iff} \quad I(S) \models E[A_2/A_1] \quad \text{for some} \ A_2
\]
\[
I(S) \models ∀A_1 E \quad \text{iff} \quad I(S) \models E[A_2/A_1] \quad \text{for all} \ A_2
\]

An interpretation is called a model of an expression (a set of expression, resp.) \( E \) if it satisfies \( E \) (all expressions in \( E \), resp.).

Satisfaction of the logical expressions of an Ambivalent Language, that is, expressions whose outermost symbols are connectives (\( \neg, \land, \lor, ⇒ \)) or quantifiers (\( ∀, \exists \) is defined exactly like in first-order logic. Satisfaction of the atoms of an Ambivalent Language is not defined like in first-order logic: it is based on variance, not on syntactical identity.

Applied to a first-order logic language, Definitions 2 and 3 amount to the definitions of Herbrand universes, interpretations and models of first-order logic. Indeed, a ground atom \( A \) of a first-order logic language is the only element of its equivalence class for the variant relation. Thus, Definition 3 is a conservative extension of first-order logic’s notions of Herbrand interpretations and models.
An interpretation as defined above can be seen as a set $A$ of atoms. An atom is satisfied in the interpretation specified by $A$ if and only if it is a variant of an element of $A$. The definition of an interpretation of an Ambivalent Logic language given in [28, 29] is more stringent: after that definition and like for first-order logic, an atom is satisfied in the interpretation specified by a set of atoms $A$ if and only if it is an element of $A$.

Consider the following set $P$ of expressions, a simple meta-program (with explicit quantifications) on the beliefs of Ann and Bill:

\begin{align*}
& \text{believes}(\text{Ann}, \text{itRains}) \\
& \text{believes}(\text{Ann}, (\text{itRains} \Rightarrow \text{itIsWet})) \\
& \text{believes}(\text{Ann}, (∀x \ (\text{believes}(\text{Ann}, x) \Rightarrow \text{believes}(\text{Bill}, x)))) \\
& (∀x \ (\text{believes}(\text{Ann}, x) \Rightarrow \text{believes}(\text{Bill}, x)))
\end{align*}

The following set of atoms specifies a model of $P$:

\begin{align*}
& \text{believes}(\text{Ann}, \text{itRains}) \\
& \text{believes}(\text{Ann}, (\text{itRains} \Rightarrow \text{itIsWet})) \\
& \text{believes}(\text{Ann}, (∀y \ (\text{believes}(\text{Ann}, y) \Rightarrow \text{believes}(\text{Bill}, y)))) \\
& \text{believes}(\text{Bill}, \text{itRains}) \\
& \text{believes}(\text{Bill}, (\text{itRains} \Rightarrow \text{itIsWet})) \\
& \text{believes}(\text{Bill}, (∀z \ (\text{believes}(\text{Ann}, z) \Rightarrow \text{believes}(\text{Bill}, z))))
\end{align*}

The definition of an interpretation given in [28, 29] is more stringent than Definition 3: Instead of relaying on the variant relationship, it requires syntactical identity. As a consequence, the set $S$ of atoms given above does not specify a model in the sense of [28, 29] of the set of expressions $P$ given above. Clearly, this is undesirable. Indeed, meta-programming requires to interpret identically meta-level expressions like the following that are variants of each other:

\begin{align*}
& \text{believes}(\text{Ann}, (∀x \ (\text{believes}(\text{Ann}, x) \Rightarrow \text{believes}(\text{Bill}, x)))) \\
& \text{believes}(\text{Ann}, (∀y \ (\text{believes}(\text{Ann}, y) \Rightarrow \text{believes}(\text{Bill}, y))))
\end{align*}

Even though an Ambivalent Logic language gives rise to impredicative expressions (like the definition (⋆) of Russell’s paradoxical set given in Section 6), the model theory given above is acceptable for the reasons mentioned at the end of Section 4. Indeed, its refers to an inductively defined universe, the set of all expressions, and impredicative expressions like

\begin{align*}
& \text{believes}(\text{Ann}, (∀x \ (\text{believes}(\text{Ann}, x) \Rightarrow \text{believes}(\text{Bill}, x))))
\end{align*}

characterise elements of that universe.

8 Conclusion

This article has proposed a simplification of the syntax of Ambivalent Logic [28, 29] and corrected a deficiency of Ambivalent Logic’s model theory. With the exception of Ambivalent Logic, all formalisations of meta-programming proposed so far remain in the realm of predicativity, that is, they adhere
to the Vicious Circle Principle. In the case of higher order logic programming, this adherence results in formalisations rejecting amalgamation. In the case of the formalisations representing formulas by terms, it results in complicated formalisations.

Ambivalent Logic’s impredicativity has been shown in this article to result in a syntax using which seemingly paradoxical expressions can be defined that are mere inconsistencies, an appropriate manner to express impossibilities. Ambivalent Logic’s impredicativity has been shown to be impeccable for reasons first stressed by Kurt Gödel and nowadays widely accepted.

Impredicativity frees from constraints and therefore holds the promise of novel and fruitful approaches to knowledge representation and declarative programming. This article is only a first step. Much remains to be done, among others:

– Generalising the model theory of this article to universes of all kinds.
– Giving this article’s logic a unification and a proof calculus, preferably a resolution calculus, and investigating their decidability and completeness. First investigations point to an homomorphism between this article’s logic and first-order logic. The author therefore conjectures that the afore-mentioned unification and resolution calculus are similar to those of first-order logic.
– Specifying a program syntax and an immediate consequence operator based on this article’s logic.
– Investigating how constructs such as modules and embedded implications [38] can be expressed in programs built on this article’s logic.
– Investigating how this article’s logic relates to Gottlob Frege’s Begriffsschrift [20].

Acknowledgments. The author is thankful to Antonius Weinziel and Bob Kowalski for fruitful discussions on the subject of this article and to anonymous reviewers for useful hints.

References

9. Cervesato, I., Rossi, G.: Logic meta-programming facilities in 'LOG. Research 
Showcase @ CMU 6-1992, Carnegie Mellon University, School of Computer Sci-
ence, Computer Science Department (1992)
12. Costantini, S.: Computational Logic: Logic Programming and Beyond (Festschrift 
in honour of Robert Kowalski), LNCS, vol. 2408, chap. Meta-reasoning: A Survey, 
pp. 254–288. Springer-Verlag
nommendation, W3C (2014)
15. Damas, L.: Type Assignment in Programming Languages. Report number cst-33-
85, University of Edinburgh (1985)
16. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Pro-
ceedings of the 9th Symposium on Principles of Programming Languages (POPL). 
programming: A short comparative study. In: Proceedings of the IJCAI'95 Work-
shop on Reflection and Metalevel Architectures and their Applications in AI. pp. 
(2015)
19. Feferman, S.: The Oxford Handbook of Philosophy of Mathematics and Logic, 
des reinen Denkens. Verlag von Louis Nebert, Halle (1879)
Logic. Tudor, New York (1944)
the ACM (JACM) 40(1), 143–184 (1993)
23. Higgins, C.P.: On the declarative and procedural semantics of definite metalogic 
M.H. (eds.) Meta-Programming in Logic Programming, Workshop on Meta-
26. Hindley, R.J.: The principal type-scheme of an object in combinatory logic. Trans-
actions of the American Mathematical Society 146, 29–60 (1969)
Ph.D. thesis, Jennings Computing Center, Case Western Reserve University (1972)
28. Jiang, Y.: Ambivalent logic as the semantic basis of logic programming. In: Hen-
teryck, P.V. (ed.) Proceedings of the International Conference on Logic Program-
An Abstract Machine for Push Bottom-Up Evaluation with Declarative Output

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
brass@informatik.uni-halle.de

Abstract. The Push Method for Bottom-Up Evaluation in deductive databases was previously defined as a translation from Datalog to C++. Performance tests on some benchmarks from the OpenRuleBench collection gave very encouraging results. However, most of the systems used for comparison compile the query into code of an abstract machine and then use an emulator for this code. Therefore, runtimes cannot be directly compared. In this paper, we propose an abstract machine for bottom-up evaluation of Datalog based on the Push Method. This also helps to clarify some optimizations we previously expected from the C++ compiler. Since the interpreted code of the abstract machine must do something useful “standalone”, we also consider declarative output with templates.

1 Introduction

The database language SQL is a very successful declarative language, but usually only parts of applications are developed in SQL, the rest is written in a standard language like Java or PHP. The purpose of deductive databases is to increase the declaratively specified part of an application (ideally to 100% for many applications). Declarative languages have important advantages:

- Programs are compact (shorter than equivalent programs in classical languages), thus program development is faster.
- The language is relatively simple, therefore it can be used also by non-experts (e.g., not everybody using SQL is a professional programmer).
- The language has a mathematical precise semantics (usually based on logic), which makes programs easier to verify.
- The language is not tied to a specific execution model, thus it is easier to execute on new computing platforms, such as multicore processors or massively parallel clouds.

One reason for the current revival of Datalog is that it is used also for applications which are not traditional database applications, such as static analysis for program code [14], cloud computing [10,16] and semantic web applications [7,13]. The commercial deductive database system LogicBlox [1] is probably successful because it offers many functions in an integrated system with only one language.
Heike Stephan and the author have developed the “Push” method for bottom-up evaluation of Datalog [3,5,6]. It applies the rules from body to head (right to left) as any form of bottom-up evaluation, but it immediately “pushes” a derived fact to other rules with matching body literals. In this way, the derived facts often do not have to be materialized, and temporary storage can be saved. It can be seen as an extreme form of seminaive evaluation that dates back to the PhD thesis of Heribert Schütz [15], see also [17]. However, we use partial evaluation and treat database predicates specially, and did performance evaluations with very promising results. In the last time, “pushing” tuples through relational algebra expressions has also become an attractive technique for standard databases [12]. It seems well suited for modern hardware because it keeps the actively used set of data small.

The Push Method was defined as a translation from Datalog to C++. Performance tests on some benchmarks from the OpenRuleBench collection [11] gave encouraging results. However, most of the systems used for comparison (XSB, YAP, DLV, HSQLDB) compile the query into code of an abstract machine (an extension of the WAM) and then use an emulator for this virtual machine. Since we compiled to native code, runtimes were not directly comparable. Experiences with compiling Prolog to machine code [8] suggest that this gives approximately a factor of 3 (the range in that paper was between 1.3 and 5.6). In a first test of an early prototype, our abstract machine was only 1.5 times worse than the compiler-based approach. This strengthens our previous performance claims.

Our performance comparison also did not contain the runtime of the C++ compiler (because the compilation result can be executed many times on different database states). For large benchmark programs such as the wine ontology, the compilation runtime is quite substantial. In general, during program development, when the program changes often and is executed only on small test data, it is preferable to reduce the compilation overhead.

Another advantage of the program execution with an abstract machine is that the user does not need to install a C++ compiler. In this way, also the interfacing with possibly different compilers is avoided.

Furthermore, in the last version of the Push Method with procedures [6], many optimizations were delegated to the C++ compiler. While this makes the method easy to understand and produces quite readable code, it reduces the understanding of the lower-level execution which contributes to the good performance results. Going down to the level of an abstract machine can help here.

The abstract machine could also be a step towards a direct translation to machine code based on the LLVM compiler infrastructure (in [12], performance improvements were noticed by compilation to LLVM code instead of C++).

So far we were concerned mainly with the computation of derived data (query results). We expected that the generated C++ program can be linked with a manually created main program that uses the computed data. If one uses an emulator for an abstract machine, however, it must be sufficiently self-contained to do something useful with the computed data. In this paper, we consider declarative output based on templates that we introduced in [4].
An Abstract Machine for Push Bottom-Up Evaluation

2

173

Input Language: An Example

The input language is basic Datalog, i.e. pure Prolog (Horn clauses) with only
constants and variables, but no function symbols. Obviously, the language should
be extended later (e.g., by adding negation, aggregation and arithmetic computations). But for the first performance tests to check the approach, this very basic
language is sufficient. As an example, consider the well-known transitive closure
program, which is one benchmark from the OpenRuleBench collection [11]:
tc(X, Y) :- par(X, Y).
tc(X, Z) :- par(X, Y), tc(Y, Z).
The query is tc(X,Y), i.e. all derivable facts should be computed. Other benchmarks check goal-directed computation, and we have also good results for the
query tc(1,X) by applying our SLDMagic transformation [2], but the focus in
this paper is on pure bottom-up evaluation.
The predicate par is a database predicate. The benchmark contains files with
e.g. 50 000 facts for this “parent” relation, such as par(1,2). In our approach
such predicates must be declared with argument types:
db par(int, int) facts ’par.dl’.
For our compiler-based approach, we have implemented a data loader for Datalog
facts. We plan to develop additional loaders for different file formats (e.g., CSV,
JSON, XML). The input to the program is made available by reading the data
files into main memory relations for the database predicates. Also command line
arguments or the data of an HTTP request could be represented in this form.
We use main-memory relations matching the required access patterns in the
rules. Therefore, in the example, the edge facts are stored two times:
– In a list par_ff that permits to iterate over all facts for the first rule (ff is
the corresponding binding pattern: “free, free”).
– In a multi-map (e.g. a hash table) par_fb for the literal par(X, Y) in the
second rule. This data structure permits to iterate over all X-values given a
value for Y (binding pattern “free, bound”). The Push Method will activate
the second rule when a new fact for the body literal tc(Y, Z) is found,
therefore we know a value for Y when we access par here.
This corresponds to a relation in a classical database with an index over the
second column (however, the index usually contains ROWIDs, which are pointers
into the main heap file for the relation, in our case it directly contains the data).
Output is defined by a series of “templates”. The idea is that one first computes the necessary data by means of Datalog rules and then instantiates templates with parameters to actually generate the output. One can also understand
the templates as special procedures that contain only printing commands, calls
to other templates, and accesses to data from the computed “answer predicates”.
A template definition starts with the name of the template and an optional
parameter list. This “template head” looks like a Prolog/Datalog literal with
only variables as arguments. It is followed by the “template body”, which is
written in “[...]”. This is a list consisting of


Text pieces written in ‘...’ (string constants),
parameters of the template,
calls to other templates possibly with values for the parameters in (...),
calls to other templates with an iterator query in the following form

\[
\text{template}(\text{Args})\langle\text{Sort}\rangle^{*}\text{Sep} \,:=\, \text{Lit}.
\]

This means that \text{template}(\text{Args}) is called once for each answer to the Dat-alog literal \text{Lit}. The literal can contain the template parameters, new variables, and constants. The variables can be used as arguments to the tem-plate call (in \text{Args}), and in the specification of the sort order of the template calls (\text{Sort}). This is a comma-separated list of sort criteria (similar to the \text{ORDER BY} clause of SQL). One can also optionally specify a separator string \text{Sep} that is inserted between each two instantiations of the template.

In the example, output is specified as follows (generation starts from “\text{main}”):

\[
\text{main}: [\text{html_begin}(\text{All Derivable tc-Facts'})
\text{ul_begin}
\text{result_item}(X, Y)<X, Y> := \text{tc}(X, Y).
\text{ul_end}
\text{html_end}
].
\text{result_item}(X, Y): [\text{li_begin}
X ', ' Y
\text{li_end}
].
\]

It contains calls to six templates of a small HTML library (see below). The third call contains the loop over the answers and produces a sorted list.

There is also a “verbatim mode”, that starts and ends with “[|”. In this mode, all characters except “[|” and “|” (and a newline immediately after the opening “[|”) are copied literally to output. With “[|” one can nest the interpreted “code mode” within the verbatim mode (e.g., for inserting a parameter value).

\[
\text{html_begin}(\text{Title}): [|<html><head><title>[Title]</title></head><body><h1>[Title]</h1></body></html> | ].
\text{ul_begin}: [|\text{ul} | ].
\text{ul_end}: [|\text{ul} | ].
\text{li_begin}: [|\text{li} | ].
\text{li_end}: [|\text{li} | ].
\]
3 The Push Method: Explanation with Procedures

The idea of the Push Method is that the producer of derived facts has the control and actively “pushes” these facts to the consumer (rules in which the produced facts match body literals). In [3], we contrasted it with the “Pull” method where the consumer fetches the next fact when needed. This open-next-close interface of the query plan operators is also known as “Vulcano-style” [9] and is very common in relational database systems. However, in the last time, push methods have also been used successfully in standard databases [12].

In [3,5], we defined the Push Method for bottom-up evaluation of Datalog relatively low-level, managed our own stack and used C++ basically as a portable assembler. Then it turned out that a high-level version with procedures worked more or less equally fast for several tested benchmarks [6]. Current compilers are able to do many of the optimizations that were explicit in the low-level version. The version with procedures is actually very similar to an approach proposed by Heribert Schütz in his PhD thesis [15]. However, at that time, the results of first performance tests were not very encouraging. The approaches differ also in the details, for instance, we treat the database predicates specially.

The idea of the method is quite simple: One creates a procedure for each derived predicate $p$ that is called whenever a fact $p(c_1, \ldots, c_n)$ is derived. The task of the procedure call is to make sure that all rule instances with $p(c_1, \ldots, c_n)$ in the body are eventually applied. This is simple for rules that contain only a single body literal with a derived predicate (“linear rules”). Since the complete relations for the other (database) body literals are known, the necessary joins, selections and projections can be immediately done in order to perform the procedure calls corresponding to the head of the rule. For instance, the procedure for the “transitive closure” program from Section 2, looks as follows:

```c
void tc(int c1, int c2) {
    // Is this fact a duplicate?
    if(!tc_bb.insert(c1, c2)) // Set data structure
        return;

    // This is the query predicate, store answer:
    tc_ff.insert(c1, c2); // List data structure

    // Rule tc(X, Z) :- par(X, Y), tc(Y, Z):
    int Y = c1;
    int Z = c2;
    cur_1_1_c lit_1(&par_fb); // iterator over multimap
    lit_1.open(Y);
    while(lit_1.fetch()) { // Loop over all X with par(X,Y)
        int X = lit_1.out_1(); // First&only output column
        tc(X, Z);
    }
}
```
`cur_1_1_c` is a class for cursors/iterators over the result of a lookup in multimaps from rows with one column to rows with one column, generated from a template class for general multimap cursors.

The procedure for a predicate \( p \) contains one code block for each rule that contains the predicate \( p \) in the body (in this case, it is only one rule).

For “complex rules” that have more than one body literal with a derived predicate, temporary storage of derived facts seems unavoidable. While for special cases, optimizations are possible, in general one creates a temporary table for each body literal with a derived fact. When a rule is activated for a new fact for a specific body literal, one applies all rule instances with this fact and the previously derived facts stored in the temporary tables for the other body literals. As explained in [15], this can be seen as an extreme form of seminaive evaluation, where the “delta” consists of a single fact.

Finally, there is a procedure “start” that applies all rules without derived predicates in the body. Since the database predicates are loaded to main memory before the evaluation of rules starts, this is simple.

```c
void start() {
    // Rule tc(X, Y) :- par(X, Y):
    cur_2_c lit_1(&par_ff); // iterator over par-list
    lit_1.open();
    while(lit_1.fetch()) { // Loop over (X,Y) with par(X,Y)
        int X = lit_1.col_1(); // First column of current row
        int Y = lit_1.col_2(); // Second column
        tc(X, Y);
    }
    lit_1.close();
}
```

4 Memory Areas of the Bottom-Up Abstract Machine

4.1 String Memory

Names of relations, output texts, and string constants in the given rules must be stored. All arguments of machine instructions are integers, therefore string constants are mapped to unique integers.

Since facts for the database predicates are loaded at the beginning of program execution, these mappings are used and extended at runtime. One can define several domains, and assign each column of a database predicate to one domain. This leads to small, sequential numbers for each domain. Values in different domains are not comparable.

There is a different map data structure for each string domain. For large domains, we use an efficient implementation of a radix tree, for small domains, a simple fixed-size hash table. Of course, the inverse mapping (from integers back to strings) must also be supported, so that query results can be printed.

There is also a system string table for texts to be printed, where uniqueness is not important. This supports only the mapping from numbers to strings.
4.2 Relations

As explained above, relations are used for storing extensions of database predicates. Currently, all columns have type `int`, strings are mapped to integers as explained above. Later, we will have to add `float` or `double` values. Also 8-bit and 16-bit integers could lead to more compact data. But in this first version, all data are standard `int` values.

Each relation data structure supports a specific binding pattern (bound/free, i.e. input/output arguments). For instance, lists correspond to “all arguments free” binding pattern and support only a “full table scan”. Sets correspond to the “all arguments bound” binding pattern and support only an element test. Multimaps have bound and free arguments: One can loop over values for the free arguments given values for the bound arguments (we do not have keys yet, otherwise we might know that there is only a single tuple for given values of the input arguments).

We assume that the user specifies for which derived predicates duplicates should be detected (similar to a “table declaration” e.g. in XSB). This must be done at least once in each recursive cycle in order to guarantee termination. There is a set data structure for each selected predicate.

Finally, the query predicate or the predicates used in output templates must also be stored in relations. While for a classical query predicate, a list data structure suffices, output templates can use a specific binding pattern, and might also require sorted output of the selected rows. Our plan is to use tree data structures for these output predicates, so that the tuples are already stored in the order in which they are required. This might also be useful for doing merge joins. Currently we only have index joins.

Note that the same predicate with the same extension can be stored in different relation data structures, if there are body literals that require different access structures.

Relations are identified in the program by relation IDs (small numbers).

4.3 Load Specifications

We have implemented a loader for data files formatted as Datalog facts. The loader needs the file names and the set of predicates to be expected in each file. For predicates, the list of argument types is needed, and the domain (string table) to be used if the argument is of type string.

Furthermore, each predicate is linked to a set of relation data structures, in which the loaded tuples are inserted. For each such relation, an extended binding pattern is specified, which defines the mapping of the loaded data tuple to an entry in the relation. There are five options for each argument $i$, most of which have an integer parameter value $j$:

- Store argument $i$ of the loaded fact in input column $j$ of the relation data structure (this corresponds to the “bound” case).
- Store argument $i$ of the loaded fact in output column $j$ of the relation data structure (this corresponds to the “free” case”).
Check that argument $i$ of the loaded fact has value $j$. Otherwise, the fact is not stored. This corresponds to the case that the body literal to be supported by the relation contains constant $j$ (or a string mapped to $j$). Thus, it is possible to do the selection already when the data is loaded. For instance, in the DBLP benchmark [11], this is a very useful feature.

Check that argument $i$ of the loaded fact has the same value as argument $j$. Otherwise, the fact is not stored. This corresponds to multiple occurrences of the same variable in a body literal.

Do nothing with argument $i$. This corresponds to an anonymous variable.

In the transitive closure example, loaded par-facts are stored

- once in a list par_ff (lists have only output columns: argument 1 is stored in output column 1, argument 2 stored in output column 2), and
- once in a multimap par_fb (with argument 1 stored in output column 1, and argument 2 stored in input column 1).

### 4.4 Variables/Registers

Data values during the computation are stored in variables (registers of the abstract machine). Each variable can store a single int value. Also arguments of procedures are passed in these variables. Variables are identified by a single (quite small) integer value. In the transitive closure example, only two variables are used (for the two arguments of the tc procedure). A program must define how many variables it is going to use.

### 4.5 Cursors

Cursors are data structures used for iterating over the result of a relation access. If the relation is a list, the cursor supports a loop over all elements of the list. If the relation is a multimap, the cursor supports to loop over the result values for a given tuple of input values.

### 4.6 Stack

Values of type int can be pushed on the stack of the abstract machine. The stack is used for saving the following data:

- Return addresses for procedure calls. In this case, the int value is an address in the code area (the next instruction to be executed after the procedure call).
- Saved values of variables: For recursive procedure calls, it might be necessary to save the value of a variable, and restore the old value later from the stack.
- Saved positions of cursors: For recursive procedure calls, it might be necessary to save the current state of a cursor on the stack, and restore the cursor after the procedure call to that state. This is not always a single integer (the position in the list), e.g., it might also include the number of elements in the list: If the cursor is used for looping over a set of derived facts for a complex rule, it is required that insertions do not change the set of tuples over which the cursor runs.
4.7 Code (Machine Instructions)

Of course, the abstract machine also contains a storage area for machine instructions. In the current prototype, this is an array of type `unsigned char`. We try to keep the code compact, therefore the instruction length is not a multiple of 32-bit integers. Each instruction has an 8-bit opcode, and then arguments as needed for the instruction. In the current (very first, experimental) prototype, the longest instruction contains an 8-bit cursor ID, a 16-bit address, and an 8-bit variable number. It is clear that this will not be sufficient for larger programs. However, in the interest of compact code (which is more cache-friendly), there probably will be short and long forms of instructions.

4.8 Instruction Pointer

As any CPU, the abstract machine has an instruction pointer (program counter) that contains the address of the next instruction to execute.

4.9 Error Indicator

There is a boolean variable (a flag) that is set when an error is detected, e.g. a stack overflow or a failed insertion into a relation due to insufficient memory. In this case, program execution terminates at the next instruction.

5 Instructions of the Bottom-Up Abstract Machine

5.1 NULL

There is a `NULL` (no op) instruction that does nothing. It could be used to fill space in the code area when alignment of arguments, e.g. on 32-bit boundaries, becomes interesting.

5.2 HALT

The `HALT` instruction finishes program execution. It is written at the end of the main program (procedure `start`).

5.3 Procedure Calls: CALL and RETURN

The instruction `CALL` has an argument for the address of the called procedure. It pushes the instruction pointer on the stack (which has already been incremented to point to the next instruction after the `CALL`), and jumps to the given address (by setting the instruction pointer).

`RETURN` sets the instruction pointer to the address on top of the stack (and pops that address).
5.4 Duplicate Check

The DUPCHECK instruction has two arguments: A set ID $s$ and a variable number $v$. The set data structure has a method for getting the number of columns $n$. Then the tuple stored in variables $v$, $v + 1$, $\ldots$, $v + n - 1$ is inserted into the set. If this insertion fails, because the tuple is already contained in the set, the machine implicitly performs a RETURN, i.e. jumps to the topmost address on the stack. We try to keep the code compact, so a single instruction corresponds to the following code from the tc-procedure shown above:

```c
// Is this fact a duplicate?
if(!tc_bb.insert(c1, c2)) // Set data structure
    return;
```

The larger the granularity of the single instructions, the smaller is the overhead for interpretation.

We also added special instructions for small numbers of columns (currently 1 and 2). This permits to have the constants compiled into the code and use a series of instructions instead of a loop ("loop unrolling").

Requiring that the argument values are stored in consecutive variables means that it might be necessary to copy values from variables at non-consecutive numbers to a fresh set of variables. The compiler can of course try to pass arguments to predicate procedures in consecutive variables, but this is not always possible without copying values (which we want to avoid). Therefore, we will add a DUPCHECK instruction with a list of variable numbers (corresponding to the number of columns in the set). While the copying to an internal array must still be done (to construct the tuple for the insert operation), the code is more compact than with a series of copy operations. Furthermore, the intention of the instruction in the assembler program is clearer.

If we later do inlining of procedures, a duplicate check instruction will be needed that does a jump instead of the procedure return.

5.5 Saving and Restoring Values of Variables

Arguments are passed to procedures in variables (which are global storage locations), not on the stack. With the methods explained in [5], one can save a lot of copying in this way. However, sometimes one has to choose between copying or generating several procedures for the same predicate with different variables for the arguments. We did a partial evaluation that also handles the case that sometimes constants are known for arguments. Compilers that do inlining of procedure calls and copy propagation also might generate code for the same procedure with arguments in different storage locations.

If a procedure uses a variable that might contain a value that is still needed by the caller (or possibly an indirect caller), the variable must be saved and later restored. Variables can be assigned in such a way that this happens only in recursive procedures. Because not all arguments necessarily change from one recursive call to the next, it is better to save only what is actually needed than
to allocate everything on the stack. E.g., in the transitive closure example, the second argument of \texttt{tc} is not changed from the body to the head. It might be helpful to write the recursive rule as

\[
\text{tc(X\_new, Y)} \leftarrow \text{par(X\_new, X)}, \text{tc(X, Y)}.
\]

Therefore, no copying is done for the variable corresponding to the second argument (in this example, the tail recursion optimization of standard Prolog implementations would basically do the same, although the rule is applied in the opposite direction).

The instruction \texttt{SAVE\_VAR} pushes the value of a variable on the stack, and the instruction \texttt{RESTORE\_VAR} pops it again into the variable. Both have a variable number as argument.

### 5.6 Saving and Restoring Cursor States

In the same way, the current state of a cursor must be saved before it is opened if it might already be open in a procedure invocation somewhere up in the call tree. It seemed better to use a global cursor and push only its position on the stack, because cursors can be quite big objects that contain parts that do not depend on the current position (the link to the relation) and parts that are redundant for speeding up access (derived from the current position).

We use different instructions for different cursor types. E.g. \texttt{SAVE\_MMAP\_1\_1} calls the \texttt{push}-method of a cursor class for multimaps from one column to one column (remember that we treat common cases specially). We try to avoid virtual methods because of their overhead. However, there will also be a \texttt{SAVE\_CURSOR} instruction that looks up the type of the cursor at runtime and does the corresponding type cast. In any case, the current position of the cursor is pushed on the stack (plus other state-dependent data, like the current length of the list). The corresponding instruction \texttt{RESTORE\_*} restores the cursor to the previously saved state. All these instructions have a single parameter for the cursor ID.

### 5.7 Instructions for Loops Over Cursors

A very common operation is to loop over tuples of a relation with a cursor. E.g., for the first rule \texttt{tc(X, Y)} \leftarrow \texttt{par(X, Y)} in the example, we must loop over all tuples in the \texttt{par\_ff} relation. Such loops have a \texttt{LOOP\_*} instruction at the beginning, and an \texttt{END\_LOOP\_*} instruction at the end. We use again specific instructions for common cases, so in the example the instructions are \texttt{LOOP\_LIST\_2} and \texttt{END\_LOOP\_LIST\_2} (the relation is a list of tuples with two columns). These instructions have two parameters: An ID of the cursor (which is already linked to the relation) and a code address.

The instruction at the start of the loop opens the cursor and fetches the first tuple. If there is none, it closes the cursor, and jumps to the address, which should be the instruction just after the loop.

The instruction at the end of the loop fetches the next tuple. If this is successful, it jumps to the given address, which should be the first instruction of
the loop body. If there is no further tuple, the instruction closes the cursor, and control passes to the next instruction (after the loop). We try to do the entire loop control in one instruction to reduce the interpretation overhead.

The loop opening instructions for multimaps have an additional parameter for the input value (see the discussion about input tuples for the duplicate check in Subsection 5.4).

5.8 Instructions for Accessing Values from Cursors

In the body of a loop controlled by a cursor, one obviously needs to get data values of the current tuple. This is done by means of GET_\_* instructions. There are again specialized instructions for common cases, e.g. GET_LIST_2_COL_1 to access the value of the first column of a cursor over a list with two columns. This instruction has two parameters: The ID of the cursor and the number of the variable into which the value should be stored.

There is also a general GET instruction that can be applied to any type of cursor and has an additional parameter for the column number.

5.9 Conditions, Jumps

The body literal that is matched with the derived literal of the procedure invocation might contain constants or the same variable in different arguments. In these cases, we must check whether the rule can be applied to the derived literal. For this purpose, IF-instructions (conditional jumps) are needed.

The instruction IF_VAR_IS has three arguments: the number \(v\) of a variable, a code address \(a\), and a data value \(n\). If variable \(v\) contains value \(n\), execution continues normally with the next instruction. Otherwise, control jumps to the instruction at address \(a\).

The instruction IF_VAR_EQ is similar, but compares the values of two variables. The jump is done if they are distinct (i.e. the following code block is executed if they are equal).

There is also a GOTO instruction for an unconditional jump.

5.10 Copying Variables and Assigning Values to Variables

The head literal of a rule might contain constants. If the corresponding procedure has a parameter for the argument, we need to assign this constant value to the corresponding variable. This is done by the ASSIGN instruction, which has a parameter for the variable and a parameter for an integer constant. Note that it is possible to create a specialized procedure for a predicate that handles only the case with this specific constant for the selected predicate argument. We formalized this kind of partial evaluation with the notion of “fact types” in [5]: A fact type consists of a predicate and for each argument:

– a storage location (i.e. variable/register number in our case), or
– a constant.

The instruction COPY copies the value of one variable to another variable.
5.11 Inserting Tuples Into Relations

In order to store tuples in result relations or temporary tables for complex rules, there is an `INSERT` instruction. Again, there are different variants for special cases, e.g. `INSERT_LIST_2` inserts a tuple into a list of tuples with two columns. The relation (in this case, a list) is specified with a relation ID as first parameter. The second parameter is the number of the variable with the value for the first column, the second column value must be in the following variable. See Subsection 5.4 for a discussion of alternatives for the specification of tuples.

5.12 Output

Templates are translated to special procedures that contain output instructions, calls to other templates, and loops over result relations. A minimal set of output instructions is:

- `PRINT_TEXT(n)`: This prints string $n$ from the first (system) string table.
- `PRINT_STR(s,v)`: This prints a string from domain (string table) with ID $s$, where the string number is stored in variable $v$.
- `PRINT_INT(v)`: This prints the value of variable $v$ as an integer.

6 Example

In this section, we show the instructions of the abstract machine for the transitive closure example from Section 2:

\[
\begin{align*}
  tc(X, Y) & : = \text{par}(X, Y). \\
  tc(X, Z) & : = \text{par}(X, Y), \text{tc}(Y, Z).
\end{align*}
\]

The shown instructions run in our first (still experimental and not quite finished) prototype. This prototype does not have output templates yet, therefore we show a variant that simply inserts the derived tuples into a list (as we already did in Section 3). This also makes the runtime more comparable with the OpenRuleBench TCFF benchmark, which does not contain output. The example uses the following relations:

<table>
<thead>
<tr>
<th>ID</th>
<th>Relation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>par_ff</td>
<td>Use of par in first rule</td>
</tr>
<tr>
<td>1</td>
<td>par_fb</td>
<td>Use of par in second rule</td>
</tr>
<tr>
<td>2</td>
<td>tc_bb</td>
<td>For duplicate check</td>
</tr>
<tr>
<td>3</td>
<td>tc_ff</td>
<td>Result</td>
</tr>
</tbody>
</table>

Two cursors are used (in general, there might be several cursors over the same relation, but in this example, there is only one cursor for each of the `par`-relations):

<table>
<thead>
<tr>
<th>ID</th>
<th>Relation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>par_ff</td>
<td><code>par(X,Y)</code> in first rule</td>
</tr>
<tr>
<td>1</td>
<td>par_fb</td>
<td><code>par(X,Y)</code> in second rule</td>
</tr>
</tbody>
</table>
The program code for the tc example consists of 17 instructions stored in 48 bytes:

```
// Procedure start: tc(X, Y) :- par(X, Y).
0: LOOP_LIST_2(0, 17)  // Loop over par_ff, if empty goto 17
4: GET_LIST_2_COL_1(0, 0)  // Var[0] = X from par cursor (ID 0)
7: GET_LIST_2_COL_2(0, 1)  // Var[1] = Y value from par cursor
10: CALL(18)  // Call tc(Var[0],Var[1])
13: END_LOOP_LIST_2(0, 4)  // If next par-tuple exists goto 4
17: HALT  // End of "main" procedure start
// Procedure tc(Var[0],Var[1]): tc(X, Z) :- par(X, Y), tc(Y, Z).
18: DUPCHECK_2(2, 0)  // If (Var[0],Var[1]) in tc_bb: return
21: INSERT_LIST_2(3, 0)  // Store result tuple in tc_ff (ID 3)
24: SAVE_VAR(0)  // This invocation will change Var[0]
26: SAVE_MMAP_1_1_CUR(1)  // Cursor 1 will be used here
28: LOOP_MMAP_1_1(1, 43, 0)  // Loop over par(X,Y) given Y=Var[0]
33: GET_MMAP_1_1_OUT_1(1, 0)  // Store X with par(X,Y) in Var[0]
36: CALL(18)  // Recursive call: tc(Var[0],Var[1])
39: END_LOOP_MMAP_1_1(1, 33)  // If next tuple exists: goto 33
43: RESTORE_MMAP_1_1_CUR(1)  // Restore used cursor
45: RESTORE_VAR(0)  // Restore changed variable
47: RETURN  // End of procedure tc
```

7 Performance

In our previous performance comparisons of the Push Method with benchmarks from the from the OpenRuleBench suite [11], we have assumed that a factor of 3 must be attributed to the compilation to machine code. The results were still encouraging. Now the important question was of course whether an interpreted version of abstract machine code is not worse. Fortunately, the first benchmark we were able to execute with our still incomplete prototype, namely the transitive closure example, is only 1.5 times slower than the machine code version:

<table>
<thead>
<tr>
<th>System</th>
<th>Load</th>
<th>Execution</th>
<th>Total time</th>
<th>Factor</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push (Switch)</td>
<td>0.004s</td>
<td>1.145s</td>
<td>1.147s</td>
<td>1.0</td>
<td>23.535 MB</td>
</tr>
<tr>
<td>Push (Proc.)</td>
<td>0.004s</td>
<td>1.176s</td>
<td>1.177s</td>
<td>1.0</td>
<td>31.392 MB</td>
</tr>
<tr>
<td>Push (Abstr.M.)</td>
<td>0.004s</td>
<td>1.714s</td>
<td>1.713s</td>
<td>1.5</td>
<td>31.397 MB</td>
</tr>
<tr>
<td>Seminaive</td>
<td>0.004s</td>
<td>2.225s</td>
<td>2.227s</td>
<td>1.9</td>
<td>31.360 MB</td>
</tr>
<tr>
<td>XSB</td>
<td>0.239s</td>
<td>4.668s</td>
<td>5.103s</td>
<td>4.4</td>
<td>135.693 MB</td>
</tr>
<tr>
<td>YAP</td>
<td>0.240s</td>
<td>10.432s</td>
<td>10.840s</td>
<td>9.5</td>
<td>147.544 MB</td>
</tr>
<tr>
<td>DLV</td>
<td>(0.373s)</td>
<td>—</td>
<td>51.660s</td>
<td>45.0</td>
<td>513.748 MB</td>
</tr>
<tr>
<td>Soufflé (SQLite)</td>
<td>(0.113s)</td>
<td>—</td>
<td>11.240s</td>
<td>9.8</td>
<td>43.083 MB</td>
</tr>
<tr>
<td>(compiled)</td>
<td>(0.030s)</td>
<td>—</td>
<td>0.797s</td>
<td>0.7</td>
<td>3.867 MB</td>
</tr>
</tbody>
</table>

Transitive Closure Benchmark tc( _, _ ), 50 000 par-facts (cyclic) [11]
8 Conclusion

Both, the translation of Datalog to C++ (and possibly other languages in future), and the translation to code of an abstract machine, have advantages of their own.

For instance, a Datalog system with an abstract machine can work standalone, and does not need a C++ compiler. Lower level optimizations can be studied better with the abstract machine than with a generation of readable C++ code and relying on optimizations of the compiler for that language. Distribution of applications as abstract machine code is simpler. If a user trusts the abstract machine, he/she does not need to trust the code for a specific application, whereas binary code is inherently more dangerous.

Program execution with an abstract machine is slower than execution of native machine code. In order to keep the overhead small, we did the following:

- Arguments to procedures are passed in variables/registers, and not on the stack. Several versions of a procedure can be generated where the arguments are contained in different variables, or are known constants (in particular, there is no fixed register for the \(n\)-th argument). In this way, copying of values is reduced, which is an important characteristic of our Push Method.
- We tried to make the granularity of the instructions large, i.e. let a single instruction of the abstract machine do a lot of things to reduce the overhead of interpretation.
- We also introduced specialized versions of instructions for common cases.
- We tried to make the code of the abstract machine compact in order to better utilize the cache.

The current state of the project is reported at the following web address:

[http://www.informatik.uni-halle.de/~brass/push]

References


Extracting and Representing Entities from Open Sources of Information in the Agatha Project

Gonçalo Carnaz\(^2\)(orcid.org/0000-0001-8285-7005), Roy Bayot\(^2\)(orcid.org/0000-0002-1290-0239), Vitor Beires Nogueira\(^1\)^\(^2\)(orcid.org/0000-0002-0793-0003), Teresa Gonçalves\(^1\)^\(^2\)(orcid.org/0000-0002-1323-0249), and Paulo Quaresma\(^1\)^\(^2\)(orcid.org/0000-0002-5086-059X)

\(^1\) Informatics Departament, University of Évora, Portugal
\(^2\) LISP - Laboratory of Informatics, Systems and Parallelism, Portugal

Abstract. The Agatha project aims to develop an intelligent system that resorts to open sources (video, audio and text) of information for surveillance and crime control. Named-entity recognition combined with ontologies is the approach followed for the textual sources. This work describes the theoretical basis together with the system implementations for the text analysis component of the Agatha framework.

Keywords: Ontology, Eurovoc, Named-entity recognition, Open Information Sources, Criminal Investigation.

1 Introduction and Motivation

The new information and communication technologies have greatly facilitated the performance of criminal organizations. The high information present on the web and new social media, and the high number of users involved, have allowed the exchange information to go unnoticed. Criminal investigation and intelligence officers find it very difficult to keep up with the agility with which members of these organizations adopt new tools, and how they put these tools for their purposes. It is therefore essential to provide competent entities for the control and prevention of organized crime with adequate tools to deal with these recent realities.

The Agatha project aims to develop an intelligent system that resorts to open sources of information for surveillance and crime control. The information sources considered are social networks, forums, images, blog sphere information and other sources of information on the web, including audio and video sources.

The system will have the ability to analyze large amounts of information and extract implicit relationships, patterns and participants, among others, through
modules dedicated to the analysis of video and image, audio and text in several languages, composed by algorithms of crawling and data mining.

The remainder of this paper is organized as follows: in Section 2 we provide a brief description of project Agatha. Sections 3 and 4 provide a state of the art of named-entity recognition and knowledge representation in the legal domain. In Section 5 we describe our system implementation and finally, section 6 concludes and provides pointers for future work.

This work was funded by the Agatha Project SI&IDT n° 18022 and LISP - Laboratory of Informatics, Systems and Parallelism, Portugal.

2 The Agatha Project

The Agatha project aims to develop an intelligent system that resorts to open sources of information for surveillance and crime control. Its development is based on:

– Data acquisition: collection of information from open sources, through crawling algorithms, data mining and ETL (Extract, Transform and Load) tools;
– Video and image analysis: extraction of video file characteristics, automatic delimitation of moments and scenes, detection of patterns and their segmentation;
– Audio and voice analysis: development of technologies capable of automatically obtaining information from audio data sources collected by the data acquisition module;
– Biometric analysis: extraction of high-quality 3D face models from low quality video files, which will then be used to obtain 2D images of these faces for facial recognition applications with special focus for forensic use. Voice biometrics will be explored. Combining the two classifiers (face and voice identification) the inherent error of each identification technique is minimized;
– Multilingual text analysis: automatic translation for use of information in different languages as well as natural language processing (NLP) techniques to extract knowledge automatically.
– Classification and semantic segmentation: content segmentation and indexing will allow easier content navigation and cross-referencing between different databases and repositories.
– Database and Repositories: to store all the information coming from the different modules (audio, video, image, text, biometry). Properly indexation will facilitate its reference and/or correlation for the remaining functionalities of the solution to be developed.
– Data management, organization and visualization: define rules for the processing of user information requests, to retrieve information from the system
database, using semantic research methodologies, implemented through a Visual Analytics component that Visual analysis of large amounts of data.

The system involves the interaction of multiple parts. For instance, it would allow crime investigators to query news articles for relevant people or parties and then use the same platform to query videos that involve those people or parties that have been searched. However, this paper is focused on information derived from text.

The text comes from transcriptions of 24 hour news provided by another partner in the project called Voice Interaction. The idea is to use third party software to perform named entity recognition as an initial way to find possible entities of interest, and the results would later be used to query an ontology for other possible relations.

3 Information Extraction by Named-Entity Recognition

One of the many methods for information extraction is Named-Entity Recognition (NER) wherein given a body of text, elements of the text are identified into predefined categories such as names, places, time, organizations, and other classes. They could be defined from categories to sub-categories.

One of the first papers in the field is that of Lisa Rau [26] that built a system to extract and recognize company names using hand-crafted rules. This approach can also be seen in the work of Aberdeen et al. [1] where rule sequences were learned from small samples for named-entity recognition. These were not used stand alone but have been mixed with hand-crafted rules. Collins et al. [14] parsed 971,746 sentences of New York Times to find candidate patterns that were stored as spelling and context. These candidate patterns are checked against seed rules such as "if a spelling is all capitalized, then it is an organization." If it satisfies a rule, it is classified according to the rule. The contexts are accumulated with the most frequent rules considered as contextual rules.

Hand-crafted rules can be daunting to make because it requires a lot of time and expertise to develop. Recent techniques have shifted to statistical and machine learning methods. Examples of such are from Bikel et al. which uses a variant of the standard Hidden Markov Models [8]. Another example is that of Baluja et al. which used Decision Trees in conjunction with features such as part-of-speech tags, dictionaries, as well as capitalization and punctuation [5]. Borthwick et al. used Maximum Entropy Models with a variety of features - binary, lexical, section, dictionary, and external system features [9]. McCallum et al. used Conditional Random Fields as an alternative [23]. Another popular method is the use of Support Vector Machines as exemplified by the work of Takeuchi et al. for MUC-6 dataset in named-entity recognition [27].
Various named entity recognition software had been available for years. This includes GATE [15], OpenNLP \(^3\), MER \(^4\), OpeNER \(^5\), as well as Stanford Core NLP [22]. However, named entity recognition is not available for Portuguese in these software, which is one of main languages for the Agatha Project. There are named-entity recognition papers for Portuguese such as that of Miranda et al. [24] as well works of Freitas et al. [17]. Even with these works, the software developed is not readily available.

There are currently two softwares known to the authors that have support for named-entity recognition for Portuguese language. The first is called Polyglot \(^6\) which is Python package for natural language processing. The named-entity recognition implementation is described by a paper by Al-Rfou et al. [2]. Their approach relies on distributed word representations or word embeddings as opposed to other works that use NER human annotated datasets or resources such as treebanks or parallel corpora. The named-entity recognition supports identification of 3 classes - locations, organizations, and persons.

The second software is called Rembrandt by Nuno Cardoso [12]. The tool itself is comprised of multiple parts - the NER tool, RENOIR (a semantic query reformulation module), SASKIA (a knowledge base), an indexer to generate standard term index and semantic term index, and finally Lucene with GeoTemporal Extensions. The NER tool uses Wikipedia as a knowledge source wherein each named entity is mapped to a Wikipedia page and tries to analyze the document structure as well as other features of the page. The tool supports 9 main categories and 47 sub-categories. The main categories include person, organization, place, datetime, value, abstraction, event, thing, and masterpiece.

4 Knowledge Representation in the Legal Domain

The following paragraphs, describe the state of art related to: ontologies for the Legal Domain, SKOS and Thesaurus (focus on Eurovoc case), as a first step to found an answer for knowledge representation to entities extracted in section 5.

4.1 Ontologies for the Legal Domain

In 1994, [29] proposed the first step to a functional ontology of law, interconnecting and argue about ontological relations with AI and Law. Therefore, this ontology is a set of categories and subcategories of legal knowledge, giving a functional view about the legal system. In [3], it is proposed an ontology as a

\(^3\) http://opennlp.apache.org/
\(^4\) https://github.com/lasigeBioTM/MER
\(^5\) http://www.opener-project.eu/
\(^6\) http://polyglot.readthedocs.io/en/latest/
support for the representation of crime and/or criminal activity in Italy, aims to
be an attempt to solve some problems found in ongoing projects that were not
based on ontologies and that did not have a conceptual definition of a knowledge
base in order to achieve a conceptual framework for the various projects, they
added a domain knowledge and also draw the classes that allow the ontological
representation of the concept of crime, they defined a suspect/criminal - a person
who acts in a manner punished by criminal law, with a given behavior in a given
time interval - Event, and the penalty applied to the perpetrated act. It will thus
support the documents management as metadata, identifying and suggesting a
crime hypothesis to the Judge, and semantically map criminal laws using the
XML language. In this paper [6] authors proposed a FAQ system for judges in
their first appointment, supported by an ontology based on professional legal
knowledge, this system used ontologies to calculate semantic distances between
user queries and stored FAQs. In [10] proposed two ontologies applied to the
legal domain:

- the FOLaw (Functional Ontology for Law) was developed to represent knowl-
edge from viewpoints of social roles and legal function and it also describes
the relations among this knowledge;
- the LRI-Core supports the construction of structured legal domains, to allow
automatic indexing of legal texts, the criminal law of the Netherlands as an
example.

In [13], developed an ontological framework of legal documents and judicial cases
to be applied to countries, such as China. Authors performed a study of psycho-
logical compensation in medical disputes, observing the judicial cases and their
court decisions. In paper [16], authors give an overview of the DALOS project,
aims that legal decision-makers have control over the legal language, from Na-
tional to European level, supported by a knowledge organization system, with a
main requirement which is deal with lexical complexity at ontological level, and
for that, developed an domain ontology related to consumer protection; Addi-
tionally, the Leibniz Center for Law at University of Amsterdam, developed the
LKIF ontology [18] [19], which emerged as part of an architecture for information
systems for the legal domain, with two main requirements:

- Translation between legal base knowledge represented in different formats
and formalisms;
- Formal representation as part of an information system architecture.

Representing the LKIF ontology with three layers: top, middle and legal, with
15 modules that represents a set of related legal concepts. Consequently, this
ontology is a set of ontologies related to the legal domain:

- abstract concepts: are defined in five closely-related modules: top, place,
mereology, time and space time;
basic-level concepts: are distributed across four modules: process, role, action and expression;
– legal concepts: consist of legal action, legal-role and norms.

The authors of [28] described from the theoretical and practical aspects of the LOIS multilingual legal knowledge base, which contains legal terminology from National and European Legislation related to consumer law domain. The key role is to use semantic resources, like lexical concepts and formal ontologies to spread legal information, supported by multilingual semantic lexicon of law.

4.2 Thesaurus: the Eurovoc Case

In the Semantic Web context, vocabularies are used to define concepts (named also as terms) and relationships, used to describe and represent an field of interest, applied to terms classification that can be used for a particular application, characterized with relationships and defined constraints.

A thesaurus $\mathcal{E}$ could be defined as a set of terms $\mathcal{T}$, that describe a vocabulary $\mathcal{V}$ of a controlled indexing language, formally organized, with a set of relations $\mathcal{R}$, between those terms.

This subsection aims to describe the Eurovoc as a multilingual (23 EU languages) thesaurus that was especially built for information processing, regarding a wide spread of documents existing in the European Union Institutions. Currently, Eurovoc version, published in 30.06.2017, is 4.6. Managed by the Publication Office. The Eurovoc thesaurus thus comprises:

– descriptors, also known as Preferred terms, i.e. words or expressions which unambiguously designate the concepts of the domain covered by the thesaurus;
– non-descriptors, i.e. words or expressions used as synonyms or non-descriptors that are never assigned as index terms, they are used as access points in the thesaurus and guide the user towards the appropriate descriptor, with an instruction (Use For (UF), and the inverse USE);
– semantic relationships, i.e. relationships based on terms meaning, firstly between descriptors and non-descriptors and secondly between descriptors.

There is three kinds of semantic relations: Broader Term (BT), Narrower Term (NT) and Related Term (RT). The Scope Note (SN) relation is used to provide a definition of a term.

Eurovoc is accomplished and compiled in accordance with two standards of the International Standards Organization:

– ISO 2788-1986: Guidelines for the establishment and development of monolingual thesauri;

7 http://eurovoc.europa.eu/
Fig. 1. Thesaurus relations representation.

- **ISO 5964-1985**: Guidelines for the establishment and development of multilingual thesauri.

The fields: micro-thesauri descriptors, hierarchical relationships and associative relationships, are equivalent in all languages.

In paper [7], the authors analyzed the semantic scope of Eurovoc thesaurus, used for indexing and retrieval of legal information in the EU institutions and national parliaments, the ontology used is DALOS [16].

Some other works proposed a thesaurus ontology mapping [4] [21] [20], not directly related with Eurovoc thesaurus.

### 4.3 SKOS

The Simple Knowledge Organization System is defined by W3C as “a common data model for knowledge organization systems such as thesauri, classification schemes, subject heading systems and taxonomies. Using SKOS, a knowledge organization system can be expressed as machine-readable data. It can then be exchanged between computer applications and published in a machine-readable format in the Web”.

A new version was released in 18 August 2009, by the W3C, all documentation is available at - [https://www.w3.org/TR/2009/REC-skos-reference-20090818/#L1368](https://www.w3.org/TR/2009/REC-skos-reference-20090818/#L1368) - that can be used to perform SKOS mapping with a thesaurus or other lexical sources. Particularly, the W3C Semantic Web Best Practices and Deployment Working Group published and maintained SKOS Core, an RDF vocabulary to express the basic structure and content of concept schemes, presented in a thesauri, classification schemes, taxonomies, terminologies and other types of controlled vocabulary. Associated with SKOS Core and Eurovoc, a EuroVoc SKOS-Core distribution is released by EU Publications Office.

There are several works related with SKOS as a Knowledge Organization System such as [30] that proposed a method of thesauri conversion (IPSV, GTAA and MeSH) into SKOS RDF/OWL schema. The authors [25] proposed a method.

8 [https://www.w3.org/TR/skos-reference/](https://www.w3.org/TR/skos-reference/)
of publishing the AGROVOC thesaurus as Linked Data, using a downloadable version of the thesauri in SKOS format. In paper [11] the authors proposed a framework to convert thesauri into OWL2 ontologies, with a source thesaurus converted by a set of conversion rules to obtain the ontology, using SKOS to support concepts mapping.

5 Agatha Framework: NER Implementation

The current system implementation can handle an NER for Portuguese through a REST API. This involves Polyglot, Rembrandt, Flask, Nginx server, gunicorn application server, and MongoDB and is illustrated in Figure 2.

Flask is a Python microframework to build web applications. It allows to you to define the routes and REST protocols enabled so that an application can have an interface through the web. For instance, our application has 3 routes defined. The first is for the address agatha.xdi.uevora.pt with a GET method enabled. This allows a user to just query all the named-entities that were processed. The second is a route to agatha.xdi.uevora.pt but with a POST method enabled. And finally, agatha.xdi.uevora.pt/remb with a POST method enabled.

When an NER task is needed to be done on a text (which is a news transcription for our case), it should be structured as a JSON file and sent as an
attachment through a POST request. The URL in the POST request determines which NER tagger to use. Polyglot is used if it is sent to agatha.xdi.uevora.pt but Rembrandt is used if it is sent to agatha.xdi.uevora.pt/remb. When the POST request is issued, the nginx server forwards the request to the application server. We chose gunicorn as the application server because it works well with the Flask application. Nginx was retained because gunicorn is susceptible to denial of service attacks, according to the documentation.

When the route given in the POST request calls for Polyglot, the Flask application extracts the data to be processed and sends it to the NER submodule in Polyglot. The submodule then returns the extracted NER in JSON format and the Flask application stores it in the MongoDB. It then sends back the JSON result through the nginx server and to the one that issues the POST request.

When the route given in the POST request calls for Rembrandt, the Flask application also extracts the data to be processed and sends it to the Rembrandt submodule. This submodule makes system calls to the Rembrandt software and extracts the NERs from the output. It then sends these NERs back as a JSON file in the same way as indicated previously.

Storing the results in MongoDB was also done to find patterns of use in the future.

6 Conclusion and Future Work

In this paper we provided an overview of ontologies applied to the legal domain, the SKOS organization system and the Eurovoc thesaurus. We also presented the concept of Named-Entity Recognition for information extraction as well as some of the software available for the Portuguese language.

Moreover, we described the current implementation of the component of Agatha project responsible for the NER task.

Future work will focus on the representing the entities extracted from our implementation into a suitable knowledge representation system that integrates SKOS and Eurovoc thesaurus with a legal ontology.

It is also planned to implement the combined named-entity recognition with ontology querying as a REST API.

Finally, we also intend to test and evaluate this framework using as data set the transcription of audio taken from Portuguese news channels.

References


Functional and Logic Programming
Concolic Testing of Functional Logic Programs

Jan Rasmus Tikovsky
Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
jrt@informatik.uni-kiel.de

Abstract. In the last years, concolic testing, a technique combining concrete and symbolic execution for the automated generation of test cases, has gained increasing popularity. Concolic testing tools are initialized with expressions on concrete input data. But instead of just evaluating them, they additionally collect symbolic information along specific execution paths. This information can be used to systematically compute alternative inputs exploring yet unvisited paths. In this way, test cases can be generated covering all branches of a given program. The first concolic testing tools have been developed for imperative languages analyzing code at a very low level. Recently, there have been also some approaches investigating the concolic execution of declarative languages. In this work, we discuss the application of concolic testing to the functional logic language Curry. More precisely, we present ccti, a concolic interpreter which is adapted for the automated generation of test cases for both purely functional and non-deterministic programs.

1 Introduction

There are several methods to verify the correctness of programs. Among these formal program verification has the most significant relevance. But as proving the correctness of programs is a rather difficult and time consuming task, testing has become the most established approach to ensure the reliability of software. In fact, program testing itself became a wide area of research over the last decade resulting in various approaches.

In general, we distinguish between testing in the large and testing in the small. The former includes the testing of complete systems as well as the verification of interfaces between larger components, while the latter is directed to minor parts of programs like one module or even only a single function.

Furthermore, regarding the consideration of source code, software testing can be divided into two categories, namely black-box and glass-box testing. As the name implies, tests of the former category treat the software to be tested like a black box ignoring its concrete implementation completely and deducing test cases from specifications. Random testing and property-based testing, falls within this category. Property-based testing uses random input data to produce results which are then matched with previously specified properties. QuickCheck [5] for Haskell or QuviQ QuickCheck for Erlang are examples of property-based testing libraries for functional languages. But also purely logic languages like Prolog and functional logic languages like Curry provide tools for property testing, namely PrologCheck [2] and CurryCheck [11].

1 http://www.quviq.com/products/erlang-quickcheck/
Glass-box testing, on the other hand, works on the source code level. By the selection of input data, execution paths are followed through the code to determine appropriate outputs. Often this process is repeated until certain code coverage criteria are met. Since glass-box testing is a systematic approach, it is well-suited for automation. Examples of glass-box testing include symbolic execution. In this process, a program is interpreted using symbolic values for inputs instead of concrete data producing constraints on those symbols for all conditional branches in the program. Applying constraint solvers, these so called path constraints can be solved to compute actual input data driving execution along the associated path.

In the last years, a combination of concrete and symbolic execution, called concolic execution, has gained more and more popularity. We explain the basic idea behind concolic testing by the following example program.

```haskell
nenthElem [] _ = Nothing
nenthElem (x : xs) n | n == 0 = Just x
                   | n > 0 = nthElem xs (n - 1)
```

**List. 1.** Selection of the n-th element of a list

The listing shows the definition of a Curry function to select the n-th element of a polymorphic list. This definition distincts three different cases via pattern matching and guards: If the given list is empty, Nothing is returned. Considering a non-empty list either the first list element is returned or the function is called recursively depending on whether the index is 0 or a positive integer number.

The objective of automated testing tools is to find enough test cases to cover every distinct case of a function at least once. For the given example two test cases would be sufficient, i.e., one using an empty list and one using a list with at least two elements and an index greater than 0 and smaller than the length of the list.

For this purpose, concolic testing tools start with some concrete inputs. While evaluating a function call with these inputs, concolic testing tools additionally collect symbolic information describing the branch decisions which are made along that execution path. These decisions are also denoted as path constraints, since input data has to satisfy them to drive execution along that path.

Concolic testing tools aim at negating such path constraints systematically and solving them in order to produce input data which drives evaluation along alternative execution paths. Repeating this process, test cases covering all program branches can be generated automatically.

For instance, during the concrete execution of “nthElem [42] 0”, we additionally consider the symbolic expression nthElem xs n with xs and n being symbolic variables. The concrete expression can be evaluated to Just 42 by applying the second rule of nthElem. During pattern matching, a branch decision is made constraining the symbolic variable xs to a non-empty list. Furthermore, the evaluation of the first guard of the second rule constrains the symbolic variable n to be equal to 0. By negating these constraints, we receive constraints associated with alternative execution paths. For example, the negation of the first path condition constrains xs to be the empty list, thus, driving execution along a different execution path, namely the one represented by the first rule of nthElem. This process can be repeated until all paths of the associated
symbolic execution tree have been visited. In that case all branches of \texttt{nthElem} are covered.

The first concolic testing tools were developed for imperative languages. Examples include DART [10] and CUTE [18] for C, and jCUTE [17] for Java. Recently, concolic execution has found its way into declarative programming languages. For the functional language Erlang there are two tools which apply a program instrumentation to collect symbolic information, namely [9] and [16]. Moreover, in [14] and [15] a method for concolic testing in Prolog is presented. Regarding the functional logic language Curry, Fischer and Kuchen [8] discuss an approach which uses narrowing to generate test cases from uninstantiated function arguments systematically.

In this work, we propose \texttt{ccti} (Curry Concolic Testing Interpreter), a tool for automated concolic execution of Curry programs. To the best of our knowledge, concolic testing so far has not been applied to functional logic programs. We present an augmented semantics for Curry’s simplified core language FlatCurry which enables the additional collection of symbolic information during concrete evaluation. This symbolic information is used to generate path constraints. By negating these constraints systematically and applying an SMT solver, namely Z3 [6], we produce input data directing the execution to yet unexplored program paths. Furthermore, we present a simple search strategy for the selection of the path constraint to be negated next.

Our work is based on approaches applying concolic testing to purely functional languages and demonstrates that some of the ideas proposed in these approaches can be applied to functional logic languages as well: For instance, we also use a simplified core language which facilitates the identification of program branches, and thus the collection of path constraints. Moreover, \texttt{ccti} provides a search strategy to explore alternative, yet unvisited execution paths which is very similar to the one presented in [9]. In contrast to the concolic testing tools for purely functional languages mentioned above, we use an interpretation- rather than an instrumentation-based approach. This is due to the fact that the combination of non-deterministic computations and sharing of common subexpressions in Curry complicates the implementation of a semantics-preserving code instrumentation. Contrary to the narrowing-based approach presented in [8], \texttt{ccti} enables the generation of test cases for programs including primitive types like integers or floats. While narrowing on those primitive types can only be applied by using alternative, data constructor-based representations of integers and floats, we can simply reuse their original representation by applying suitable theories of the SMT solver.

The rest of this paper is structured as follows: In Sect. 2 we describe the functional logic language Curry as well as its simplified core language FlatCurry. Sect. 3 gives a brief introduction to satisfiability modulo theories (SMT) problems and demonstrates the modelling and solving of such problems applying the Z3 solver. The general idea of concolic testing of FlatCurry programs is explained in Sect. 4. Afterwards, we present a variant of the natural semantics of FlatCurry programs augmented for concolic testing. In Sect. 5 we take a closer look at the search algorithm applied to investigate the symbolic information which has been collected during concolic execution. Sect. 6 presents parts of the implementation of \texttt{ccti}. Finally, we discuss the applicability of \texttt{ccti} considering some practical examples in Sect. 7 before we conclude in Sect. 8.
2 Curry

Curry is a declarative programming language integrating well-known features from functional programming, like higher-order functions and lazy evaluation, as well as elements of logic programming, like non-determinism and computations with partial information. We will give only a short overview here. For a detailed introduction we refer to [13].

Curry’s syntax is very similar to that of the functional programming language Haskell. Curry supports the declaration of algebraic data types via the keyword `data`. Identifiers of types and data constructors start with an uppercase letter, whereas variable and function names usually begin with a lowercase letter.

For instance, the `Maybe` type representing optional values in Curry which we used in List. 1 is defined as follows.

```haskell
data Maybe a = Nothing | Just a
```

Functions are defined via rules and pattern matching. In contrast to Haskell, Curry supports the definition of non-deterministic operations by specifying overlapping rules. The following listing shows the definition of an operation which inserts an element in a list at an arbitrary position.

```haskell
insertND x [] = [x]
insertND x (y:ys) = x : y : ys
insertND x (y:ys) = y : insertND x ys
```

Consider the expression "`insertND 42 [1,2]`". Evaluating this expression in a Curry system will yield three non-deterministic results, namely `[42,1,2]`, `[1,42,2]` and `[1,2,42]`. Rather than specifying overlapping rules, one can also use Curry’s `choice` operator “?" in order to define non-deterministic operations. The `choice` operator is predefined as follows

```haskell
x ? _ = x
_ ? y = y
```

Apart from non-deterministic operations Curry also enables computations with partial information by using free variables in expressions instead of standard input values.²

For instance, the expression “`let x free in not x`” is reduced to the results `{x=False} True and `{x=True} False by binding the free variable x appropriately.

Due to the support of non-determinism and partial data structures, Curry uses an alternative evaluation mechanism compared to Haskell, namely needed narrowing [3]. Basically, needed narrowing corresponds to lazy evaluation using unification instead of pattern matching for the passing of parameters. In case an argument of a function which is required for further evaluation contains a free variable, this variable is bound to a constructor term so that evaluation can continue.

Furthermore, there is a core language of Curry named FlatCurry which provides a simplified representation of programs. Due to its simplicity, it is common practice to implement analysis tools and transformations for FlatCurry rather than for full Curry. An abstract representation of the syntax of FlatCurry programs is depicted in Fig. 1

² Note that variables need to be explicitly declared as free.
Fig. 1. The FlatCurry representation of programs

where sequences of objects $o_1, \ldots, o_n$ are denoted by $\overline{o_n}$. For the sake of simplicity, we assume in the following that literals and literal pattern like numbers or characters are represented as nullary constructors and constructor pattern, respectively.

A FlatCurry program consists of a sequence of function definitions. Every function is specified by a single rule consisting of pairwise different variables $\overline{x_n}$ on its left-hand side and an expression on its right-hand side. Any pattern matching in the original Curry program has been made explicit by the use of case expressions.

All case expressions include a unique identifier $id$. Additionally, all local function declarations have been lifted to the top level in FlatCurry.

FlatCurry is not only the basis for the implementation of analysis tools but also for the description of Curry’s semantics. In Sect. 4 we consider the operational semantics of FlatCurry which was originally presented in [1], revised by Hanus and Peemöller in [12] and augmented by us for concolic testing.

3 Satisfiability Modulo Theories

In this section we give a brief overview of satisfiability modulo theories (SMT) and its dedicated solvers. Moreover, we present SMT-LIB, a library providing common standards and benchmarks for the comparison of SMT solvers.

3.1 General Overview

An SMT problem [7] is a decision problem which can be represented as first-order logic formulas containing special predicate symbols with additional interpretations. These interpretations are predefined by so-called theories which can be applied during modelling and solving of SMT problems. For instance, there are theories for integer and real arithmetic, but also for uninterpreted functions, arrays, bit-vectors and recursive datatypes. Hence, an SMT instance is a generalization of a Boolean satisfiability (SAT) instance including additional predicates from various underlying theories.

Note that this identifier is not part of the original FlatCurry syntax, but it is helpful to associate symbolic information with a specific case expression and can be added by a simple transformation.

\[
P := \overline{D_m} \quad \text{(program)} \\
D := f(\overline{x_n}) = e \quad \text{(defined function)} \\
e := x \quad \text{(variable)} \\
| c(\overline{e_k}) \quad \text{(constructor call)} \\
| f(\overline{e_k}) \quad \text{(function call)} \\
| \text{let } \overline{x_n} = \overline{e_n} \text{ in } e \quad \text{(recursive let binding)} \\
| \text{let } \overline{x_n} \text{ free in } e \quad \text{(free variables)} \\
| e_1 \ ? \ e_2 \quad \text{(non-deterministic choice)} \\
| \text{case } id \text{ of } \{ p_i \mapsto e_i \} \quad \text{(case expression, } p_i \text{ pairwise different)} \\
p := c(\overline{x_n}) \quad \text{(constructor pattern)}
\]
There is a wide range of applications for SMT solving, for example software verification, constraint solving, planning and software testing - to mention only a few. There are also many SMT solvers implementing various APIs and providing different built-in theories. In this work, we focus on the Z3\textsuperscript{4} solver developed by Microsoft \cite{6}. Z3 is an efficient, open-source SMT solver supporting the SMT-LIB standard. We primarily chose Z3, because, in addition to basic types like integers, it provides a theory for the definition of algebraic data types which have just recently been added to the SMT-LIB standard.

### 3.2 SMT-LIB

As mentioned above, SMT-LIB\textsuperscript{5} is a library which aims at facilitating research in the SMT sector. Among other things, it provides descriptions of background theories, benchmarks for the comparison of SMT solvers, as well as a standardized input and output language for such solvers \cite{4}. When we refer to SMT-LIB in the following, this input and output language is meant.

An SMT-LIB script is a sequence of commands describing an SMT problem. For instance, the declare-const command declares a constant of given type (respectively sort). Z3 internally maintains a stack of declarations and formulas provided by the user. In order to add a formula to this stack, we can use the assert command. As mentioned before, a formula is a first-order formula including predicate symbols like $< \text{ or } +$ with additional interpretations. With the command check-sat we can ask the solver to check the satisfiability of the current formulas on the stack. If the formulas are satisfiable, Z3 will answer with sat, otherwise with unsat. In case Z3 can not determine the satisfiability of a formula, it will return unknown. If a formula is satisfiable, i.e., there is an interpretation for the user-declared constants, which makes the asserted formulas true, then we can retrieve the whole interpretation or only single bindings using the commands get-model and get-value, respectively.

In addition to these commands, Z3 also supports the declaration of polymorphic algebraic data types via the command declare-datatype.\textsuperscript{6} After their declaration, the type and value constructors can be used like any predefined sort or value.

We conclude this section with a small SMT-LIB script demonstrating some of the commands above. Reconsidering the example from the introduction with the initial call "nthElem [42] 0", we demonstrate the representation of path constraints in SMT-LIB. As mentioned above, during concolic execution we do not only consider the concrete call but also a symbolic one, namely "nthElem xs n". While evaluating the given expression, the variables from the symbolic call are constrained by the branch decisions made along the concrete execution path. The path constraints for the given call can be represented by the formula $xs = y : ys \land n = 0$, where $xs$, $y$, $ys$ and $n$ are symbolic variables and $:$ is the constructor for non-empty lists. In order to compute input data which drives the evaluation along an alternative execution path, we can negate

\textsuperscript{4} https://github.com/Z3Prover/z3

\textsuperscript{5} http://smtlib.cs.uiowa.edu/index.shtml

\textsuperscript{6} Note that we present the syntax of the declare-datatype command as it is specified by the SMT-LIB standard version 2.6. At the moment of writing, version 2.6 had just been released and Z3 still used an alternative syntax for the declaration of data types.
a particular path constraint and try to solve the resulting formula with the SMT solver. For instance, we can negate the first constraint of the example above and represent the resulting formula in SMT-LIB as follows.

```
1 (declare-datatype List (par (A) ((nil)
2 (cons (head A) (tail (List A))))))
3 (declare-const xs (List Int))
4 (declare-const n Int)
5 (assert (and (forall ((y Int) (ys (List Int)))
6 (not (= xs (cons y ys))))
7 (= n 0)))
```

List 2. Representation of path constraints in SMT-LIB

The first two lines show the declaration of a type representing polymorphic lists in SMT-LIB. In contrast to Curry, data type declarations in SMT-LIB also include selector definitions for the arguments of constructors like head. Lines 3 and 4 include the necessary constant declarations for the model. The SMT-LIB formula representing the negated path constraint is depicted in lines 5 and 6. Note that we need to universally quantify the arguments of the cons constructor in the formula in order to receive an alternative constructor binding for xs. Otherwise, Z3 will just bind xs to a non-empty list with more elements.

If we ask Z3 to check the satisfiability of this problem and return a binding for xs and n, if possible, it will yield sat and the answer ((xs nil) (n 0)).

4 Concolic Interpretation of FlatCurry Code

In this section we describe the concolic interpretation of FlatCurry programs. First, we explain at which points of evaluation symbolic information has to be traced and which information is required for the generation of path constraints. Then, we present an operational semantics for FlatCurry programs which enables the tracing of this information during evaluation.

4.1 Tracing of Symbolic Information

As we have sketched in the introductory example in Sect. 1, the basic idea of concolic testing is to evaluate a program using concrete input data and collect symbolic information at the same time. This symbolic information corresponds to the branch decisions made along a concrete execution path.

Recall that Curry programs use (overlapping) rules and pattern matching for case distinctions. In FlatCurry programs, overlapping rules and pattern matching have been made explicit by the use of non-deterministic choices and case expressions, respectively. As non-deterministic choices and case expressions are the only kinds of branches included in FlatCurry programs, these expressions are the ones of interest for the collection of symbolic information during evaluation.

First, we take a look at purely functional programs. Reconsider the introductory example in List. 1. In the following listing the FlatCurry representation of nthElem is depicted. Note that guards are transformed to case expressions with the respective conditions as arguments.
We assume that the concolic execution starts with the call “nthElem [42] 0”. The case expressions in the program demand their arguments to be evaluated to head normal form, so that a matching branch can be selected. Thus, first [42] is reduced to head normal form selecting the second branch of case₁. Next, the head normal form of \( n = 0 \) is computed selecting the first branch of case₂ and yielding the result \( \text{Just } 42 \).

In order to reproduce this specific execution path for \( \text{nthElem} \), we maintain a symbolic variable for every branch decision made along this path and store this variable together with the selected constructor. For branch decisions which involve comparison operators on numerical literals, e.g. \( n = 0 \), we save the constraint associated with the chosen branch. In addition, we always store the case identifier and the index of the selected branch. This information is needed during search to keep track of already visited branches. Hence, we receive the following symbolic trace for the given example:

\[
\begin{cases}
\text{case₁, } 2/2, \ x\text{sym₁, } ()
, \text{case₂, } 1/2, \ n\text{sym} = 0
\end{cases}.
\]

Before we conclude this subsection, we take a look at a non-deterministic program. Below the FlatCurry representation of \( \text{insertND} \) is shown.

\[
\text{insertND } x\text{x} = \text{case₁ } x\text{x of } \[] \rightarrow \text{x}
\ y:y\text{s} \rightarrow (x \ y \ y\text{s}) \ ? \ (y : \text{insertND } x\ y\text{s})
\]

We consider the call “\( \text{insertND } \text{True } [\text{False}] \)”, which evaluates to the non-deterministic results \( \text{[True, False]} \) and \( \text{[False, True]} \). Regarding the collection of symbolic information in such programs, two approaches are possible: On the one hand, we could generate traces including non-deterministic branch decisions. In that case we would receive a trace which selects the second branch of case₁ followed by a non-deterministic choice between an empty trace and one selecting the first branch of case₁ for the example considered above. On the other hand, we could encapsulate any non-determinism by constructing a search tree during evaluation. The non-deterministic choices occurring in the program would correspond to the branches of this tree and its leaves would include the various non-deterministic results as well as the respective symbolic trace. Afterwards, we could explore this search tree collecting all possible traces in a list.

Since the interpreter, on which we based the implementation of \( \text{ccti} \), already supported encapsulation of non-determinism, we chose the latter approach. Hence, the following list of traces is computed for the given example:

\[
\begin{align*}
\{ & \text{case₁, } 2/2, \ x\text{sym₁, } ()) \\
, & \text{case₁, } 2/2, \ x\text{sym₁, } ()), \text{case₁, } 1/2, \ x\text{sym₂}, \ () \\
\}
\]

\footnote{Note that in the actual implementation further information is collected which is required for the transformation of FlatCurry to SMT-LIB and vice versa.}
For programs including free variables in case expressions we apply narrowing during evaluation to consider all possible bindings for these variables and trace the branch decisions resulting from these bindings accordingly. Note that all traces resulting from a non-deterministic computation or a narrowed free variable are considered during the search for alternative execution paths.

4.2 Augmented Semantics for Concolic Execution

We conclude this section with a description of an augmented operational semantics for FlatCurry enabling the collection of symbolic information during evaluation.

The semantics for concolic execution presented below addresses normalized FlatCurry. During normalization of a FlatCurry program, constructor and function calls are flattened as well as case expressions. For this purpose, we introduce let bindings for the arguments of calls and case expressions, e.g. the function call “not False” is flattened to the expression “let x1 = False in not x1” where x1 is a fresh variable.

We use the following definitions and notations in the presentation of the semantics for concolic execution of normalized FlatCurry programs:

1. $\mathcal{V}$ is a set of variables.
2. $\mathcal{Exp}$ is a set of FlatCurry expressions.
3. The symbol “free” denotes a free variable.
4. A heap is a partial mapping from variables to either FlatCurry expressions or to the special symbol “free”: $\mathcal{Heap} = \mathcal{V} \rightarrow \{\text{free}\} \cup \mathcal{Exp}$
5. The empty heap is denoted by $\emptyset$.
6. $\Gamma[x]$ represents the value a variable $x$ is bound to in a heap $\Gamma$.
7. $\Gamma[x \mapsto e]$ corresponds to a heap $\Gamma'$ with $\Gamma'[x] = e$ and $\Gamma'[y] = \Gamma[y]$ for all $y \neq x$.
8. A value is either a free variable which is bound in the associated heap or a constructor applied to a sequence of variables: $\mathcal{Value} = \{x | c(x^n)\}$
9. A symbolic trace $T$ is a list of $\mathcal{SymInfo}$ objects.
10. A $\mathcal{SymInfo}$ object is a tuple consisting of a case identifier, a branch number, a symbolic variable and the identifier of a FlatCurry constructor.
11. The operation ++ concatenates two lists.

The operational semantics (also referred to as the natural semantics) of normalized FlatCurry uses a heap structure to represent the sharing of expressions and computes a value for a given FlatCurry expression. In addition to this structure we use a symbolic trace to collect and pass symbolic information during evaluation. This trace is extended whenever a branch decision has been made.

The individual evaluation steps of the natural semantics are formalized using the inference rules depicted in Fig. 2. The inference rules of the semantics include judgements of the form $\Gamma, T : e \vdash \Delta, Y : v$ which can be read as “the FlatCurry expression $e$ under the heap $\Gamma$ and with incoming symbolic trace $T$ evaluates to value $v$, the (possibly modified) heap $\Delta$ and the (possibly extended) trace $Y$”.

Apart from the rules (Select) and (Guess), the augmented semantics is equivalent with the FlatCurry semantics presented in [12] except for the fact that symbolic traces are additionally passed through the judgements. Below we give a short description for every rule and explain the modifications to the rules (Select) and (Guess) which are required for concolic execution.
Fig. 2. Natural semantics for concolic execution of normalized FlatCurry programs

A value can not be further evaluated and, thus, is directly returned.

Flattened function calls are further evaluated by evaluating the right-hand side of
the function. For that reason, we assume that the program \( P \) is a global parameter of the calculus. In order to prevent name clashes, we apply a renaming substitution \( \sigma \) whenever new variables are introduced during evaluation.

The bindings of a \texttt{let} expression are renamed and then added to the heap. After that, the main expression of the \texttt{let} \( e \) is evaluated with respect to the bindings.

For the evaluation of non-deterministic choices one subexpression is chosen non-
deterministically to be further evaluated.

Similar to ordinary \texttt{let} expressions logic variables are renamed and added to the heap. Then, the evaluation continues with the main expression \( e \).

In case the inspected expression of a \texttt{case} expression is reducible to a
constructor-rooted term, the right-hand side of the corresponding \texttt{case} alternative is selected and further evaluated. In addition the trace is extended with symbolic information binding the symbolic variable associated with the case expression to the chosen constructor.
If the argument of a case expression evaluates to a free variable. One of the case alternatives is non-deterministically chosen. The free variable is bound to the corresponding pattern and any variables inside this pattern are bound as free. Moreover, depending on the selected alternative the symbolic variable associated with the case expression is bound to the respective constructor.

5 Search Strategies

In the previous section we described the collection of symbolic traces during the evaluation of a FlatCurry expression. A single trace corresponds to a path through the associated symbolic execution tree and the symbolic information derived from a single case expression corresponds to a node of this tree. To produce new test cases, we have to search for unexplored paths through that tree. Hence, we need to select a node with unvisited branches and negate the path constraint associated with that node. If there is a solution for the resulting constraints, new input data which will drive execution along one of these branches can be computed.

A naive strategy - similar to the one presented in [9] - is to choose the first node with unvisited branches which is closest to the root of the symbolic execution tree. This strategy is currently used in ccti.  

The general search algorithm of ccti is depicted in Fig. 3. Basically, two data structures are used during search: On the one hand, there is a priority queue $Q$ storing the nodes of the symbolic execution tree in a strategy-defined order. The naive strategy mentioned above can be implemented by using a priority function preferring the node with the lowest depth in the tree. On the other hand, we maintain a map of all case expressions $M$ including still unvisited branches. Both data structures are updated by processing the trace information collected during a concolic execution.

The central function of the search is SEARCHLOOP. It evaluates the function to be tested $f$ with the given inputs $in$ yielding potentially non-deterministic results $res$ and a list of symbolic traces $T_s$. The inputs and the results form a new test case which is added to the set of test cases $T$. By calling PROCESS the traces are processed to update the priority queue and the case map with the information collected during the previous evaluation. More precisely, for every SymInfo object $(cid, bnr, x, c)$ included in a trace, the queue is extended with information on the case identifier $cid$, the associated symbolic variable $x$ and the set of path constraints $C$ using ENQUEUE. This set includes all constraints associated with the path leading from the root of the symbolic execution tree up to that particular node. VISIT marks the selected branch $bnr$ as visited and adds the chosen FlatCurry constructor $c$ to the set of known constructors for a particular $cid$. Before processing further SymInfo objects of the trace, the set of path constraints is extended with the constructor decision made in the current SymInfo object by applying CONSTR.

While the priority queue $Q$ is not empty, we dequeue the next entry $(d, cid, x, C)$ from the queue. In order to compute input data driving execution along an alternative branch of the case expression identified by $cid$, we have to generate an appropriate

---

Note that ccti provides a flexible interface for the implementation of alternative search strategies.
path constraint. Hence, we select the set of already known constructors for that case expression from the case map and constrain the associated symbolic variable $x$ to be different than any known constructor by calling `GETCONS` and `NONEOF`, respectively. Next, we extend the set of path constraints with the new constraint and call `SOLVE` to apply the SMT solver. In case the constraints are satisfiable, the resulting model is translated into valid FlatCurry inputs with `TOFCY` and a new iteration of the concolic search is started. Otherwise, we proceed with the search for new input data.

\[
\text{SEARCHLOOP } f \text{ args } T Q M =
\begin{align*}
&\text{let } (\text{res}, T_s) = \text{eval } (f \text{ args}) \\
&T' = T \cup \{(\text{args}, \text{res})\} \\
&(Q', M') = \text{fold PROCESS } (Q, M) T_s \\
&\text{in SEARCHINPUT } f T' Q' M'
\end{align*}
\]

\[
\text{PROCESS } [] = Q M C d = (Q, M)
\]

\[
\text{PROCESS } ((\text{cid}, \text{bnr}, x, c) : T_s) Q M C d =
\begin{align*}
&\text{let } Q' = \text{ENQUEUE } Q (d, \text{cid}, x, C) \\
&M' = \text{VISIT } M (\text{cid}, \text{bnr}, c) \\
&C' = C \cup \{\text{CONSTR } x c\} \\
&\text{in PROCESS } T_s Q' M' C' (d + 1)
\end{align*}
\]

\[
\text{SEARCHINPUT } f T \varnothing = M = T
\]

\[
\text{SEARCHINPUT } f T ((d, \text{cid}, x, C) \Rightarrow Q') M =
\begin{align*}
&\text{let } cs = \text{GETCONS } M \text{ cid} \\
&pc = \text{NONEOF } x cs \\
&sa = \text{SOLVE } (C \cup \{pc\}) \\
&\text{in if } sa == (\text{sat}, m) \\
&\text{then SEARCHLOOP } f (\text{TOFCY } m) T Q' M \\
&\text{else SEARCHINPUT } f T Q' M
\end{align*}
\]

Fig. 3. Basic search algorithm of ccti

6 Implementation

This section gives a brief overview of the implementation of ccti which is completely implemented in Curry. The concolic execution part of ccti is implemented by a FlatCurry interpreter implementing the augmented operational semantics of FlatCurry presented in Sect. 4.2. In this section we focus on the integration of SMT in Curry.

As explained before, we want to apply SMT solvers to compute alternative inputs from the information included in a symbolic trace. Hence, we need to transform the path constraints, i.e., the constructor decisions made along an execution path, into an SMT-LIB formula.

In Sect. 3.2 we demonstrated how to model path constraints for our running example in SMT-LIB (see List. 2) by declaring corresponding SMT-LIB types for the types used
in the considered Curry program and representing path constraints as simple relational formulas on constructor terms and integers, respectively.

To simplify the translation of path constraints we provide some libraries in Curry. Among these are a representation of SMT-LIB scripts as abstract data types, a pretty printer and a parser to send String representations of the scripts to the SMT solver and parse its responses. Moreover, an interface to call SMT solvers via Curry as well as a transformation library to convert FlatCurry expressions to SMT-LIB terms and vice versa are provided.

When we run ccti on a Curry module, the module and all its dependent modules are parsed to FlatCurry. To prepare for a type-safe translation of FlatCurry constructor calls to corresponding SMT-LIB terms, the transformation library then builds up bidirectional maps mapping both FlatCurry type and value constructors to their associated sort or term on SMT-LIB side. Furthermore, a corresponding SMT-LIB declaration for all data types occurring in the program is generated.

We also construct a type environment mapping the symbolic variables occurring in the trace to their FlatCurry type and SMT-LIB sort, respectively. On the one hand, this information is required for the declaration of variables in SMT-LIB. On the other hand, it is needed to transform possible results found by the solver into type correct FlatCurry expressions. Note that these results correspond to alternative input data for the function to be tested and, thus, their FlatCurry representation is required to start the next iteration of the concolic execution.

We conclude this section taking a look at another example for the generation of an SMT-LIB script. Reconsidering our running example let us assume that we call ccti with the initial call "nthElem [] 0" this time. The concolic execution of this call produces a symbolic trace which constrains the list argument to be an empty list. If we try to model this path constraint in SMT-LIB, there is a problem: nthElem is defined on polymorphic lists and the example call does not specify a type for the list elements. Nevertheless, that type information has to be determined for the translation, because SMT-LIB does not allow the declaration of polymorphic constants. Hence, during translation of FlatCurry types to SMT-LIB sorts, all occurrences of type variables are instantiated with a monomorphic type. With regard to the generation of test cases, it seems reasonable to use a type for instantiation which includes more than one value but is also simple. For this reason, we use Curry’s Ordering type, which is equivalent to the one in Haskell, whenever polymorphic types need to be instantiated.

Reconsidering the example call of nthElem from above, the resulting path constraint is represented in SMT-LIB as shown below and running Z3 with this script yields the answer ((xs (cons lt nil))).

(declare-datatype Ordering ( (lt) (eq) (gt) )
(declare-datatype List (par (A) ((nil)
 (cons (head A) (tail (List A)))))))

(declare-const xs (List Ordering))
(assert (not (= xs nil)))

Note that we actually use a variant of FlatCurry called TypedFlatCurry which corresponds to FlatCurry with the exception that expressions are additionally annotated with type information.
7 Application of \textit{ccti}

In this section we want to take a closer look at the usage and applicability of \textit{ccti} regarding some practical examples. Note that this work is still in progress. Hence, we only discuss the general applicability of our approach.

Currently, \textit{ccti} expects a Curry module to include a main function calling the function to be tested with user-specified inputs in order to initiate the concolic execution. For the future, we plan to support concolic testing of multiple functions in a single run of \textit{ccti} as well as a random-based generation of initial input data.

For our running example with the initial call “\texttt{nthElem [42] 0}” \textit{ccti} produces four test cases. Among these is also one resulting in a failure, because our implementation of \texttt{nthElem} does no handle negative indices. The minimum number of three test cases (including the one with the failure) is generated, if we call \textit{ccti} with a list with at least two elements and an index greater than 0.

\begin{verbatim}
data Nat = IHi | O Nat | I Nat

add IHi y = succ y
add (O x) IHi = I x
add (O x) (O y) = O (add x y)
add (O x) (I y) = I (add x y)
add (I x) IHi = O (succ x)
add (I x) (O y) = I (add x y)
add (I x) (I y) = O (add (succ x) y)

succ IHi = O IHi
succ (O x) = I x
succ (I x) = O (succ x)
\end{verbatim}

\textbf{List. 3. Addition of binary numbers}

For a more complex example, we consider the implementation of an addition operation on a representation of binary numbers in Curry, which is depicted in List. 3. Calling \textit{ccti} with the initial call “\texttt{add IHi IHi}”, generates nine test cases. These test cases cover all branches, but, in fact, even six test cases would be sufficient for full coverage. This minimum number of test cases is found by \textit{ccti}, if we use “\texttt{add IHi (I (O IHi))}” to initialize the concolic execution.

We conclude this brief case study considering our running example for a non-deterministic operation, namely \texttt{insertND}. Calling \textit{ccti} with “\texttt{insertND True []}”, the following two test cases will be produced.\(^\text{10}\)

\begin{verbatim}
insertND True [] = {[True]}
insertND True [False] = {[True, False], [False, True]}
\end{verbatim}

Finally, we take a look at an example showing a limitation of the search algorithm currently applied in \textit{ccti}. List. 4 shows a definition of an operation to compute all permutations of a list.

\(^{10}\text{Note that we use a set notation to represent multiple non-deterministic results.}\)
Considering the initial call “perm [False]” we receive the single test case perm [False] = [False]. This test case covers both rules of perm but only the first rule of insertND. The problem of the current search strategy is that all function calls are treated equally. Thus, reconsidering our example regarding that strategy, it is sufficient, if the non-empty list branch of perm is visited either of the two calls, i.e. the top-level call of perm or the recursive one. Since our initial call already covers the non-empty list branch when the top-level call of perm is evaluated, ccti does not consider this branch for the recursive call. For that reason, the recursive call of perm can only result in an empty list, and thus the second and third rule of insertND are never tested.

For full coverage of insertND, ccti needs to produce an input list for perm with at least two elements. To compute such a list, we need to reconsider all branches in the recursive call of perm, even if they already have been visited in the top-level call. Hence, to achieve full coverage, we need an alternative strategy taking context information of function calls, i.e. possible previous calls, into account. This approach corresponds to the Function Coverage criterion discussed in [8].

8 Conclusions and Future Work

In this work, we have presented ccti, a tool for concolic testing of functional logic programs written in Curry. We have extended the operational semantics of FlatCurry - a simplified core language of Curry - to collect the necessary information for concolic testing during concrete evaluations. ccti is based on a FlatCurry interpreter implementing this semantics. In order to compute input data for the exploration of alternative execution paths, we apply an external SMT solver which has been integrated in Curry. In contrast to the narrowing-based generation of test cases, path constraints on literals can be mapped directly to corresponding constraints in SMT-LIB by making use of suitable theories of the SMT solver.

Although the naive search algorithm currently provided by ccti is incomplete, first applications of ccti show that our approach is applicable for the automated generation of test cases of functional logic programs.

For future work, we plan to implement alternative strategies and coverage criteria considering context information of symbolic nodes in order to overcome the problems mentioned above. Moreover, we intend to further investigate the applicability of ccti by evaluating the generation of test cases for more complex programs.

References


CalcuList: a Functional Language Extended with Imperative Features

Domenico Saccà and Angelo Furfaro
DIMES, Università della Calabria, 87036 Rende, Italy
sacca@unical.it, a.furfaro@unical.it

Abstract. CalcuList (Calculator with List manipulation), is an educational language for teaching functional programming extended with some imperative and side-effect features, which are enabled under explicit request by the programmer. In addition to strings and lists, the language natively supports JSON objects. The language has a Python-like syntax and interactive computation sessions with the user are established through a REPL (Read-Evaluate-Print-Loop) shell. The object code produced by a compilation is a program that will be eventually executed by the CalcuList Virtual Machine (CLVM).

Keywords: Functional Language · Imperative Features · Side Effects

1 Introduction

A recent trend in programming languages is a renewed interest towards functional programming, particularly in combining it with other paradigms, mainly imperative programming. This paper will focus on combining functional and imperative programming in a new language called CalcuList. Within this framework, observe that functional features have been recently added to existing imperative languages, e.g., lambda expressions in Java [4], and new languages, e.g., Scala [6], supporting both paradigms, have been introduced as well. We point out that no matter is the number of functional features introduced in un imperative language, most programmers continue to use the imperative features and neglect the functional ones. In contrast, imperative features alone cannot work in CalcuList and, therefore, any application has a functional core. Thus CalcuList belongs to the family of functional languages.

Purely functional programming usually designates a specific functional programming paradigm that treats all computation as the evaluation of mathematical functions. Purely functional programming forbids changing-state and mutable data and mainly consists in ensuring that functions will only depend on their arguments, regardless of any global or local state (i.e., it has no side effects). Many so-called functional languages are less pure (impure), as they contain imperative features. For instance, most of the languages of the Lisp Family [12] were designed to be multi-paradigm and mainly present a purely functional
interface for many operations (particularly simple mathematics and list/set operations) and an impure interface (more often object-oriented) for things where side effects are desirable, such as I/O.

Haskell [7] is a language of the Lisp Family that is characterized by a purely-functional core. It provides relevant features including polymorphic typing, static type checking, lazy evaluation and higher-order functions. It also has an innovative type system which supports a systematic form of overloading and a module system.

Haskell also expresses side effects but it does it by using a clean approach based on monads that can be thought of as composable computation descriptions. The essence of monad is thus separation of composition timeline from the composed computation’s execution timeline, as well as the ability of computation to implicitly carry extra data, as pertaining to the computation itself, in addition to its one (hence the name) output, that it will produce when run (or queried, or called upon). This lends monads to supplementing pure calculations with features like I/O, common environment or state, etc. Thus in Haskell, though it is a purely-functional language, side effects that will be performed by a computation can be dealt with and combined purely at the monad’s composition time.

In contrast with Haskell, CalcuList describes impure effects and actions inside the language itself, although it isolates impure components using an approach, different from Monads, which is derived by the semantic rules of an Attribute Grammar [8]. An attribute grammar extends a classical context-free grammar to specify the context-sensitive aspects of the syntax of a programming language (such as checking that an item has been declared and that its use is consistent with the declaration) as well as its operational semantics (e.g., by defining a translation into lower-level code of a specific machine architecture). In a similar way as a context-free grammar production is enriched with semantic rules to assign values to attributes associated to the symbols occurring in the production, CalcuList enriches the expression defining a function with statements assigning values to global variables and function parameters.

It is interesting to observe that the Curry language [1], defined by extending the functional core of Haskell, deals with the issue of combining functional and logic programming. CalcuList does not include any logic programming feature, a part from some syntactic notation borrowed from the logic language Prolog [2] to represent some basic operations on lists (see Section 2.2).

CalcuList (Calculator with List manipulation), is an educational programming language for teaching the functional paradigm, suitably extended with some imperative features involving side effects. The programmer may use CalcuList as a pure functional language, in that side effects are required to be explicitly enabled. The use of side-effects, in some cases, simplifies the developed code or even results in a better computational efficiency.

CalcuList syntax is rather similar to that of Python [11] and natively supports processing of strings, lists and JSON objects [3]. The language is strongly typed but most of the type checking is done at run-time. An interactive computation session with the user is established by means of a REPL (Read-Evaluate-Print-
Loop) shell. During a session, a user may define a number of global variables and functions and run suitable computations on them as queries, that are computed and displayed on the fly. CalcuList expressions and functions are first compiled and then executed each time a query is issued. CalcuList can be thought of as an “interactive compiler” rather than an interpreter and both static and dynamic type-checking are performed. The object code produced by a compilation is a program that will be eventually executed by the CalcuList Virtual Machine (CLVM).

CLVM is an abstract computing machine that has a set of instructions that are implemented using micro-instructions operating with three memories (MEM, divided into two parts: STACK and HEAP, CODE and OUTPUT) and a number of registers. The clops (CalcuList OperationS) of an instruction is the number of micro-instructions executed for its implementation.

The remainder of this paper is organized as follows. Section 2 introduces basic notions about the CalcuList functional core. Section 3 describes the imperative aspects of the language and related side effects. Finally, Section 4 draws the conclusions and discusses future work.

2 An Overview of CalcuList Functional Core

2.1 Basic Computations

The main interface to CalcuList, like other languages (e.g. Python, Scala), is a REPL environment where the user can issue valid expressions (queries in CalcuList), which in turn are parsed, evaluated and printed before the control is given back to the user.

The basic types for CalcuList are five: (1) double (i.e., real number in 64-bit double-precision floating-point format), (2) int (i.e., a 32-bit integer), (3) char (i.e., character represented in the UNICODE format), (4) bool (i.e., a Boolean with values true or false) and (5) null (that has a unique value named null as well). The language also supports three compound types: string (immutable sequences of characters), list (sequences of elements of any type), and json (JavaScript Object Notation), that is a lightweight data-interchange format. Details on the usage of strings are given next, whereas lists and jsons will be treated in Section 2.2 and in Section 2.3, respectively.

The first three basic types are numbers and they are used in arithmetic operations. If the operands are of different type, they are automatically casted to their most general numeric type: double is more general than int that, in turn, is more general than char. Moreover, the result of an arithmetic operation of two char terms returns an int value, as it happens in Java.

The arithmetic binary operators that may be used by number terms are +, −, ∗, /, //, %. The first three operators work just like in most other languages (for example, Java or Python). On the other hand, the operator / performs floating-point division also when both operands are integer, whereas // performs integer division (i.e. quotient without remainder) like in Python. Finally, the modulus operator % computes the remainder like in Java.
The language includes unary versions of the operators + and - to change sign to a term. Moreover, the unary operator - applied to a char term yields an integer result. As we shall see later, the operator + is further overloaded as it may be used for concatenating strings and lists.

The language includes comparison operators >, >=, <, <=, equality == and inequality !=, which work just like in most other languages (for example, Java or Python). The two operands of a comparison operator must be both: (1) numeric, or (2) bool (with false < true) or (3) string (with lexicographic order). On the other hand, the two operands for equality and inequality may be of any (possibly heterogeneous) type. Finally, the logical operators follows Java syntax: ! (not), && (and), || (or). Short-circuit evaluation is adopted for binary logical operators, i.e., given A && B (resp., A || B), B is evaluated only if A is true (resp., false).

A string in CalcuList is an immutable sequence (possibly empty) of characters. Two strings can be concatenated by the overloaded operator + returning a new string. Like in Java, it is also possible to concatenate a string with a character (but not the reverse) – the latter one is automatically casted to a string. Slice operators à la Python are available to extract a substring. Given a string s, s[i] returns the i-th character of s, s[i:] returns the substring of s starting from the i-th character up to its end; s[:i] returns the substring of s starting from the initial character up to (i − 1)-th character of s; s[i:j] returns the substring of s that begins at the index i and ends at the index j − 1.

The syntax for an expression is like in Java. An expression e is either simple or conditional with format e1 ? e2 : e3, where e1 is a logical (i.e., with type bool) expression and e2 and e3 are expressions (possibly conditional in their turn). The meaning of a conditional expression is: its value is equal to the value of e2 if e1 evaluates to true or, otherwise, to the value of e3.

The language allows the user to define global variables. A global variable V is defined as ”V = e”, where e is an expression: the type of V is inferred from the type of e and its value is kept until a later re-definition, which may change also the type. Compound assignment operators +=, -=, *=, /= are available as well.

At the start of a CalcuList session, the set of global variables is empty. The state of a session is given by the set of all defined global variables with the most recently assigned values and types for them.

A query is an expression e prefixed by the ^ symbol and prints the value of the expression at the current state. Examples of variable definitions and queries are shown in the CalcuList session of Figure 1 - they are prompted by ”>>” and are terminated by a semicolon. The state at point of the session indicated by the comment ”/* checkpoint 1 */”, is: {<x, bool, true>, <y, string,"Hello World">}.

In addition to queries, the user may submit “service commands” (preceded by !) to inquiry about the variables and functions defined during the session, the history of all definitions, the state of the internal memory, the current release of CalcuList and the number of micro-instructions performed in the last execution, to enable or disable the debugger and to save or import a session of definitions.
A function is defined by giving it a name, followed by: its comma-separated parameter names (included in parentheses), the colon symbol “:” and the (typically conditional) expression that computes the value of the function. Parameters as well as the return value of a function are defined without specifying types for them, so that static type checking is performed only for constant operands. A function is compiled while it is defined and, as for Python, the type checking is done at run time, after its call, when all the types become available. In Section 4 we shall discuss a forthcoming extension of CalcuList for inferring types for some function parameters. Lazy evaluation is not supported and at the moment, there is no plan to introduce it in the future.

No side effects are allowed in the basic definitions of functions, i.e., functional operations do not modify current global variables and always create new data objects. Differently from Python, global variables are not accessible inside a function so that a possible change of state does not affect the function behavior. An example of function computing a Fibonacci number by a forward execution, starting from the first two Fibonacci numbers, is shown in Figure 1. The value 55 of the fibonacci number 20 is computed and stored into the variable z: this value is of type int as confirmed by the answer to the query `^z@type`.

Observe that, given an expression e, whose value is one of the eight admissible types, the casting operator e@type returns the type of e – in the example of Figure 1, `^z@type` returns the type int of the value 55. The eight types can be also directly assigned as values to variables and their type is type, i.e., `^int@type` returns type. The state at the end of the session is: `{<x, bool, true>, <y, string, "Hello World">, <z, int, 55>}."

2.2 Lists

The list is a powerful compound data type used by CalcuList for constructing dynamic data structures and implementing recursive algorithms. A list L consists of a number (possibly zero) of comma-separated elements between square brackets. As in Python, the elements can be of any type, including list and json,
and heterogeneous. They are numbered with an index starting from zero: the first element (called the head of the list) is \( L[0] \) (also denoted simply by \( L[.]. \)), the second element is \( L[1] \) and so on. List assignment is a shallow operator as the list elements are shared by the two lists involved in the assignment. Two lists can be compared only with the operators \( == \) and \( != \).

A list can be extended by adding additional comma-separated elements on top, followed by the append operator (the bar “|”) - the syntax has been inspired by Prolog [2]. For instance, given a list \( L, M=x,y|L \) extends \( L \) by adding two new elements, \( x \) and \( y \), on top of \( L \). Thus, append is a shallow operator so that, for instance, if an element of \( L \) is changed, the corresponding element in \( M \) is changed as well. Also the concatenation \( L1+L2 \) of two lists is shallow as the elements of \( L2 \) are afterwards shared by \( L1 \).

Another shallow operator on a list \( L \) is \( L[>\] \), which returns the (possibly empty) tail of \( L \), i.e., the list starting from the element \( L[1] \). If \( L \) contains exactly one element then \( L[>\] \) is the empty list \( [] \). On the other hand, if \( L \) is empty, then the evaluation of \( L[>\] \) will report an execution error.

Slice operators on a list \( L \) (say with \( n \) elements) are deep in the sense that they clone the elements. In particular, \( L[:\] \) clones the whole list \( L \) and \( L[i:] \) clones the sublist of \( L \) from the element \( L[i] \) to the end; in the same way, \( L[i1:i2] \) and \( L[:i] \) clone the corresponding sublists.

```
>>> member(x,L): L !=[] && (x==L[.] || member(x,L[>]));
>>> listRev(L): L==[]? [L]; listRev(L[>])+[L[.]];
>>> rev(L): revl(L,[]);
>>> range(x1,x2): x1>x2? [x1| range(x1+1, x2)];
>>> L = range(1,1000);
>>> !clops;
2795726
>>> rev(L);
>>> !clops;
296308
>>> merge(01,02): 01==[]? 02:02==[]? 01[.]: 01[.]<02[.]? [01[.]|merge(01[>], 02)]; [02[.]|merge(01,02[>])];
```

**Fig. 2: Functions on Lists**

Figure 2 includes a number of functions for handling lists. First of all, the classical `member` function that checks whether an element belongs to a list. The `listRev` function constructs the reverse of a given list. At the generic recursion step on a current list \( L \), the function concatenates the tail of \( L \), recursively computed, with its head. The complexity if `listRev` is quadratic in the size of the list length, say \( n \) - actually the number of operation is \( n(n − 1)/2 \). A linear-time implementation is given by `rev`, which uses “tail recursion”.

To compare the two implementations, we construct a “long” list with 1000 elements by means of the `range` function and use the service command `!clop`, which returns the number of micro-instructions (CalcuList micro OperationS) of
CalcuList Virtual Machine that have been performed in the last execution – in the example of Figure 2, `listRev(L)` is executed by 2,795,726 clops, whereas `rev(L)` is executed by 296,308 clops, i.e., it is ten times faster. Tail recursion enables avoiding to scan the entire first term list in the concatenation operation – in CalcuList an element of a list cannot be directly accessed by its index but only after scanning all the previous elements. A careful reader may observe that the clops difference should be larger. Indeed, the smaller difference is the result of two factors: (1) the implementation of the concatenation operator `+` for lists is internally optimized by CalcuList during the search of the last element of the first list operand and (2) the implementation of tail recursion is not yet optimized by CalcuList, so that we do not avoid allocating a new stack frame for the tail-recursion.

Let us now raise an important issue about possible side effects that could be induced by the usage of shallow list operations in a function. For instance, the function `listRev` contains the shallow operation `listRev(L[>])+[L[.]]`. However it is easy to see that the operation does not have any side effect on the parameter `L` since the list corresponding to `listRev(L[>])` is constructed from an empty list (at the last step of the recursion) that is later extended with elements copied from `L` at each previous step. Also the function `rev1` contains a shallow operation `L[.]|R`. In this case, the second parameter `R` of the function is modified by the computation. But this parameter has been passed by the function `rev` with the initial value of an empty list so that, by providing syntactic restrictions for which `rev1` can be only called by `rev`, no side effects will affect the session state. As we shall discuss in Section 4, such restrictions as well as the check that shallow list operations are used safely within a function definition will be introduced in a forthcoming version of CalcuList. At the moment, the check is left to the programmer.

Figure 2 includes another function that makes use of shallow operations: `merge(O1,O2)`. In this case, the exit conditions do not return an empty list as for `listRev` but residual portions of the two parameters `O1` and of `O2` that are, however, cloned to avoid possible side effects.

### 2.3 JSON

JSON (JavaScript Object Notation) [3] is language-independent data interchange format, which has been originally introduced as a subset of the JavaScript scripting language. The use of JSON is now widespread in many applications, e.g. web-services, and supports for it has been added to the standard libraries of many programming languages. Recently, a great deal of interest is focused on JSON database stores that are NoSQL database systems using JSON documents instead of relation tuples [5].

CalcuList natively supports JSON objects (referred to simply as `json`), which are at runtime represented as (possibly empty) sequences of fields separated by comma and enclosed into curly braces. A field is a pair (key, value) separated by a colon: `key` is a string and `value` can be of any type: double, int, char, bool,
string, null, list and (recursively) json. Two fields of a json object must not have
the same key.

In Figure 3 we define a variable \texttt{emps} as a list of three JSONs, each rep-
representing an employee. For instance, \texttt{emps[1]} is the employee with name \texttt{"e2"},
aged 32 and working for the projects \texttt{"p1"} and \texttt{"p2"}. Given a json \texttt{J} and a key
\texttt{k}, \texttt{J[k]} denotes the value of the field of \texttt{J} with key equal to \texttt{k} – if there is no such
a field then the value \texttt{null} is returned as it happens for \texttt{emps[0]["projects"]}
(the string \%* next to the query is a display option to force the writing of the
null value). Given a value \texttt{v}, \texttt{J[k] = v} modifies the value for the field if \texttt{J} includes
the key \texttt{k} or otherwise, \texttt{J} is extended with a new field with key \texttt{k} and value \texttt{v}.
The assignment of a json to a variable is a shallow operation. To clone a json \texttt{J},
it is sufficient to type \texttt{J[:]}. Two jsons can be compared only with the operators
\texttt{==} and \texttt{!=} and the equality is true if and only if the two json values point to the
same data structure in the memory. The length of a json \texttt{J}, i.e., the number of
its fields, can be obtained by invoking the built-in function \texttt{len(J)}.

2.4 Higher-Order Functions

CalcuList supports higher-order functions since a function parameter can also
be a function; however, a function cannot return a function. In Section 4, we
shall explain the reason of such a choice and, in addition, we shall discuss how
to overcome this limitation in a forthcoming version of CalcuList. A function
parameter \texttt{f} is written as \texttt{f/n}, where \texttt{n} is the arity of the function \texttt{f}. This
syntax has two advantages: functions parameters can be easily distinguished
from regular ones and an early static check on arity can be easily done, whereas
a run-time check would require a more complex framework.

Some examples of higher-order function are presented in Figure 4. The func-
tion \texttt{map} receives a list \texttt{L} and two functions as parameters, both with arity 1: the
function \texttt{f} checks whether an element has a certain property and, in case the
test succeeds, \texttt{m} maps the element into another value. The function \texttt{map} scan all
elements of \texttt{L}, for each of them it performs the possible mapping and eventually
returns the list of the results. The first map query filters all numbers divisible
by 2 or by 3 (see function \texttt{d2or3}) in the range from 1 to 10 and replaces each of
these values with their cube (see lambda function, written in Python like syntax)
– let \texttt{M} denote the result of this query.

The subsequent function \texttt{reduce} receives a list \texttt{L}, a function parameter \texttt{f}
(mapping two values into another value) and an initial value that is returned at
map (L,f,m) : L == []? [] : f(L[.])? [m(L[.]) | map (L[>], f, m)] : map (L[>], f, m);

>> d2or3 (x) : x %2 == 0 || x %3 == 0;

>> ^ map (range (1,10), d2or3, lambda x: x*x*x);
[ 8, 27, 64, 216, 512, 729, 1000 ]

>> reduce (L,f/2, init) : L == []? init : f(L[.], reduce (L[>], f, init));

>> sum (x,y): x+y; >> prod (x,y): x*y;

>> ^ reduce (map (range (1,10), d2or3, lambda x: x*x*x), sum, 0);
2556

>> ^ reduce (map (range (1,10), lambda x: x %2 == 0 && x %3 == 0, lambda x: x*x*x), prod, 1);
36

>> jsFilter (LJ, filtC, K, V) : LJ == []? [] : filtC (LJ[.], K, V)? [LJ[.]] | jsFilter (LJ[>], filtC, K, V);

jsFilter (LJ[>], filtC, K, V);


>> jsFilter (emps, select1KV, "age", 28);
[ { "name": "e3", "age": 28, "projects": [ "p1", "p3" ] } ]

>> select1KinV (J, K, V): J[K] != null && member (V, J[K]);

>> jsFilter (emps, select1KinV, "projects", "p1") ;
[ { "name": "e2", "age": 32, "projects": [ "p1", "p2" ] },
{ "name": "e3", "age": 28, "projects": [ "p1", "p3" ] } ]

Figure 4: Higher Order Functions on Lists and Jsons

The final stage of the recursion, when L reduces to an empty list. Then reduce applies f to two actual parameters: the list head and the result of reduce for the list tail. Therefore, the list is eventually reduced to a single value by recursively applying f from right to left. The first reduce query computes the sum of all elements in M (see the list previously defined), whereas the second one computes the product of the squares (see second lambda function parameter) of all integers in the range from 1 to 10 that are divisible both by 2 and by 3 (see first lambda function parameter).

Figure 4 also includes a high-order function to filter the jsons that satisfy some conditions in one of the fields: jsFilter, whose filtering condition parameter is expressed by a function filterC, that receives the current json and a field, say with key K and value V, to be used for the evaluation. Next, the figure shows two implementations for filterC: (1) select1KV, which checks whether the json has a field equal to (K, V), and (2) select1KinV, which checks whether the value V is included in the list of elements that represents the value for the the field K in the json. Two queries on the variable emps defined in Figure 3 are issued: the first one filter all employees with age 28 and the second one the employees that work for at least the project "p1".

3 Imperative Aspects of CalcuList

3.1 Local Variables

To simplify the writing of a function and, sometime, to optimize its execution (mainly, by avoiding to call twice the same function), it is possible to define local variables inside a function as follows: f(...): <LV_1,..., LV_n> expr. The n variables are only used by the function f and are stored into each of the frames of each called f instance.
The local variable values are initialized and updated by means of the so-called Local Setting Commands (LSC), that are imperative statements without side effects. An LSC is an assignment of an expression value to a local variable, surrounded by adorned curly braces \{!, !\}. If \texttt{expr} is a simple expression, LSCs can be inserted before it as: \texttt{LSC* expr}, where \texttt{LSC} is executed before the function starts its execution. Note that, as indicated by the Kleene operator *, \texttt{LSC} may indeed consist of a sequence of commands, each of them surrounded by curly braces.

In case \texttt{expr} is a conditional expression, the local setting commands may be also inserted before each sub-expression, e.g.,

\[
\texttt{LSC_1^* cond? LSC_2^* expr_1 : LSC_3^* expr_2}
\]

In this case, \texttt{LSC_1} is executed before the function starts its execution and, on the basis of the result of \texttt{cond}, either (1) \texttt{LSC_2} is executed before the execution of \texttt{expr_1} or (2) \texttt{LSC_3} is executed before the execution of \texttt{expr_2}. If \texttt{expr_1} (or \texttt{expr_2}) is in turn a conditional expression, additional local setting commands can be introduced using the same schema as above.

As mentioned in Section 1, the usage of LSCs has been inspired by the semantic rules of an Attribute Grammar [8]. A local variable can be thought of as an inherited grammar attribute and an LSCs as a semantic rule setting the attribute value.

As a first example, consider the (clockwise) rotation of the \(n\) elements of a list by \(k\) positions consists of (i) shifting the elements to the right by \(k\) positions and (ii) moving the last \(k\) elements to the top of the list. For instance, given the list \(L = [5, 1, 4, 20, 15, 13]\). The rotation of \(L\) by \(k\) positions is \([15, 13, 5, 1, 4, 20]\) if \(k = 2\) or \([4, 20, 15, 13, 5, 1]\) if \(k = 4\). If \(k \geq n\), \(k\) is replaced by \(k \mod n\), i.e., the remainder of dividing \(k\) by \(n\). If \(k = 0\) the list remains unchanged. If \(k < 0\), the rotation is anti-clockwise for it is made by (i) shifting the elements to the left by \(k\) positions and (ii) moving the first \(k\) elements to the end of the list – this corresponds to a clockwise rotation by \(-k\) positions. The definition of the function \texttt{rotate} is shown in Figure 5 and two local variables

\[
\[
\]

Fig. 5: Functions with Local Variables
(n and k1) are used to simplify both the writing and the implementation. The built-in function \texttt{len} returns the length of a list. Observe that the function does not have side effects.

As second example, we implement Quick Sort, which is a well-known sorting algorithm, which runs in logarithmic time in the average case. Let x be the list head and T be the list tail. Then we partition T into the list T0 of the elements y of T such that y ≤ x (or larger, depending on the the ordering o) and the list T1 of all other elements. Then, recursively sort T0 and T1 and finally return the concatenation of the two resulting ordered lists, separated by the element x. The definition of Quick Sort is shown in Figure 5. Observe that, given T, the function \texttt{part} returns T0 and T1 as two elements of a list after one scan of the list T. For each stage of quicksort, the function \texttt{part} is called once inside the LSC and the result is assigned to the local variable T.

### 3.2 Global Variables inside Functions

One of the peculiar characteristics of pure functional programming languages is being stateless, i.e. each computation can only work on data which are received as its input parameters and on data obtained as outputs from operations invoked during its execution. The use of global variables is then not allowed. While this forces the user to develop, and hence reason, in functional style, there are some situations where the ability to have side-effects may simplify the developed code or even result in a better computational efficiency.

A function in CalcuList can be simply declared with side effects by adding * next to its name (\textit{star function}) so that the usage of a number of global variables is enabled – the property of a function to be with or without side effects cannot be changed after its first definition. Global variables to be used by a star function must be explicitly listed in its definition body. Then, they not only are made accessible to the function, but also they may have assigned new values during the function execution, as in an imperative programming language. Actually also parameter values can be modified during the execution of a star function. Obviously, a star function cannot be called by a non-star one.

There are situations (e.g., while importing definitions elaborated in previous sessions) in which a user may accidentally erase the value of a global variable by introducing one with the same name. To mitigate the risk of creating unwished homonyms, CalcuList allows the user to assign a label to variables. For instance the command \texttt{L: V1,V2,V3} declares three labeled variables \texttt{L.V1}, \texttt{L.V2} and \texttt{L.V3} – they are all initialized to null. After its definition, a labeled variable may have assigned a value by a suitable assignment statement. A label definition can be reissued to declare additional variables with that label and to reset previously-defined labeled variables to null.

To be used inside a function definition, a global variable must be declared as a labeled variable and its label (say L), followed by *, must be listed between brackets as <L*> at the beginning of the function definition. The labeled variables are then addressed inside the function without writing their label. It is possible...
3.3 Functions with side effects

A star function may have side effects, that is: (1) labeled global variables can be used as terms inside the expression defining the function, in addition to constants, parameters and function calls and (2) both labeled global variables and parameters can be updated inside a function by means of the so-called global setting commands (GSC). A GSC is an assignment of an expression value to a labeled global variable or to a function parameter and represents an imperative statement with side effects. The syntax of GSCs is similar to the one of LSCs, described above. The local variable values are initialized and updated by means of the so-called Local Setting Commands (LSC), that are imperative statements without side effects. An LSC is an assignment of an expression value to a local variable, surrounded by curly braces \{!, !\}. If expr is a simple expression, GSCs can be inserted not only before the expression expr but also after it:

\[
GSC_1^* \text{expr} \ GSC_2^*
\]

where \(GSC_1\) is executed before the function starts its execution and \(GSC_2\) at the end of its execution. Note that \(GSC_1\) and \(GSC_2\) may indeed consist of a sequence of commands, each of then surrounded by curly braces \{!, !\}.

In case expr is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

\[
GSC_1^* \text{cond} \ GSC_2^* \text{expr}_1 \ GSC_3^* : \ GSC_4^* \text{expr}_2 \ GSC_5^*
\]

In this case, \(GSC_1\) is executed before the function starts its execution and \(GSC_2\) at the end of its execution. Note that \(GSC_1\) and \(GSC_2\) may indeed consist of a sequence of commands, each of then surrounded by curly braces \{!, !\}.

In case expr is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

\[
GSC_1^* \text{cond} ? \ GSC_2^* \text{expr}_1 \ GSC_3^* : \ GSC_4^* \text{expr}_2 \ GSC_5^*
\]

In this case, \(GSC_1\) is executed before the function starts its execution and \(GSC_2\) at the end of its execution. Note that \(GSC_1\) and \(GSC_2\) may indeed consist of a sequence of commands, each of then surrounded by curly braces \{!, !\}.

In case expr is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

\[
GSC_1^* \text{cond} ? \ GSC_2^* \text{expr}_1 \ GSC_3^* : \ GSC_4^* \text{expr}_2 \ GSC_5^*
\]

In this case, \(GSC_1\) is executed before the function starts its execution and \(GSC_2\) at the end of its execution. Note that \(GSC_1\) and \(GSC_2\) may indeed consist of a sequence of commands, each of then surrounded by curly braces \{!, !\}.

In case expr is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

\[
GSC_1^* \text{cond} ? \ GSC_2^* \text{expr}_1 \ GSC_3^* : \ GSC_4^* \text{expr}_2 \ GSC_5^*
\]

In this case, \(GSC_1\) is executed before the function starts its execution and \(GSC_2\) at the end of its execution. Note that \(GSC_1\) and \(GSC_2\) may indeed consist of a sequence of commands, each of then surrounded by curly braces \{!, !\}.

In case expr is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

\[
GSC_1^* \text{cond} ? \ GSC_2^* \text{expr}_1 \ GSC_3^* : \ GSC_4^* \text{expr}_2 \ GSC_5^*
\]

In this case, \(GSC_1\) is executed before the function starts its execution and \(GSC_2\) at the end of its execution. Note that \(GSC_1\) and \(GSC_2\) may indeed consist of a sequence of commands, each of then surrounded by curly braces \{!, !\}.

In case expr is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

\[
GSC_1^* \text{cond} ? \ GSC_2^* \text{expr}_1 \ GSC_3^* : \ GSC_4^* \text{expr}_2 \ GSC_5^*
\]

In this case, \(GSC_1\) is executed before the function starts its execution and \(GSC_2\) at the end of its execution. Note that \(GSC_1\) and \(GSC_2\) may indeed consist of a sequence of commands, each of then surrounded by curly braces \{!, !\}.

In case expr is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

\[
GSC_1^* \text{cond} ? \ GSC_2^* \text{expr}_1 \ GSC_3^* : \ GSC_4^* \text{expr}_2 \ GSC_5^*
\]

In this case, \(GSC_1\) is executed before the function starts its execution and \(GSC_2\) at the end of its execution. Note that \(GSC_1\) and \(GSC_2\) may indeed consist of a sequence of commands, each of then surrounded by curly braces \{!, !\}.

In case expr is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

\[
GSC_1^* \text{cond} ? \ GSC_2^* \text{expr}_1 \ GSC_3^* : \ GSC_4^* \text{expr}_2 \ GSC_5^*
\]
and after the execution of $\text{expr}_2$. If $\text{expr}_1$ (or $\text{expr}_2$) is in turn a conditional expression, additional local setting commands can be introduced using the same schema as above.

Note that also the usage of GSCs has been inspired by the semantic rules of an Attribute Grammar. In this case, a global variable can be thought of as either an inherited or a synthesized grammar attribute and a GSCs corresponds to a semantic rule setting the attribute value. In Figure 6 the variable $\text{zeroD}$ is defined with label $\text{MATH}$ and is used in the $\text{div}$ function to detect division by zero. Next a function $\text{listDiv}$ is defined that, given a number $x$ and a list $L$ of elements, returns the list of all divisions of $x$ by every element in $L$ if the division of $x$ by that element is feasible or ‘*’ otherwise. The overall implementation makes use of two additional global variables with label $\text{MATH}$: $\text{numErr}$, which counts the number of wrong divisions, and $\text{somma}$, which compute the sum of all correct divisions. The local variable $d$ is also used to store the division result.

As an example of star function without global variables, consider the function, $\text{swap}$ reported in Fig. 6, that swaps two elements $i$ and $j$ of a list $L$. The function always returns $\text{true}$ after swapping the two elements. The $\text{swap}$ function has side effects for it alters the content of its formal parameter $L$ and, therefore, of the global variable $K$, which is passed as argument (actual parameter).

```plaintext
>> newVx_1 (m,j) : j >=m? [ ] : [ 0 | newVx_1 (m,j+1)];
>> newVx (m) : m <= 0? [ ] : newVx_1 (m,0);
>> triangLS_sum (A,X,i,j,n1): j>n1? 0: A[i][j]*X[j]+triangLS_sum (A,X,i,j+1,n1);
>> triangLS_1 *(A,B,X,i,n1): i<0? X: A[i][i]==0? exc("matrix A singular"): (! X[i]=(B[i]-triangLS_sum (A,X,i,i+1,n1))/A[i][i] !)
triangLS_1 (A,B,X,i-1,n1);
>> triangLS (*A,B): <n>((! n=len(A) !) n!=len(A[0])||n!=len(B)?
exc("matrix A and vector B are not conformant"): triangLS_1 (A,B,newVx (n),n-1,n-1);
>>’triangLS([[1,2,-1],[0,2,4],[0,0,-2]],[6.5,3.0,-1.5]);
[ -6.5, 3.0, -1.5 ]
```

Fig. 7: Resolution of a Upper-Triangular Linear Equation System

An example of star function that shows the capability of adding imperative statements while preserving the basic functional style is presented in Figure 7: resolution of a linear equation system $A \times X = B$ for the case $A$ is a an upper triangular matrix, i.e., all the entries below the main diagonal are zero. Note that a necessary and sufficient condition to have finite solutions is that the diagonal does not contain zeros. Using an imperative language, this problem can be easily solved in quadratic time. The star function $\text{triangLS}$ implements a classical imperative algorithm on matrices in a straightforward way; however, its overall complexity is cubic as shown next. The function, after first checking whether the matrix $A$ and the vector $B$ are conformant (i.e., $A$ is quadratic, say with size $n$, and $B$ has size $n$ as well), calls $\text{triangLS}_1$ by setting the index $i = n - 1$. Then $\text{triangLS}_1$ computes $X[i] = B[i]/A[i][i]$, and recursively continues the computation by decreasing the index $i$ down to 0 so that, at a generic step $i$,
X[i] = (B[i] - \sum_{i<j<n} A[i][j] \cdot X[j]) / A[i][i]. Note that if the function finds a zero in the diagonal, it stops the computation by raising an exception. The cost of the summation in the generic step is not linear but quadratic as the direct access is not available in CalcuList to access list elements. The complexity can be improved by rewriting the functions \texttt{triangLS\_sum} and \texttt{triangLS\_1} as follows:

\texttt{triangLS\_sum}(A, X, i, n1) : A[i] == 0? 0: A[i]\cdot X[i] + \texttt{triangLS\_sum}(A[i], X[i], n1);
\texttt{triangLS\_1}(A, B, X, i, n1) : \langle A, X, i\rangle? i<0? X:
{ ! Ai = i==0? A[i]: A[i][i-1] ! } { ! Xi = i==0? X: X[i-1] ! } Ai[0]==0? exc("matrix A is singular"):
{ ! Xi[0]=(B[i]-\texttt{triangLS\_sum}(A[i], X[i], n1))/Ai[0] ! }
\texttt{triangLS\_1}(A, B, X, i-1, n1);

where \(L[i]\) return a shallow copy (i.e., without actually scanning all elements) of the sublist of \(L\) starting from the element \(L[i+1]\) up to the end. In this way the summation at the generic step has linear time complexity and, therefore, the overall computation is done in quadratic time. As a small experiment, we randomly generated two problem instances: the first one with \(n=50\) and the second one with doubled size, i.e., \(n=100\). Using the functions in Figure 7, the number of clops increased of 6.87 times while the increase was only of 3.80 times for their improved versions. Moreover, the number of clops by the improved version were around 23\% of the number of clops by the first version for \(n=50\) and 13\% for \(n=100\).

Figure 8 includes a star function that manipulates a list of jsons. As a list of jsons can be thought of as a documentary database, star functions have the crucial role of updating the database. As an example we have written the function \texttt{giveBonus} that assigns a bonus of a given amount to all employees in a list – the amount is incremented if an employee already has a bonus. Obviously the function has side effects as it updates the elements in the parameter \texttt{emps}. We present a query that assigns a bonus of 100 Euro to all employees in the global variable \texttt{emps} (defined in Figure 3) who work in project \texttt{p1}.

We conclude by mentioning that versions of GSCs can be used to implement advanced printing features: they are called \textit{Global Printing Commands (GPS)} and are enclosed between the curly braces \{\{\}\}. The print output can be redirected to an external text file by the command >>\texttt{(file name)}. There is also a command <<\texttt{(file name)} for reading a CalcuList value, such as a list or a json, from an external text file.

Fig. 8: JSON Manipulation in CalcuList
4 Conclusion

In this paper we have presented a new educational functional programming language extended with imperative programming features: CalcuList, whose imperative features are enabled under explicit request by the programmer. CalcuList expressions and functions are first compiled and then executed each time a query is issued. The object code produced by a compilation is a program that will be eventually executed by the CalcuList Virtual Machine (CLVM). The system provides an assembler component to run CLVM programs using an assembler language. The CalcuList programming environment has been implemented as a small-sized Java project in Eclipse 4.4.1 with 6 packages and 20 classes all together. The Jar File for using CalcuList may be downloaded from the link in [9]. The size of this file is rather small: 134 kb. The draft of a tutorial on CalcuList may be downloaded from the link in [10].

CalcuList has been used since 2011 by the first author as a didactic tool for his course of Formal Languages, thought at the first year of the Master in Computer Engineering at University of Calabria. The course used to focus on basic notions of languages, grammars and compilers and on two styles of declarative paradigms: logic programming (mainly Prolog and Datalog) and a functional language. As it was hard to introduce another language to illustrate the functional paradigm in a limited number of teaching hours, the first idea was to focus on the functional features supported by classical imperative languages. The results were very disappointing: the declarative style was at most adopted as syntactic sugar that flied away at the first serious attempt to replace iteration with recursion. At that point, also considered that students were eager to learn new emerging languages such as Python, the first author invented a new functional language that at the appearance looked as Python, but it did not support any iterative construct. So, to pass the exam, a student had to eventually learn a functional programming language, although with a practical look.

The first version of CalcuList was rather rudimental and a number of features have been later on added year after year on demand. For instance, when it became popular the MapReduce programming paradigm that runs in the background of Hadoop to provide scalability and easy data-processing solutions, higher-order functions were introduced but without the possibility to return a function as this feature was not necessarily required to explain MapReduce. Later on, as the manipulation of JSONs became a new frontier, they were added to CalcuList as first class objects. In addition, to enable the update of a document JSON while filtering it, Global Setting Commands (GSCs) were introduced to perform operations with side effects using a style borrowed by the semantic rules of attribute grammars. In sum, in six years CalcuList has moved from a simple pure functional core to a flexible functional programming environment with some (limited and controlled) imperative features that sometimes may very much simplify a strict pure functional notation.

We conclude by mentioning that, stimulated by the accurate and competent remarks by the three anonymous referees, we already started to work on the following extensions of CalcuList:
1. remove the present limitation that a function cannot return a function – this extension should not require much work;
2. infer types for function parameters and returned value, although we shall not be able to provide a complete inference as we shall preserve the present weak typing for lists and jsons;
3. optimize tail recursion implementation that is presently done in the naive way of assigning a stack frame to every instance of the tail recursive function – this extension only requires some minor rewriting of some CalcuList classes;
4. provide a statically test of whether a function with shallow list operations has side effect or not – the extension is non complex if applied to each function separately from the other ones, but the real challenge is to consider a group of functions where a function may pass a list parameter according to the typical scheme for tail recursion.

References

CPM: A Declarative Package Manager
with Semantic Versioning
– System Description –

Michael Hanus  Jonas Oberschweiber
Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. We present CPM, a package manager for the declarative multi-paradigm language Curry. Although CPM inherits many ideas from package managers for other programming languages, a distinguishing feature of CPM is its support to check the rules of semantic versioning, a convenient principle to associate meaningful version numbers to different software releases. Although the correct use of version numbers is important in software package systems where packages depend on other packages with specific releases, it is often used as an informal agreement but usually not checked by package managers. CPM is different in this aspect: it provides support for checking the semantic requirements implied by the semantic versioning scheme. Since these semantic requirements are undecidable in general, CPM uses the property-based testing tool CurryCheck to check the semantic equivalence of two different versions of a software package. Thus, CPM provides a good compromise between the use and formal verification of the semantic versioning rules.

1 Introduction

Complex software systems are usually not built from scratch but re-use various components. To structure such systems, software packages with well-defined APIs (application programming interfaces) are used. A software package consists of one or more modules and is used as a building block of a larger system. Hence, a software system or complex package depend on other packages. Since packages change over time, e.g., new functionality is added, more efficient implementations are developed, or the usage of operations (i.e., the API) is changed, it is important to use an appropriate version of a package. Finding them and managing these dependencies is a non-trivial problem. As a solution to it, package managers use version numbers associated to package releases and allow to express such dependencies as relations on version numbers.

Semantic versioning is a recommendation to associate meaningful version numbers to software packages. In the semantic versioning standard,\(^1\) a version number consists of major, minor, and patch number, separated by dots, and an optional pre-release specifier consisting of alphanumeric characters and hyphens

\(^1\) http://www.semver.org
appended with a hyphen (and optional build metadata, which we do not consider here). For instance, 0.1.2 and 1.2.3-alpha.2 are valid version numbers. Furthermore, an ordering is defined on version numbers where major, minor, and patch numbers are compared in lexicographic order and pre-releases are considered unstable so that they are smaller than their non-pre-release versions. For instance, 0.1.2 < 0.3.1 < 1.1.2-beta < 1.1.2. Furthermore, semantic versioning requires that the major version number is incremented when the API functionality of a package is changed, the minor version number is incremented when new API functionality is added and existing API operations are backward compatible, and the patch version number is incremented when the API functionality is unchanged (only bug fixes, code refactorings, etc).

The advantage of semantic versioning is an increased flexibility to choose packages when building larger software systems. For instance, if package A requires some functionality which has been introduced in version 1.4.1 of package B, one can specify that A depends on B in a version greater than or equal to 1.4.1 but less than 2.0.0. Thanks to semantics versioning, a package manager can choose newer versions of B (as long as they are smaller than 2.0.0), when they become available, in order to build A.

However, semantic versioning requires that, if some operation $f$ is defined in two versions of a package with identical major version numbers, these two definitions are semantically equivalent. Since this property is obviously undecidable in general, the developer is responsible for this semantic compatibility so that this is not checked in contemporary package management systems. Improving this situation is the objective of the Curry package manager CPM.

In order to check the semantic equivalence of a unary operation $f$ defined in versions $v_1$ and $v_2$ of some package, one can rename the definitions of $f$ in these packages to $f_{v_1}$ and $f_{v_2}$, respectively, and check the property $\forall x. f_{v_1}(x) = f_{v_2}(x)$. Ideally, one should prove this property. Since fully automatic proof techniques are available only for limited domains, CPM uses property-based testing instead. Property-based testing automates the checking of properties by random or systematic generation of test inputs. It has been introduced with the QuickCheck tool [6] for the functional language Haskell and adapted to many other languages, like CurryCheck [8] for the functional logic language Curry. Although property-based testing provides no formal guarantees, in practice it is quite successful if the generated input data is well distributed.

In the following, we briefly survey Curry and CurryCheck before we provide an overview of CPM and its implementation of semantic versioning checking.

## 2 Functional Logic Programming and Curry

Functional logic languages combine the most important features of functional and logic programming in a single language (see [7] for a recent survey). In particular, the functional logic language Curry [11] conceptually extends Haskell

---

2 Although this property is necessary, it is not sufficient to ensure semantic equivalence in functional logic programs [5]. Nevertheless, we use it here for the sake of simplicity.
with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. The syntax of Curry is close to Haskell. In addition to Haskell, Curry applies rules with overlapping left-hand sides in a (don’t know) non-deterministic manner (whereas Haskell always selects the first matching rule) and allows free (logic) variables in conditions and right-hand sides of rules. Function calls are evaluated lazily where free variables as demanded arguments are non-deterministically instantiated [2].

Example 1. The following simple program shows the functional and logic features of Curry. It defines an operation “++” to concatenate two lists, which is identical to the Haskell encoding. The operation ins inserts an element at some (unspecified) position in a list:

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

Note that ins is a non-deterministic operation since it might deliver more than one result for a given argument, e.g., the evaluation of ins 0 [1,2] yields the values [0,1,2], [1,0,2], and [1,2,0]. Curry has many other features not described here, like monadic I/O and modules as in Haskell, set functions [4] to encapsulate non-deterministic search, and functional patterns [3] to specify complex transformations in a high-level manner. For instance, we can provide an alternative and more compact definition of ins with a functional pattern:

ins’ x (xs++ys) = xs++[x]++ys

3 Property-based Testing with CurryCheck

Property-based testing [6] is a useful technique to improve the reliability of software packages. Basically, properties are expressions parameterized over input data. CurryCheck [8] is a property-based test tool for Curry which automates the tests whether properties hold on various inputs. CurryCheck extracts and tests all properties, i.e., top-level entities with result type Prop, contained in a source program. For instance, if we add to the program of Example 1 the property

\[
\text{concIsAssoc} :: [\text{Int}] → [\text{Int}] → [\text{Int}] → \text{Prop} \\
\text{concIsAssoc} xs ys zs = (xs++ys)++zs \Leftrightarrow xs++(ys++zs)
\]

and run CurryCheck on this program, the associativity property of list concatenation is tested by systematically enumerating lists of integers for the parameters xs, ys, and zs. The property “\Leftrightarrow” has the type \(a \rightarrow a \rightarrow \text{Prop}\) and is satisfied if both arguments have a single identical value.

To check laws involving non-deterministic operations, one can use the property “\Leftrightarrow” which is satisfied if both arguments have identical result sets. For instance, the requirement that list insertion increments the list length can be expressed by the property

\[
\text{insLength} x xs = \text{length } (\text{ins } x xs) \Leftrightarrow \text{length } xs + 1
\]

Since the left argument of “\Leftrightarrow” evaluates to many (identical) values, the set-based interpretation of “\Leftrightarrow” is relevant here. This is reasonable since, from a
declarative programming point of view, it is irrelevant how often some result is computed. The semantic equivalence of \texttt{ins} and \texttt{ins'} defined above can be checked with the property
\begin{equation}
\texttt{insSameAsIns'} \ x \ \texttt{xs} = \texttt{ins} \ x \ \texttt{xs} \ \sim > \ \texttt{ins'} \ x \ \texttt{xs}
\end{equation}

4 CPM: The Curry Package Manager

The Curry Package Manager CPM\textsuperscript{3} is a tool to distribute and install Curry software packages and manage version dependencies between them. A CPM package consists of at least one or more Curry modules and a package specification, a file in JSON format containing the package’s metadata. Beyond some standard fields, like author, name, or synopsis, the metadata of each package contains the version number of the package (in semantic versioning format) and a list of dependency constraints. A \textit{dependency constraint} consists of the name of another package and a disjunction of conjunctions of version relations, which are comparison operators (\(<\), \(\leq\), \(\geq\), \(=\)) together with a version number. Conjunctions are separated by commas, and disjunctions are separated by \text{||}. Hence, the dependency constraint
\begin{equation}
\text{"B" : "}\geq 2.0.0, < 3.0.0 \ | | > 4.1.0\text{"}
\end{equation}
expresses the requirement that the current package depends on package B with major version 2 or in a version greater than 4.1.0.

CPM has various commands to manage the set of all packages and install and upgrade individual packages. Since CPM uses a central index of all known packages\textsuperscript{4} and their versions, an important command is \texttt{cpm update} which downloads the newest version of this index. The command \texttt{cpm list} shows a table of all packages (sorted by various criteria which can be specified as command options), \texttt{cpm search} allows to search for a term within all packages, and \texttt{cpm info} shows detailed information of a package.

The command \texttt{cpm install} installs a package by resolving all dependency constraints of the current package and all dependent packages. This is a classic constraint satisfaction problem. CPM uses a lazy functional approach based on [12] to solve all dependency constraints and find appropriate package versions. If there is a solution to these constraints, CPM automatically installs local copies of all required packages (either from a cache or by downloading them from a central repository). If there are several possible versions of some package to install, CPM uses the newest one. There is also a command \texttt{cpm upgrade} to replace already installed packages by newer versions, if possible. The details of these processes are outside the scope of this paper and are described in [13]. Since the number of packages in the current CPM index is limited, we tested our dependency resolution algorithm on a large set of packages (the central package index of \textit{npm}, the Node package manager) and obtained acceptable run times on realistic examples (see [13] for details).

\textsuperscript{3} \url{http://curry-language.org/tools/cpm}

\textsuperscript{4} Currently, CPM manages more than 50 packages and 400 modules.
CPM also supports package testing, documentation, and compilation. The command `cpm test` applies CurryCheck to all source modules of the package (or to some test suite specified in the package’s metadata). The command `cpm doc` generates the documentation of a package, i.e., the API documentation (in HTML format) automatically extracted from source programs and, if provided, manuals in PDF format. If the package’s metadata specifies a main module and the name of an executable, the package and all its dependencies are compiled by the command `cpm install`, which also installs the generated binary in the `bin` directory of CPM. Hence, complete Curry applications can be wrapped in a package so that they are easily installed by a single command.

As mentioned above, CPM adheres to the semantic versioning standard as sketched in Section 1. CPM supports the automated checking of the rules of semantic versioning by the command `cpm diff`. For instance, to compare the current package to a previous version 1.2.4 of the same package, one can invoke the command

```bash
> cpm diff 1.2.4
```

This starts a complex comparison process which is described in the next section.

## 5 Semantic Versioning Checking

Semantic versioning checking is the process to compare the APIs of two versions of some package and report possible violations according to the semantic versioning standard. In the case of Curry, the API of a package is the set of all public data types and operations occurring in the exported modules\(^5\) of this package. The semantic versioning checker of CPM performs the following steps:

1. The signatures of all API data types and operations occurring in both versions of the package are compared. If there are any syntactic differences and the major version numbers of the packages are identical, a violation is reported.
2. If there is some API entity \(f\) occurring in version \(a_1.b_1.c_1\) but not in version \(a_2.b_2.c_2\), then a violation is reported if \(a_1\) and \(a_2\) are identical but \(b_1\) is not greater than \(b_2\).
3. If the major version numbers of the packages are identical, then, for all API operations occurring in both package versions, the behavior of both versions of such an operation is compared (see below). A violation is reported if any difference is detected.

The implementation of the first two steps can be achieved by a straightforward syntactic comparison of the packages. To implement step 3, i.e., to compare the behavior of some operation \(f\) defined in versions \(v_1\) and \(v_2\) of some package, the code of both packages is copied and all modules of these packages (and

---

\(^5\) The metadata of a package can also specify a subset of all modules as “exported” so that only operations in these modules can be used by other packages. If this is not explicitly declared, all modules of the package are considered as exported.
all packages on which these packages depend) are renamed with the version number as a prefix. For instance, a module \( M \) occurring in package version 1.2.3 is copied and renamed into module \( V_{1.2.3}.M \). Thus, if there is a unary operation \( f \) occurring in module \( M \) in package versions 1.2.3 and 1.2.4 to compare, one can access both versions of this operation by the qualified name \( V_{1.2.3}.M.f \) and \( V_{1.2.4}.M.f \). Thus, CPM generates a new “comparison” module which contains the following code:

```haskell
import qualified V_1_2_3_M
import qualified V_1_2_4_M

check_M_f x = V_1_2_3_M.f x <~> V_1_2_4_M.f x
```

Due to the use of the property “<~>”, CPM can also compare the computed results of non-deterministic operations. If this module is passed to CurryCheck and the property is satisfied for all generated test inputs, we have some confidence about the semantic equivalence of \( f \) in both packages. This approach works under the following assumptions:

1. The input and result types of \( V_{1.2.3}.M.f \) and \( V_{1.2.4}.M.f \) are identical.
2. The operations to be compared are terminating on all input values.

Unfortunately, the first assumption is not satisfied if \( f \) works on a type \( T \) defined in module \( M \), since the comparison module contains two copies of this type: \( V_{1.2.3}.M.T \) and \( V_{1.2.4}.M.T \). In order to generate a single property to compare both versions of \( f \), CPM generates a bijective mapping between both renamed types

\[
t_T :: V_{1.2.4}.M.T \rightarrow V_{1.2.3}.M.T
\]

This operation can inductively be defined for all data constructors of type \( T \), since the structure of \( T \) must be identical in both versions (otherwise, semantic versioning is syntactically violated). If \( f \) is of type \( T \rightarrow T \), then CPM generates the following property to compare both versions of \( f \):

```haskell
check_M_f x = V_{1.2.3}.M.f (t_T x) <~> t_T (V_{1.2.4}.M.f x)
```

If the second assumption (termination) is not satisfied, the property tester might not terminate. To avoid this situation, CPM analyzes the operations to be compared before the comparison properties are generated. For this purpose, CPM exploits the Curry analysis framework CASS [10], which provides a simple termination analysis, and generates the above properties only for operations which are definitely terminating. CPM also accepts specific pragmas where the programmer can annotate operations as terminating for cases where the termination checker is not powerful enough.

Unfortunately, this is not sufficient to check operations that are intentionally non-terminating since they generate infinite data structures. In order to check such operations, e.g., stream generators, CPM analyzes the “productivity” of these operations and compare finite approximations of their results. For instance, consider the following operations which generate infinite lists of ascending integers starting from the given argument:

```haskell
ints :: Int \rightarrow [Int]  ints2 :: Int \rightarrow [Int]
ints n = n : ints (n+1)  ints2 n = n : ints2 (n+2)
```
Although these operations compute different infinite lists, this difference cannot be detected by the property
\[
\text{checkInts } x = \text{ints } x \sim< \sim \text{ints2 } x
\]
since its evaluation does not terminate. However, both operations are root-productive: there is no infinite sequence of evaluation steps which does not produce a constructor at the root. A productive operation is one which is root-productive and all operations occurring in derivations of this operation are also productive. Hence, \text{ints} and \text{ints2} are productive whereas \text{loop} defined by
\[
\text{loop } n = \text{loop } (n+1)
\]
is not productive. The productivity property of operations can be approximated by a program analysis with a fixpoint computation on all program rules (see [9] for details).

CPM implements such a program analysis and uses its result to compare also non-terminating but productive operations. For this purpose, it limits the size of the data structures to be compared. For instance, the size of a potentially infinite list can be limited by an operation which has a first “size” argument (represented as a Peano number with the constructors \(\mathit{Z}\) and \(\mathit{S}\)):
\[
\begin{align*}
\text{limitList } \mathit{Z} & \quad \mathit{[]} = [] \\
\text{limitList } (\mathit{S} \ n) \mathit{[]} & \quad \mathit{[]} = [] \\
\text{limitList } (\mathit{S} \ n) \ (x:xs) & \quad x : \text{limitList } n \ xs
\end{align*}
\]
Now one can check the observable equivalence of \(\text{ints}\) and \(\text{ints2}\) by the following property:
\[
\text{limitCheckInts } n \ x = \text{limitList } n \ (\text{ints } x) \sim< \sim \text{limitList } n \ (\text{ints2 } x)
\]
For this property, CurryCheck finds a counter-example for the input arguments \(n=(\mathit{S} \ (\mathit{S} \ \mathit{Z}))\) and \(x=1\). With this approach, CPM can also compare different versions of non-terminating but productive operations. Hence, the overall strategy of CPM to compare two different versions of an operation \(f\) is as follows:

1. If \(f\) is terminating, the results of both versions are directly compared (with a type mapping, if required).
2. If \(f\) is non-terminating but productive, the results of both versions are limited to a given size, where the size parameter is also part of the test inputs.
3. If \(f\) is non-terminating and not productive, both versions are not compared and a warning is issued.

If CPM cannot automatically derive the productivity of an operation, the programmer can explicitly annotate operations as productive so that they are checked with the strategy explained above. More details about this analysis and its implementation can be found in [9].

6 Concluding Remarks

We have presented a software package manager for Curry with support for semantic versioning. Although there exist many package managers which use similar versioning schemes, to the best of our knowledge, CPM is the first package...
manager which provides automated support for semantic versioning checking. The Elm package manager\(^6\) also performs semantic versioning checks but this is based on simple syntactic API comparisons. Hence, it can not detect semantic differences when API types are unchanged, like replacing a decrement by an increment operation.

We have shown that declarative languages in combination with powerful property testing tools are a good basis for automated semantic versioning checking. Hence, our approach can also be transferred to Haskell with QuickCheck \([6]\), Prolog with PrologCheck \([1]\), or Erlang with PropEr \([14]\). For a fully automatic tool, it is necessary to ensure the termination of the checking process. Although this can be achieved by time limits, more powerful checks require a careful program analysis, as we have done with analyzing the productivity of possibly non-terminating operations.

CPM’s semantic versioning checking is a tool that can be used by the package developer to check the changes introduced in a new version of the package. Since it is a fully automatic tool, it can also be used in the workflow to publish new package versions in the central repository of CPM, which is not yet implemented but a topic for future work.

References


\(^6\) http://elm-lang.org/


Declarative XML Schema Validation
with SWI–Prolog
System Description

Falco Nogatz¹, Jona Kalkus², and Dietmar Seipel¹

University of Würzburg, Department of Computer Science,
Am Hubland, D – 97074 Würzburg, Germany
¹{falco.nogatz,dietmar.seipel}@uni-wuerzburg.de,
²jona.kalkus@stud-mail.uni-wuerzburg.de

Abstract. XML Schema is a well–established mechanism to define the
structure and constrain the content of an XML document. While this
approach taken by itself is declarative, currently available tools for XML
validation are not. In this paper we introduce an implementation of an
XSD validator in SWI–Prolog, made publicly available as the package li-
brary(xsd). Our approach is based on flattening the XSD and XML doc-
ments into Prolog facts. The top–down validation makes great use of
Prolog’s backtracking and unification capabilities. To ensure the compli-
ance to the XSD standard and to support the test–driven development,
we have created a test framework based on the Test Anything Protocol
and SWI–Prolog’s quasi–quotations.

Keywords: XML Schema, XSD, XML, SWI–Prolog, Validation, Quasi–
Quotation

1 Introduction

The Extensible Markup Language (XML) [1] is one of the most used data formats
to store and exchange structured data. Especially in the context of web services,
XML documents are often used for data transfer and as configuration files. These
use cases emphasise the importance for tools that ensure an expected format of
the used XML documents.

One approach to specify the structure and content of XML documents is to
use an XML Schema Definition (XSD) [2]. It is used to specify the allowed ele-
ments in an XML document, their data types, and additional rules the document
has to comply with. While every XML document has to be well–formed, i.e. it
has to follow the general syntax rules for XML, XSD is used to ensure validity in
terms of conformity according to the specified data types and rules.

Version 1.0 of the XSD specification was originally published in 2001, a sec-
ond edition followed in 2004. Since then, a great number of tools to validate
XML documents against a given XSD has been published. In 2012, the XSD 1.1
specification [3] became a W3C Recommendation. It introduces new, significant
features like the ability to define assertions based on XPath expressions and conditional type assignments. Although completely backward compatible, these new features require the handling of expressive, declarative rules which can often not be easily added to existing tools, because they are mostly based on imperative programming languages. Therefore, the number of XSD validators which support the most recent XSD 1.1 standard is still limited. Three of the most popular tools with support for XSD 1.1 are: Apache XercesJava, Oxygen XML Editor, and Saxon XSLT.

SWI-Prolog [5] already has a good support for the work with XML. Together with Prolog’s built-in backtracking and unification abilities, this makes it a good target platform for a new, extensible XSD validation software. In this paper, we present an approach to process XML and XSD files using SWI-Prolog. The validation module `library(xsd)` unfolds a given XML and its XSD into a knowledge base representing the documents as Prolog facts. This way it is possible to define declarative Prolog rules that ensure the schema properties for all instance nodes that can be unified, resulting in a validation where the XML nodes are processed in a top-down manner.

To ensure the compliance to the XSD standard, our implementation comes with a test framework based on the Test Anything Protocol (TAP) [6]. It makes great use of SWI-Prolog’s quasi-quotations [7] to directly embed example XML documents into Prolog source code as an external domain-specific language.

Our XSD validator is available as a package for SWI-Prolog and listed in its package list at http://www.swi-prolog.org/pack/list?p=xsd. It can be easily installed using `pack_install(xsd)` and used similar to built-in libraries by calling `use_module(library(xsd))`. The validator is published under MIT License as open source at https://github.com/jonakalkus/xsd.

The remainder of this paper is organised as follows. In Section 2 we introduce the work with XSD and XML files in SWI-Prolog and present possible representations in Prolog. In Section 3, the validation process is described. The embedding of XML into SWI-Prolog using quasi-quotations is presented along with the test framework in Section 4. Finally, we conclude with a summary and discussion of future work in Section 5.

## 2 On the Integration of XML in SWI-Prolog

Prolog is well-known for processing natural language. However, Prolog is also an excellent language to work with data given in a formal language. SWI-Prolog is already widely used to process XML documents. Recently, the integration with semantic data given as RDF/XML [8] in Prolog has been an emerging field of research, resulting in great support for RDF/XML in SWI-Prolog [9,10] and

---


2.1 A Motivating Example

As a motivating example, we will consider a small XSD, as shown in Figure 1. Following the formal description of the XSD language [2], it mainly consists of descriptions of elements, simple and complex types, and attributes. We assume basic knowledge about XSD here and provide only a short, informal description.

The XSD characterises XML documents with a single root node `<person>`, and `<name>` and `<email>` child nodes. The example document given in Figure 2 is valid against this XSD, while the second XML given in Figure 3 is not valid because of its missing `<email>` node and the wrong value abc for the attribute no of the data type id.

```xml
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string" maxOccurs="unbounded"/>
        <xs:element name="email" type="xs:string" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="no" type="id" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="id">
    <xs:restriction base="xs:int">
      <xs:minInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Fig. 1: Example XSD

```xml
<person no="123">
  <name>John Doe</name>
  <email>john@doe.com</email>
  <email>j.doe@example.com</email>
</person>
```

Fig. 2: Valid XML

```xml
<person no="abc">
  <name>John Doe</name>
</person>
```

Fig. 3: Non–valid XML

The aim of library(xsd) is to identify the XML of Figure 2 as valid, and the XML of Figure 3 as invalid. It provides a single predicate xsd_validate(+XSD,+XML) which succeeds only for XML documents that are valid according to the given XSD.

2.2 Parsing XML with library(sgml)

XML is an application profile of the Standard Generalized Markup Language (SGML) [12] and therefore just a subset of SGML. As a result, it is possible to use an SGML parser to load XML files in Prolog. The first SGML parser for SWI–Prolog was created by Anjo Anjewierden and was based on the SP parser4. Today’s versions of SWI–Prolog come with a faster SGML parser implemented as a C–library [13,10]. Both SGML parsers share the same output format and a

4 http://www.jcclark.com/sp/ [accessed 8 July 2017]
similar interface. Since XSD is an application of XML, the SGML parser can be used for both input file formats.

The SGML parser can be used in SWI-Prolog after loading the module `library(sgml)`, which by default is bundled with SWI-Prolog. It provides a predicate `load_structure(+Source,-Out,+Options)` to load structured files like SGML, HTML, or XML. Most importantly we use the options (i) `dialect(xmlns)`, to read in the given files as XML documents using the built-in namespace handling, and (ii) `keep_prefix(true)`, to store the namespace's URI along with the node's type. The latter option was requires SWI-Prolog of at least version 7.3.26.

2.3 Nested Term Representations

SWI-Prolog’s built-in SGML parser returns a nested list. Each node is represented by a Prolog term of the form

```
  element(ns(Prefix,URI):Type,Attributes,Children).
```

For instance, the XSD of Figure 1 generates the following term:

```
  element( ns(xs,'http://www.w3.org/2001/XMLSchema'):schema,
          [ xmlns:xs='http://www.w3.org/2001/XMLSchema' ],
          [ element(
              ns(xs,'http://www.w3.org/2001/XMLSchema'):element,
              [name=seq], [ ... ], ... ) ]
  )
```

Seipel et al. have transformed this data structure into a more convenient form called field notation [14]. It is based on association lists and triples of the form

```
  Type:Attributes:Children
```

and integrates a declarative query mechanism called FNQUERY [15].

In `library(xsd)`, we use a top-down validation approach, where the validator simultaneously traverses the XSD and XML document, beginning with the `<xs:schema>` resp. root node. At first sight, the nested term representation looks like a good data structure for this approach using tree traversal. However, in XSD it is possible to define types globally (like the simple type `id` in Figure 1) which are usually referenced by other elements which are not necessarily part of the same XML sub-document. The same applies for named element references using `<xs:element ref="..." />`. Therefore, element types and names would have to be stored globally.

2.4 XML Flattening

The nested term can be avoided by flattening: the contained elements are asserted as facts with a unique identifier. Based on the identifier it is possible to retrieve, for example, the parent node, all siblings, or any descendant. In addition to this, globally defined types and named elements can be easily accessed.

In [16], Nogatz et al. introduced `xsd2json`, a tool that translates an XSD into an equivalent JSON Schema using Prolog and Constraint Handling Rules
To represent the XSD as CHR constraints, a similar flattening step has been applied. The flattening implemented in `library(xsd)` is for the most part an adapted version, which asserts Prolog facts instead of generating CHR constraints. It can be separately used as `xml_flatten(+XML,?Handle)` in the sub-package `library(xsd/flatten)`. If not provided, it returns a unique identifier `Handle` to reference an already flattened XML file. This handle is part of every asserted fact to distinguish multiple loaded XML files. In the code examples in this paper, we use `xsd` as the handle of a loaded XSD document, and `xml` for the loaded XML document.

The asserted facts are similar\(^5\) to the CHR constraints generated by `xsd2json`:

- `node(Handle, ID, Namespace, Type)`
  For each XML node a new `node/4` predicate is asserted, only holding its `Namespace` and `Type`.
- `node_attribute(Handle, ID, Attribute, Value)`
  For each XML attribute a new `node_attribute/4` is asserted, holding the attribute’s name and value\(^6\).
- `text_node(Handle, ID, Text)`
  If an element’s child is simply a text and no nested XML, a `text_node/3` is asserted with its `Text`.

The node’s unique identifiers `ID` are generated inductively: (i) the root node has an `ID` of `[0]`, and (ii) the `ID` of all other nodes is of the form `[Position|Parent_ID]`, with `Position` starting from 0 and being incremented for every sibling. This way an element’s siblings, ancestors, and descendants can be retrieved by simple unifications based on the element’s identifier.

For instance, the flattening of the XSD of Figure 1 generates the following `node/4` and `node_attribute/4` facts:

```prolog
?- xml_flatten('file.xsd',xsd), listing([node/4,node_attribute/4,text_node/4]).
```

```prolog
node(xsd, [ 0], ns(xs, 'http://www.w3.org/2001/XMLSchema'), schema).
node(xsd, [ 0, 0], ns(xs, 'http://www.w3.org/2001/XMLSchema'), element).
node(xsd, [ 0, 0, 0], ns(xs, 'http://www.w3.org/2001/XMLSchema'), complexType).
node(xsd, [ 1, 0], ns(xs, 'http://www.w3.org/2001/XMLSchema'), simpleType).
node(xsd, [0, 1, 0], ns(xs, 'http://www.w3.org/2001/XMLSchema'), restriction).
% ... and 5 other node/4
node_attribute(xsd, [ 0], xmlns:xs, 'http://www.w3.org/2001/XMLSchema').
node_attribute(xsd, [ 0, 0], name, person).
node_attribute(xsd, [ 0, 1, 0], name, id).
node_attribute(xsd, [0, 1, 0], base, 'xs:int').
% ... and 9 other node_attribute/4
```

Because the XSD of Figure 1 does not contain an XML node with only text content, no `text_node/3` fact is asserted.

\(^5\) `xsd2json` is in active use and maintained. It is available as open source at [https://github.com/fnogatz/xsd2json](https://github.com/fnogatz/xsd2json) (MIT License). Because of recent improvements, the constraint functors have been slightly changed compared to [16].

\(^6\) XML attributes in general do not have any namespaces. For special attributes like the declaration of the namespace prefix `xs` in `xmlns:xs="..."`, this is handled separately.
3 Top–Down Validation by Simultaneous Tree Traversals

To validate an XML against its XSD both documents are traversed simultaneously. The document’s nodes are validated step–by–step, beginning with its root node with the unique identifier of [0], followed by its descendants [0,0], [1,0], and so on. The number of XML and XSD nodes which are involved in a single validation step varies: a single XML element might require several alternative XSD nodes (e.g., in case of <xs:choice> definitions); then again several XML elements can be specified by a single XSD node (e.g., in case of <xs:element maxOccurs="unbounded">).

We expect that both the XSD and XML are well–formed; the XSD is expected to strictly follow the XSD specification. The main predicate xsd_validate/2 internally uses the predicate validate(+S_Handle,+D_Handle) of the sub–package library(xsd/validate). It validates an XML against a given XSD provided by its D_Handle resp. S_Handle. It is implemented as follows:

validate(S_Handle, D_Handle) :-
    validate(D_Handle, [0], 1, S_Handle, [0]).

The rules on how to validate a given XML node are stated using the predicate validate(+D_Handle,+D_ID,?Vals,+S_Handle,+S_ID). Given the two current positions in the XML and XSD trees, specified by the appropriate pair (Handle, ID), we have implemented rules to confirm its validity. The additional argument Vals is an internal counter which is used to ensure, among others, the correct number of elements in a <xs:sequence> with respect to the minOccurs and maxOccurs properties. For common selections we provide predicates like child(+Handle,?ID,?Child), which returns a child of the node with the given ID, and vice versa.

Complex Type Validation. library(xsd) has only a single rule that can be applied for the initial goal validate(xml,[0],1,xsd,[0]):

validate(D_Handle, D_ID, 1, S_Handle, S_ID) :-
    node(S_Handle, S_ID, ns(_, 'http://www.w3.org/2001/XMLSchema'), schema),
    child(S_Handle, S_ID, S_Child),
    node(S_Handle, S_Child, ns(_, 'http://www.w3.org/2001/XMLSchema'), element),
    validate(D_Handle, D_ID, 1, S_Handle, S_Child).

It reads as follows: The current XML position is valid if the XSD is a <xs:schema> node containing a <xs:element> child node which is valid, too. This is correct according to the XSD specification as there can be various root nodes defined in the XSD. They have to be handled as alternatives, i.e. there must be at least one that is valid. If there is one, Prolog’s backtracking mechanism will find the appropriate S_Child and continues the validation at this point.

Rules that handle the validation of complex types contain validate/5 predicates in the rule’s body to recursively validate all the contained elements. The following more complex source code example demonstrates how to validate a single element:
A `<xs:element>` node is only valid if it is allowed at this position according to the `minOccurs` and `maxOccurs` properties set in the XSD’s `attribute/4`. In addition, the element must be valid itself, i.e. of the correct type, etc. In the last part, all sibling nodes referenced by the same XSD position are ensured to be valid, too.

The `attribute/4` is a wrapper for the asserted `node_attribute/4` predicate. It takes into account default values according to the XSD specification. If, e.g., a `minOccurs` is set explicitly in the XSD document via `node_attribute/4`, it is returned by `attribute/4`, otherwise the default value of 1 is used.

**Simple Type Validation.** The leaves of an XML document tree are mostly formed by elements of simple types. `library(xsd)` provides the sub-package `library(xsd/simpletype)`, which validates XSD types like `xs:int`, `xs:string`, etc. It also considers constraining facets like `<xs:minInclusive>`. For pattern-based restrictions as they are used by `<xs:pattern>`, or XSD’s `xs:date` and `xs:time` data types, we make use of SWI-Prolog’s `library(regex)`.

**Backtracking.** In the tool `xsd2json` as presented in [16], Nogatz et al. also flattened a given XSD in order to translate it into an equivalent JSON Schema. Instead of asserting Prolog facts, the `node`, `node_attribute`, and `text_node` terms are propagated as CHR constraints. The XSD document is later translated using a tree traversal, too. However, the XSD validation in `library(xsd)` makes great use of backtracking which would not be possible in CHR which is a committed-choice language. E.g., when validating elements and sequences with overlapping `minOccurs` and `maxOccurs`, there is often not just a single rule which could be applied. There are also XSD elements which define alternatives explicitly, e.g., in `<xs:choice>`, or the constraining facet `<xs:enumeration>` These alternatives are directly supported by Prolog’s built-in backtracking mechanism.

**Performance Improvements using Memoisation.** Prolog’s backtracking technique allows a compact definition of the validation rules. However, once the backtracking has to be done, part of the already inferred knowledge gets discarded, even though there are some sub-goals which might occur identically in later computations again. This behaviour can be observed especially for XSD documents with nested `<xs:sequence>` or `<xs:choice>` nodes with high `maxOccurs` properties.

We implemented a wrapper which stores already computed validations in a dynamic predicate `xsd_table(Original_Call,Valid). If `validate_tabled/5`

---

7 https://github.com/mndrix/regex [accessed 8 July 2017], The Unlicense.
is called with arguments that have been checked before, its result
Valid={true, false} is returned immediately:

```prolog
:- dynamic xsd_table/2.
validate_tabled(D_Handle, D_ID, Vals, S_Handle, S_ID) :-
    ( xsd_table(validate(D_Handle, D_ID, Vals, S_Handle, S_ID), Valid) ->
        !, call(Valid) % still trigger backtracking if invalid
    ; validate(D_Handle, D_ID, Vals, S_Handle, S_ID) ->
        asserta(xsd_table(validate(D_Handle, D_ID, Vals, S_Handle, S_ID), true))
    ; asserta(xsd_table(validate(D_Handle, D_ID, Vals, S_Handle, S_ID), false)),
        !, false ). % trigger backtracking
```

This memoisation technique is possible only because an XML fragment is valid
against a given XSD fragment independently of its surrounding elements. The
triple (D_ID, Vals, S_ID) is unique and it is not possible to be valid once and
invalid later, or vice versa. In edge cases with many nested <xs:sequence> or
<xs:choice> nodes, this saves up to 98% of the computation time.\(^8\)

Compared to the traditional tabling implementations in Prolog [18,19], this
technique also stores failing computations. SWI-Prolog’s current tabling imple-
m entation only stores goals which can be inferred. It is therefore not possible to
use its library(tabling) to both store failing goals as well as retain Prolog’s back-
tracking semantics, since call(false) will prevent the addition of any tabled
predicate.

4 Test Framework Using Quasi–Quotations and TAP

library(xsd) has been developed in a test–driven approach. Currently its compli-
ance to the XSD standard is ensured by more than 350 tests. Their definitions
and the provided test framework take more than three times the lines of code as
the core library. It has been used in a continuous integration environment using
the Travis CI\(^9\) service.

We have implemented a test framework based on the Test Anything Proto-
col (TAP) [6]. The SWI-Prolog package library(tap)\(^10\) generates a TAP–conform
text output. This interface is supported by a wide range of tools for running,
rendering and analysing the test results.

The test framework is based on normal XSD and XML documents. XSD docu-
ments can be directly used. Since a single XSD should test only a small, specific
aspect of the validator, it is possible to define various test cases for each XSD,
e.g., satisfactory and failing documents. To place all XML test documents in a
single Prolog file, we have used quasi–quotations [7]. They had been added to
SWI-Prolog in version 6.3.17 and are a good mean to embed external domain–
specific languages into SWI-Prolog without any modification [20,21]. This way

---

\(^8\) library(xsd) provides the options ‘without-tabling’(Bool) and profile(Bool).

The example in /test/example/choice_minmax returns: without memoisation 0.55s
with 3,628,657 inferences; with memoisation 0.01s with 50,370 inferences.

\(^9\) https://travis-ci.org/ [accessed 8 July 2017]

\(^10\) https://github.com/mndrix/tap [accessed 8 July 2017], The Unlicense.
the XML can be easily annotated directly from within Prolog. The example XML of Figure 3, which should be recognised as non–valid, is embedded into the Prolog source code of our test framework using the following snippet:

```prolog
'missing email node'(fail): { xml 
  <person no="abc">
    <name>John Doe</name>
  </person> }
```

5 Conclusion and Future Work

In this work, we have presented a declarative approach for Xsd validation in Swi–Prolog. Due to its backtracking and unification mechanisms, Prolog suits very well for implementing an Xsd validator which processes the Xsd and XML document simultaneously in a top–down manner. We have presented an alternative XML representation in Prolog. It is based on three dynamic predicates which are asserted for the given nodes. This flattening results in a non–nested representation that can be easily queried. With the help of the presented inductive rule to generate new unique identifiers, it is simple to find all ancestors, descendants and siblings of a given XML node using unification.

The library(xsd) is available at https://github.com/jonakalkus/xsd and published under MIT License. It requires Swi–Prolog of at least version 7.3.26. Because it has been developed in a test–driven approach, it provides a decent test framework with currently more than 350 tests. Although not yet feature–complete, this covers the bigger part of XML Schema 1.0.11

To support features of the not yet widely adopted Xsd 1.1 standard, library(xsd) currently misses support for XPATH expressions. Although Swi–Prolog provides a library(xpath), it is not compatible with our flattened representation of the XML documents.

The current implementation uses a memoisation technique which is based on the assertion of a dynamic table predicate. As mentioned in Section 3, it might be worth striving for the addition of failing computations to Swi–Prolog’s native tabling implementation, too.

References


11 A list of currently supported Xsd features can be found at https://github.com/jonakalkus/xsd/blob/master/FEATURES.md [accessed 8 July 2017].
Abstract. In general, even though Prolog is a dynamically typed language, predicates may not be called with arbitrarily typed arguments. Assumptions regarding type or mode are often made implicitly, without being directly represented in the source code. This complicates identifying the types or data structures anticipated by predicates. In consequence, Covington et al. proposed that Prolog developers should implement their own runtime type checking system.

In this paper, we present a re-usable Prolog library named plspec. It offers a simple and easily extensible DSL used to specify type and structure of input and output arguments. Additionally, an elegant insertion of multiple kinds of runtime checks was made possible by using Prolog language features such as co-routining and term expansion. Furthermore, we will discuss performance impacts and possible future usage of these annotations.

Keywords: Prolog, runtime checks, type system, data specification

1 Introduction

In general, even though Prolog is a dynamically typed language, predicates may not be called with arbitrarily typed arguments. Assumptions regarding type or mode are often made implicitly, without being directly represented in the source code. In general, calling a predicate with an unintended argument might lead to stack overflows, infinite loops or any kind of undesired behavior. This complicates identifying the types or data structures anticipated by predicates.

For instance, assume you want to call a Prolog predicate in a newly acquired library. Documentation reveals that it implements the desired functionality, yet the call fails. The cause is ambiguous: it could be that the input was as intended, but no solution exists. Another possibility is that the input is unintended, however a call to a transformation predicate beforehand would have solved the issue.

Ideally, available documentation can be used to resolve any ambiguities. However, documentation in natural language has its limits: it cannot convey the entirety of information precisely and often gets outdated when changes are made
to the code. As an example, consider the following excerpt taken from the docu-
mentation of member/2 as implemented in SWI-Prolog [19]: “member(?Element, ?List) is true if Element occurs in the List.”

One issue is that behavior is entirely undefined in case the second argument is not a list. In consequence, one cannot distinguish between failures such as member(a, [b,c,d]), where the second argument is a list but does not contain the element a, and member(a, a), where the second argument is not a list.

In its current implementation, the predicate succeeds even if the second argument is not a proper list, i.e., a list not terminated by []. In consequence, a call such as member(a, [a,b|x]) is successful. Judging by the documentation alone, it remains unclear whether this is intended.

To overcome the limitations of documentation and to gain automatic verification, Covington et al. proposed that Prolog programmers should implement their own ad-hoc runtime type system [3]. Instead, we argue that by making use of Prolog language features, a simple and easily extensible DSL can be shipped as a reusable library called plspec.

The library is open source and freely available under MIT license. It can be downloaded from the GitHub Repository found at https://github.com/wysiib/plspec. It has been tested with both SWI Prolog and SICStus Prolog.

plspec is heavily influenced by clojure.spec [5], which was recently added to Clojure. The motivation for clojure.spec is similar to the one for plspec. Both languages are dynamically typed, often rendering it hard to identify which data should be passed to functions and what values are returned. Additionally, nested data structures can be large and confusing to inspect without tool support. Both libraries enable describing data based on construction out of small and simple building blocks. clojure.spec utilizes functions as building blocks, while plspec maintains a database of specifications described by Prolog terms.

In Prolog, we can insert runtime checks in order to distinguish between failures due to no existing solution, and those where no solution ever could exist because the input data is not handled at all. Furthermore, we can check whether variables are bound to invalid values inside of the called predicate. These kinds of errors might be hidden if the predicate fails later on due to unrelated reasons. Finally, we can add guarantees that if a predicate was called in a certain way and succeeds, variables will be bound to data in a specific format.

Note that plspec is more than a simple type checker for Prolog’s type system. Rather, it can be seen as an additional optional [2] dependent type system:

- plspec does not change the semantics of annotated Prolog programs in any way.
- plspec’s annotations are entirely optional. In particular, one can only partially annotate predicates.
- Specs may be instrumented in order to take into account runtime values. In this case, plspec specifications define a system of dependent types.

In the following, we will focus on how plspec’s annotations can be instrumented for different types of runtime checks, including traditional contracts [12] by specifying pre- and postconditions as well as invariants on variables.
2 Usage and Semantics

Our goal is to associate predicates with information regarding type, form and mode of arguments, most importantly what a valid argument looks like.

In order to describe data, we use so-called specs. A spec is either defined by a programmer by registering it via an interface predicate, a combination of multiple existing specs or one of following built-ins.

2.1 Built-in Specs

We implemented most predicates that can be used to examine terms as atomic specs. These are float, integer, number, atomic, atom, var, nonvar and ground. To verify that a term matches its spec, we call the built-in Prolog predicates with the same name, ensuring that these specs bear the common meaning and are easy to understand. Additionally, we add any to describe any Prolog term.

Furthermore, one can describe non-scalar data using recursive specs. The spec list(X) is matched if and only if the value is a (potentially empty) list of elements satisfying the spec X. Lists with a fixed length can be described via tuple(X), where X is a list of specs which describe the element in that position. As an example, tuple([integer, atom]) is matched by the value [3, a], but neither [a, 3] nor [3, a, b].

Compound terms can be described via compound(X), where X is a compound term with the functor the term shall have. Its arguments have to be specs that describe what kind of data should be contained in that position of the term. For example, compound(foo(atom, var)) is matched by foo(bar, X).

Finally, specs can be combined with so-called connectives. So far, built-ins are and(X) and one_of(X), where X is a list of specs. In the case of and, all specs have to be matched. For one_of, it is sufficient if at least one spec is fulfilled.

2.2 Preconditions

Preconditions are a way to overcome the problems presented in Section 1. The idea is that all valid combinations of arguments to a predicate should be enumerated by the developer. In Prolog, there are multiple ways to call a predicate regarding instantiation of variables. However, with preconditions the developer can clearly state which calls were considered during implementation and testing.

In consequence, when using specs we can be sure that a failure of a predicate with a fulfilled precondition is intended behavior and, analogously, if the precondition is violated it is a type error.

In order to define a precondition, the interface predicate spec_pre/2 is used. Apart from the predicate, it takes a list of specs as an argument which can be understood as the argument vector passed to the predicate. It is allowed to specify multiple preconditions with the semantics that at least one precondition has to be matched. Otherwise, the error handler is called. For preconditions, the value a predicate is called with is passed to the predicate implementing the spec immediately.
An example is shown in Fig. 1. We define a predicate `even_pred/1` that succeeds if the parameter is an even integer and fails for odd integers. In particular, the meaning of the spec is that only integer values are valid parameters. Otherwise, no guarantees are made whether there is correct behavior in this call, may it be failure or throwing an exception.

Thus, if we pass a variable to the annotated predicate, we do not get an exception from `is/2` that the arguments are not sufficiently instantiated but rather a print and an exception from `plspec`. This standard error handler can be replaced by a custom one, for example one that calls `trace` in order to start the debugger at this particular point in the program.

### 2.3 Invariants

Invariants have a more sophisticated semantic: intuitively, they specify the data structures that the predicate should work with. As soon as variables are bound to a value, they are checked as far as possible according to the spec. If the binding involves other variables, their check will be delayed until they get bound.

When a variable is bound to anything that cannot satisfy the spec anymore, the error handler will be called. One can specify invariants via `spec_invariant/2`. Again, the second argument is a list of specs with the same interpretation as above, i.e., for invariants the spec predicate is only called with ground values.

This allows uncovering the kind of programming error shown in Fig. 2: there, we call the predicate `invariant_violator` with an anonymous variable. In the first rule, it will be bound to the list `[1]`. However, the specification of the argument to `invariant_violator` says that it should be atomic if bound. Since `[1]` is neither a variable nor atomic, the error handler is called.
If we would not specify this invariant, the first rule would fail since \([1]\) is not equal to \([2]\). Thus, Prolog would backtrack into the second rule and bind the variable to the atomic value \(a\). The invalid binding of \(X\) to \([1]\) could not be determined without reading the source code. In particular, unit tests could never expose this issue. This kind of programming errors might trigger unintended co-routines whose effects might be hard to pinpoint.

Invariants are implemented by making use of co-routines. Thus, if the Prolog implementation does not support this feature, only pre- and postconditions are available. If the application itself uses co-routines, the effect depends on the execution order. However, as long as these co-routines do not fail beforehand, it has no influence on \textit{plspec}.

2.4 Postconditions

While we ensured correct calls of predicates with preconditions and that variables are never bound to “incorrect” values with invariants, postconditions are an important contract that if a certain condition held upon entry of a predicate, a second condition is implied on success. As for preconditions, the resulting value is used in order to call the predicate implementing the spec.

In particular, this allows to specify a promise that variables will be bound to values of a specified type. No promise is made if the predicate fails since no variables are bound then.

In \textit{plspec}, one can use one or more instances of \texttt{spec_post/3} for postconditions. Apart from the predicate, it takes two lists of specs understood as argument vectors. The semantic is that if the first list of specs matches when the predicate is called, the second list of specs has to match if the predicate succeeds.

In Fig. 3, we define two postconditions for an implementation of the member predicate. The first postcondition guarantees that if the predicate succeeds and
`:− spec_post(mymember/2, [var, any], [list(any), any]).`
`:− spec_post(mymember/2, [list(int), var], [list(int), int]).`  

`my_member([H|_], H).  
my_member([_|T], E) :-  
    my_member(T, E).`

**Fig. 3.** An Example for Postconditions

`:− defspec(tree(X), one_of([[compound(node(tree(X),X,tree(X)))]),  
                             atom(empty)))).`

**Fig. 4.** A Spec for a Tree of a Given Type

the first argument was a variable, then it will be bound to a list. A different promise is made in the second precondition: if now the first parameter of the call is a homogeneous list of type `int`, the second one is a variable and the predicate succeeds, then the variable will be bound to a value of type `int`.

### 3 Implementation

Specifications which are readable and easy to understand are useful for documentation purposes without any additional code being executed. In this section, we will explain how we maintain the spec database, how specs are validated and how we instrument the annotations described in Section 2 for runtime checks.

#### 3.1 Maintenance and Addition of Specs

Specs are stored in Prolog’s fact database. For simplicity, we distinguish between different kinds of specs that are handled separately. The reason for this is that they have different roles. Since `plspec` was designed with extensibility in mind, users can define specs themselves and add them to `plspec` dynamically.

In the following, we present the reason for distinguishing between different kinds of specs and present each of them. Built-in specs are implemented in the same way users could implement them without modifying `plspec`’s source code.

**Aliasing defspec/2** allows defining new specs via composing existing ones. The first argument is an alias for the resulting spec, while the second argument consists of other specs. Recursive specs are allowed. However, they should consume at least one bit of information of a term in order to avoid infinite loops.

A built-in alias for `integer` is `int`. In the database, they are stored as a dynamic fact that maps the alias to the composition of specs. If an alias is encountered by the verification predicate, it just looks up its definition and continues with that spec.
Newly defined specs might also be compound terms which pass information, e.g., inner specs, to the other specs in form of variables. As an example, Fig. 4 shows how to define a spec for a tree of elements of a given type. A tree is defined to be either the atom `empty` or a compound term with the functor `node` and three arguments: the first and last argument are trees of the same type, whereas the middle argument is any value of the given type.

Valid values for `tree(int)`, a tree of integers, include `node(empty, 1, empty)` and `empty`. Neither `node(empty, not_an_integer, empty)`, where the middle value is not of the given type, nor `tree(empty, 1, empty)`, where the functor does not match, are valid.

**Verification via Predicates** Another option is to implement a spec via a predicate that succeeds if a value is valid and fails otherwise. This can be achieved with `defspec_pred/2`, where the first argument is the new spec and the second is the predicate used for validation, possibly with some arguments specified.

Again, new specs might be compound terms and pass information to the predicate. The value that should be checked will always be appended as last argument to the predicate call.

Note that this implementation of specs is only suitable for values that are bound in a single unification step. Otherwise, another mechanism should be used as shown below. As an example, we can reuse the predicate `even_pred/1` from Fig. 1 which tests whether an integer is even or not. In order to use this predicate as a spec, it can be defined by `:- defspec_pred(even, even_pred)`.

Then, every time the spec `even` is used, `even_pred/1` is called with the value as argument. If it fails, the value is considered invalid. Since `even_pred/1` was annotated earlier, it will throw an exception if the value is not an integer.

Regarding built-ins, most atomic specs like `integer` or `nonvar` are implemented this way. When such a spec is encountered in `plspec`, the predicate is simply called with the current value.

Thus, this predicate should not have any side-effects or bind variables used in the passed term which might fire additional co-routines. In fact, checking specifications at runtime should not interfere with the execution of the annotated program in any way. In order to ensure this, we copy each term before using it to check a `plspec` annotation. If the spec predicate succeeds, the original term is compared to its copy. If a variable was bound, an error message will be printed.

**Recursive Spec Predicates** The third way to define specs is more involved. If a value is not bound in a single unification step but rather “consumes” only some part of the value, an appropriate spec can be registered by calling `defspec_pred_recursive/4`.

Recursive specs can be implemented based on a predicate verifying a part of the property, the “consumption” mentioned above. Afterwards, it hands back control to `plspec` and exposes new specs and variables that should be checked.

This predicate is the second argument to `defspec_pred_recursive/4`. It will be called with all arguments directly wired in the spec definition. Additionally,
the value is passed to the predicate. The last two arguments to that predicate are two variables. The first variable should be bound to a list of specs and the second variable to a list of values which might still be variables themselves. \textit{plspec} will take these values and check them against the returned specs.

The third argument to \texttt{defspec\_pred\_recursive/4} is a predicate which merges the results of those checks. The basic operations \texttt{and} as well as \texttt{or} already are implemented and can be used. If a property like “exactly \textit{m} out of \textit{n} specs shall be true” is desired, this predicate has to be implemented by the user.

Finally, the fourth and last argument is the merge predicate which is called for invariant checks. It has to account for the fact that values might not be fully instantiated yet. In \textit{plspec}, this predicate is implemented using co-routines in order to wait for further instantiation of the data to be verified. \texttt{and\_invariant} as well as \texttt{or\_invariant} are already implemented.

Internally, we implemented the checks for compound terms, lists and tuples like this. The functor of a compound term is immediately checked. Following, the specs of its arguments and the current values are returned because they might involve variables that are bound later.

As an example, consider the spec \texttt{list(int)} and the value \([1,X|T]\). A given list is deconstructed as far as possible in order to check the outer spec, i.e., the value is actually a list. Then, the inner spec \texttt{int} is repeated for all elements. Here, we check that both 1 and \texttt{X} are integers. Since \texttt{X} is a variable, this check is handled by a co-routine that fires once \texttt{X} is bound. In presence of non-instantiated tails, the outer spec is kept and delayed until further instantiation. This means, a co-routine is set up that recursively checks that \texttt{T} also matches the spec \texttt{list(int)}. The spec \texttt{tuple(_)} is implemented similarly. In both cases, the resulting specs need to be merged with \texttt{and}.

**Connectives** Connectives are specs that do not consume any part of a value. While they are implemented exactly like the recursive specs above, they are stored separately. Many connectives might have infinite equivalent specs, e.g., \texttt{int} is the same as \texttt{or([int])} and \texttt{or([int, int])}. Thus, connectives are avoided when enumerating possible specs for a value.

These kind of specs are registered by calling \texttt{defspec\_connective/4}, where arguments and semantics exactly match those of \texttt{defspec\_pred\_recursive/4}.

As above, built-in examples are \texttt{one\_of} as well as \texttt{and}, which allow specifying at least one or all specs have to match a value. \texttt{one\_of} is implemented with \texttt{or} as the merge predicate.

### 3.2 Instrumenting Specifications for Runtime Checks

In order to insert runtime checks for the properties specified in \textit{plspec} annotations, we make use of term expansion, i.e., source-to-source transformation.

Since annotations can also function as plain documentation, the user can explicitly state which predicates should be expanded by inserting runtime checks utilizing the given annotations.
We will explain the term expansion on the example of the second, recursive rule of our my_member/2 predicate shown in Fig. 3.

Consider Fig. 5: in lines 2–5, we check whether any precondition is specified. If there is at least one precondition, the plspec_some call will check whether at least one precondition is satisfied and an error is thrown. If no precondition was satisfied, no check will be performed. The check will simple try to conform each spec with each value the predicate was called with.

Afterwards, specified invariant checks are set up in lines 6–8. Note that there is no call to an error handler yet. Instead, the check and potential error handling happens inside of co-routines which will be described in more detail later.

The unification with the head of the rule happens in line 9. Note that A and B in line 1 are fresh variables. Otherwise, if the arguments do not unify with the head, we would not have an opportunity to catch potential errors there.

In line 10, the premises of the implications stated for postconditions are verified. Conclusions of the postconditions and whether they hold are checked again in line 12. The error handling for postconditions is not shown here because it is part of the check_posts predicate. Between these two steps that verify the postcondition, the original goal remains in line 11. This ensures the correct values are used for both parts of the postcondition.

3.3 Co-Routining for Invariants

Invariants are violated as soon as variables are bound to incorrect values. This can be checked by setting up a number of co-routines.

defspec_pred is a special case of defspec_pred_recursive: it consumes the entire value in one go without producing new values. The trade-off is that values for this kind of spec must be bound in a single step. Otherwise, the co-routine that blocks until the value is not a variable anymore fires on a partially instantiated term and fails. On the other hand, blocking until a value is ground does not catch errors where partial instantiation is undesired. This allows easy implementations because no internal structure of a term has to be exposed.

On the other hand, defspec_pred_recursive produces new specs and new values. For example, one can bind a variable to a compound term with a given
and_invariant([], [], _, true).
and_invariant([HSpec|TSpec], [HVal|TVal], Location, R) :-
  setup_check(Location, ResElement, HSpec, HVal),
  and_invariant(TSpec, TVal, Location, ResTail),
  both_eventually_true(ResElement, ResTail, R).

both_eventually_true(V1, V2, Res) :-
  when((nonvar(V1); nonvar(V2)),
    (V1 == true -> freeze(V2, Res = V2)
    ; nonvar(V1) -> Res = V1
    ; V2 == true -> freeze(V1, Res = V1)
    ; nonvar(V2) -> Res = V2)).

Fig. 6. An Implementation of and Based on Co-Routines

functor but bind its arguments later on. These arguments as well as their corresponding specs have to be exposed to plspec, that will set up new co-routines on them in return. This way, all invalid bindings of variables can be accounted for.

The tricky part is that results of subterms usually only propagate one at a time. If the third argument of a compound term is bound incorrectly, but the first argument remains a variable, plspec has to immediately fail. Otherwise, the first variable might not be bound at all and the error would go unnoticed.

Thus, there is a need for a second merge predicate that is able to deal with co-routines. An implementation that merges the results with the connective and is shown in Fig. 6.

The predicate setup_check will set up co-routines in the same way as the original spec did, using the exposed structure of terms. If the check succeeds, ResElement is bound to true or, otherwise, an error term containing a reason.

The connective is chained between the results. For example, if the term foo(1, a, X) is matched against compound(foo(int, atom, var)), the predicate int(1), atom(a), var(X) is formed. Each of the three calls is set up individually using its own co-routine. As soon as one fails, the entire formula is false and all co-routines are terminated by unifications in both_eventually_true.

Analogously, in order to implement or, a single true suffices in order for the formula to be true and to terminate all co-routines that were set up on the other disjuncts. Additionally, it has to be propagated when all disjuncts fail in order to throw an error. However, it is enough to check all alternatives only when we can determine all of them. Because we only want to raise an error if the entire disjunction evaluates to false but one alternative cannot be evaluated yet, we can understand non-termination as “still possible”.

4 Performance Impact

Since all specs are checked at runtime, naturally there is an overhead. In this section, we discuss which predicates should be annotated by measuring the per-
formance impact caused by the runtime checks of pls
spec. As a first example, we consider member/2 that succeeds if the second argument is a list and this list contains the first argument.

In Fig. 7, the definition of member/2 is shown. Additionally, we define a predicate member_entry/2 that wraps the member/2 predicate. One could argue, that valid calls to member/2 should have a list as a second argument. While it is totally sound that the predicate just fails if the second argument is not a list, in most cases such a call indicates a programming error somewhere in the code.

Thus, we add annotations to member/2 and, analogously, to member_entry/2 as shown in Fig. 8. The spec_pre directive allows that the element might be of any type, but the second argument is either a variable or a proper list. Secondly, spec_invariant ensures that if the second argument is bound, it still has to be possible for it to become a proper list. Lastly, spec_post guarantees that if the predicate succeeded for any input, that the second argument will be a proper list.

:- spec_pre(member/2, [any, one_of([var, list(any)]))].
:- spec_invariant(member/2, [any, list(any)]).
:- spec_post(member/2, [any, any], [any, list(any)]).

Fig. 7. Definition of member/2

Fig. 8. Possible Specs of member/2

:- spec_pre(reverse/3, [list(any), list(any), var]).
:- spec_pre(reverse/3, [var, list(any), list(any)]).
:- spec_invariant(reverse/3, [list(any), list(any), list(any)]).
:- spec_post(reverse/3, [list(any), list(any), var],
               [list(any), list(any), list(any)]).
reverse(L, Rev) :-
     reverse(L, [], Rev).
reverse([], Acc, Acc).
reverse([H|T], Acc, Rev) :- !,
     reverse(T, [H|Acc], Rev).

Fig. 9. Annotated Version of reverse
Table 1. Runtimes and Inference Count of Multiple Kinds of Annotations.

<table>
<thead>
<tr>
<th>Program</th>
<th>Index</th>
<th>Runtime (msecs)</th>
<th>Inferences</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>len=10</td>
<td>len=100</td>
<td>len=1000</td>
</tr>
<tr>
<td>member</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>member-entry</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>member-recur</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>reverse</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>reverse-entry</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>reverse-recur</td>
<td>2</td>
<td>121</td>
<td>11903</td>
</tr>
</tbody>
</table>

We consider three benchmark configurations: first, the predicate is not annotated with a spec. Second, a spec is applied to the entry point, but not the recursion. Third, the spec is checked in each recursion step.

These calls are made to member/2 with an integer Index and a list of integers ranging from 1 to N, and to reverse/2 with the same list and a variable. Additionally, we benchmarked calls to reverse with an accumulator that is implemented and annotated as in Fig. 9. For the entry level benchmark, we only annotate reverse/2, dropping the second spec in each of the argument vectors. Each run is repeated ten times and the median runtime is given.

All benchmarks were run on an Intel(R) Core(TM) i7-7700HQ CPU running at 2.80GHz. We used SWI Prolog version 7.2.3 and configured it to use increased stack size by starting it with the parameters -G100g -T20g -L2g. Benchmarks were run sequentially to avoid issues due to scheduling or hyper-threading.

Table 1 depicts the results of the benchmarks. Columns show the length of the list split by runtime of the query as well as amount of inferences. For the member predicates, lookups of different indices are benchmarked in each row.
The programs “member” and “reverse” stand for the original predicates without annotations, whereas the suffix “entry” and “recur” distinguish between the annotation at entry-level and recursion-level respectively.

As depicted in Table 1, for both member and reverse, the amount of additional inferences and runtime is about constant if only the entry level is annotated. While still growing linearly in size with the list, impact remains reasonable.

However, if the specs are checked in every single recursion step, for member, the overhead quickly grows linearly in the length $l$ of the list as well as linearly in the index $i$ that is looked up, causing a quadratic overhead of $i \times l$.

The overhead for reverse actually grows quadratic in the size of $l$. This is because in every step, the entire list without its head is validated against the spec again. We can clearly see that this becomes very slow even if list size increases moderately and such instrumentation should be avoided.

Since this overhead is enormous, recursive predicates should not be annotated. Instead of checking the same property again and again, one can annotate an invariant on entry level. Then, the performance impact is barely noticeable.

5 Related Work

The idea of integrating runtime checks based on annotations into Prolog is not new. Annotating pre- and postconditions has, for instance, been suggested in [9].

In contrast to our approach, the authors extend the usual notion of pre- and postconditions by annotations attached to the Prolog ports for fail and redo. In consequence, they work closer to the execution model of the underlying Prolog interpreter. Furthermore, the author provides the calling context, e.g., the parent predicate, to the specification under test. This allows for more fine-grained reasoning. Our approach on the other hand provides checking of invariants at any point of Prolog execution by means of co-routines.

The work around assertion checking in CiaoPP [16], uses abstract interpretation to try to discharge assertions at compile time. Assertions which cannot be checked statically are performed at runtime, using program transformation. To our knowledge, CiaoPP only supports Ciao Prolog. While plspec requires co-routining for its full functionality, pre- and postconditions work with any Prolog implementation that supports term expansion.

A different approach to testing has been followed in [13]. In contrast to our approach, the authors do not focus on the introduction of runtime checks into Ciao Prolog, but rather try to unify unit testing and runtime checking. This way, only one kind of annotation is needed for different testing purposes. We extend upon this work by the introduction of invariance annotations and the ability to use connectives as discussed in Section 3.1. So far, we have not evaluated if we can extract unit tests from our annotations, but intend to do so.

Documentation of Prolog code has been considered in [18], where the authors introduce PDoc, a documentation format used for literate programming. The corresponding Prolog package has since been included in SWI Prolog. Instead of integrating documentation into the Prolog code itself, the \LaTeX{} package \textit{pl} [14]...
embeds code into the documentation. Using the package, a single source file can be run both by any \LaTeX \penalty0 binary and a Prolog interpreter.

Aside of Prolog, other declarative logic programming languages feature comparable systems. Mercury [17] includes a type system [7,4] together with a set of mode annotations [15]. However, the type system implemented in Mercury differs from the one we suggested: Though it supports higher-order functions, it neither allows types to be defined by a predicate nor to define a union of two types. In contrast to plspec, Mercury allows for type variables to be used. This makes it possible to specify, for instance, that the output of a function will have the exact same type as the input, regardless of the type itself.

Similar annotations to those in plspec can be found in Erlang’s type specification language [8]. These are used, e.g., in the program analyzer Dialyzer [11]. In Erlang, it is only possible to create new types by defining a union of two existing types, which may be pre-defined or an atomic singleton like the number 42 or the atom foo. As discussed in Section 3.1, plspec allows to define a type for all values that fulfill a given predicate.

Furthermore, Erlang allows specifying types for higher-order functions which plspec does not support. Function specifications in Erlang can be regarded as pre- and postconditions in plspec. Just like Mercury, Erlang supports type variables.

6 Future Work

While plspec is capable of exposing real errors in real world Prolog applications, several improvements to the library should be made:

- The default error messages have room for improvement. Whenever possible, the smallest subterm that makes a spec invalid should be included separately. This allows developers to identify faster and easier what went wrong.
- We can imagine adding further annotations. For example, it can be desired that co-routines are terminated when a certain predicate succeeds or that predicates must never fail given their precondition is fulfilled.
- In Section 4, we found that checking annotations of recursive predicates is very slow. If we added static analysis or used gradual typing, most of that overhead could be avoided.

Apart from documentation and runtime checks, there are several applications that could benefit from these annotations and may be subject of future research.

It is desirable that for existing code, one does not have to write specs by hand. Due to the logic and declarative nature of Prolog, we can easily find matching specs to a given value by calling the verification predicate with a variable for the spec. While this allows us to generate a spec for a given value, it is not yet possible to generate a spec that matches all elements in a series of data.

If this functionality existed, one can think further: with additional tool support, specs as well as entire contracts could be inferred, for example, simply by running unit tests that contain only calls which are known to be valid.
Furthermore, some of these annotations could be re-usable for a partial evaluator such as LOGEN [10]. An issue with LOGEN is that even though its binding-time analysis already generates annotations, usually its user has to improve them manually. Some information that plspec covers, e.g., how predicates are intended to be called, might reduce the manual work required.

Another area is data generation based on a spec. We could use our annotations to generate arbitrary data featuring a certain structure or other properties.

This could be achieved by linking plspec to existing test frameworks for Prolog such as [1]. The authors follow an approach to test case generation and shrinking similar to Erlang’s QuickCheck [6]. However, we would regard test failures as failing predicates if a spec is matched. In consequence, we would not describe actual output values in terms of input values.

Besides, often predicates only transform data into a different structure. With annotations that precisely describe different data structures passed to and returned from a predicate, it might be feasible to both repair incorrect and synthesize new programs solely based on plspec’s annotations.

Finally, plspec could make use of existing annotations, for example mode or meta-predicate annotations. They could be converted directly into our format.

7 Conclusion

In this paper, we presented the library plspec. It provides a DSL that can be used to document Prolog predicates in a way that is straightforward. This DSL is easily extensible without getting involved with internal implementation details and flexible enough to suit the needs of a broad range of Prolog programs. Furthermore, these annotations can be used in order to quickly and effortlessly enable runtime checks if required.

While the performance hit might be too big for recursive predicate, we argue that, firstly, most checks suffice to be made at the entry level because of the recursive implementation of specs for recursive data. Furthermore, invariants are powerful enough to catch incorrect bindings at a deeper recursion level. Secondly, plspec is a tool intended to catch errors during development. Our runtime checks should not be deployed as production code and if so, only very carefully.

References

Closed Types for Logic Programming

João Barbosa, Mário Florido, and Vítor Santos Costa
Faculty of Science of the University of Porto
Portugal

Abstract. Type systems are a powerful tool in modern programming languages. Static descriptive type inference algorithms for logic programs, which do not rely on a priori type declarations, usually abstract the program success set. This makes types over generous in several cases, due to the unconstrained use of logic variables which may cause the acceptance of more terms than intended in a successful computation. We argue that in a fully typed logic programming language we should have type constraints over all variables. In this paper we propose that all uses of logic variables in a program should be type constrained, where by constraints we mean that either the type of the variable is strictly smaller than the set of all possible terms, or that there is an equality constraint between different types. The types in which all variables are constrained are named closed types. Here we define the notion of closed types and a closure operation which transforms general regular types into closed types.

1 Introduction

Types in logic programming have been proposed since the early days of the field [1–4]. The approaches are very diverse, ranging from strict type disciplines based on the notion of well typed programs defined by a type system [5, 6] (often inspired by work in the functional language community), to more permissive frameworks based on type inference of regular types where types are a conservative approximation of the program semantics [1, 2, 8, 10, 11, 14, 17, 19, 20, 31]. More recently, several researchers have challenged the logic programming community by making the point that type systems should be an available tool in logic programming systems [7, 21, 22, 27, 24].

This raises the question of whether untyped logic programming languages such as Prolog should continue to be untyped, an approach that has been successful with languages such as Python and JavaScript, or whether types should be at the heart of logic programming systems.

This work is motivated by this challenge. Our goal is pragmatic: we would like to use types in Prolog to facilitate the task of the programmer. In other words, our ideal type system would be the one that captures most mistakes with least programmer efforts. Several mistakes in logic programs arise from the unconstrained use of the logic variables, which may cause the acceptance of more terms than intended in a successful computation.
We argue that such programs are overly generous and that in a fully typed logic programming language all uses of the logic variables in a program should be type constrained in some way. The types in which all variables are constrained are here named closed types. Consider the following example, where, here and in the rest of the paper, $\tau_p^i(\alpha)$ denote the polymorphic type (with type parameter $\alpha$) of the $i$-th argument of the Prolog predicate $p$ and $+$ denote a disjunction of types in a type definition:

**Example 1.** Consider the well-known Prolog `append` definition:

\[
\text{append}([], X, X).
\]

\[
\text{append}(X|L1, L2, [X|L3]) : - \text{append}(L1, L2, L3).
\]

Then:

\[
\begin{align*}
\tau_1^{\text{append}}(\alpha) &= [\ ] + [\alpha | \tau_1^{\text{append}}], \\
\tau_2^{\text{append}}(\beta, \gamma) &= \beta + \gamma, \\
\tau_3^{\text{append}}(\beta, \alpha) &= \beta + [\alpha | \tau_3^{\text{append}}],
\end{align*}
\]

is a semantically valid type for `append` in the sense that it over-approximates the predicate’s Herbrand model.

This type accepts more terms than the intended interpretation of `append` which is relating three lists as arguments (for example it validates calls such as `append([], 1, 1)`). This problem is known since the early beginning of research on types for logic programming: Zobel said about type inference algorithms “inferred types may have no relation to a predicate’s ‘intended’ types, and are simply cartesian products, of sets of ground terms, that contain all tuples of ground terms that can occur in the predicate’s success set” [14]. Of course one may say that the definition of intended interpretation is not known in advance, but we argue that the intended interpretation of a program is the interpretation any programmer would have when programming the same specification in a typed programming language. This view is implicit in previous work by Naish [15] and explicit in type systems which make it mandatory to explicitly declare type definitions of program functors [5, 6].

Thus, going back to our example, we argue that the type:

\[
\begin{align*}
\tau_1^{\text{append}}(\alpha) &= [\ ] + [\alpha | \tau_1^{\text{append}}(\alpha)], \\
\tau_2^{\text{append}}(\alpha) &= [\ ] + [\alpha | \tau_2^{\text{append}}(\alpha)], \\
\tau_3^{\text{append}}(\alpha) &= [\ ] + [\alpha | \tau_3^{\text{append}}(\alpha)],
\end{align*}
\]

is a (polymorphic) closed type for `append`. Note that this type does not over-approximate the predicate model, but, somehow, it corresponds to the most general accepted intended interpretation of the predicate `append` as a program which appends two lists of elements of type $\alpha$. This is also a well-typing of `append` by the most strict systems [5, 6] based on an Hindley-Milner style of type system definition.
The predicate in the example above is also typed by an over-approximation of the predicate model, requiring the first argument to be a list, but the second argument is not type constrained, whereas closed types, having a more restrict type definition, require the second and third argument to be type constrained to be a list as well.

Our framework relies on two important premises:

- The programmer may not declare types at all, thus we assume that there is a type inference algorithm [8, 9] which automatically infers a type for a given untyped program. Moreover we assume that the inferred types are based on regular types [2, 10, 11].
- Types are polymorphic [2, 3], based on a notion of parametric polymorphism.

The novel contributions of this paper are:

- **Closed types**: we define a new notion of closed type giving a bound to the set of possible types for a predicate argument where those possible types define a datatype.
- **A closure operation**: we present an algorithm to transform types into types which are closer to the programmer’s ‘intended’ definitions [15].

We would like to see this proposal as following in the line of Lee Naish’s “Specification = Program + Types” [15]. In this case, closed types can be seen as a declarative way to complement control, by restricting programs in order to facilitate the development of large applications, while preserving the flavor of logic programming without the need of a priori type declarations.

## 2 Understanding Closed Types

Here, we informally introduce the idea behind closed types through several illustrating examples and we collect constraints on what the definition of closed types should be. Throughout the rest of the paper, we will assume that type variables are represented by $\alpha, \beta, \gamma$, type constructors are represented by $f, g$ and types are represented by $\tau, \sigma$.

Our motivation for closed types stems from experience in both functional and logic programming languages. Functional languages have a similar challenge in variant datatypes (such as data declarations in Haskell), and require a similar restriction. A second motivation stems from deductive databases [13, 30].

Consider the following Datalog program:

$$i(X, Y) :- e(X).$$

This program is not allowed because $Y$ matches any object in the database. Datalog implementations address this problem by explicitly disallowing unconstrained head free variables as they match the full Herbrand base. Since this program is not allowed in Datalog, it should not be considered well typed.
Let us begin by considering the problem with unconstrained occurrences of free variables. A free variable in a Prolog predicate may be instantiated with any term. We argue that those variables should be type constrained by a specific type, and thus we have this first principle:

**Principle 1.** *Types must be strictly smaller than the set of all possible terms.*

*Example 2.* Let us consider the following definition for predicate \( p \):

\[ p(1, X). \]

The type we get for the second argument is \( \tau^2_p = \alpha \), where \( \alpha \) is a type variable that appears only once in the definition of the predicate and as such, it will appear only once in the set of types for the predicate. This makes the type for the second argument open.

Even if we are willing to accept this principle, however, there are still some difficulties. We still want to allow predicates which relate uniformly elements of any type, when those types are related. This is essential to define parametric polymorphic predicates. The key principle here is:

**Principle 2.** *All uses of the logic variables in a program should be type constrained.*

*Example 3.* Consider the predicate:

\[ \text{identity}(X, X). \]

The type of both arguments is the same and equal to a type variable: \( \tau^1_{\text{identity}} = \tau^2_{\text{identity}} = \alpha \). Note that \( X \) occurs twice in the predicate definition, thus there is a constraint on the type, so it should be considered closed with respect to \( \{ \tau^1_{\text{identity}} = \alpha, \tau^2_{\text{identity}} = \alpha \} \).

These two principles will give rise to the definition of *unconstrained type variable* which is going to be formalized later in the paper, but which basically identifies dangerous uses of type variables which do not filter the set of possible substitutions applied to program variables.

Another consequence of our first principle is that types made from a disjunction of several types should be closed, in the sense that they must define a datatype with a finite number of choices. This is exactly what happens in datatype definitions in most programming languages, for example data definitions of Haskell.

This notion of closed datatype definitions with a finite number of choices will give rise to the definition of *closed composite types* (which also will be formalized later) which corresponds to sums of types where none of its components is an unconstrained type variable.

These two principles are in the basis of the notion of closed types. But in this paper we also define an algorithm which closes (open) types. Several types may
by chosen to replace unconstrained type variables in order to close the type. Thus we must make a choice about how to close types. The next principle motivates our choice. We see the set of symbols used in the program as a closed alphabet to build closed types. Thus, given that the program is made from an alphabet consisting of a set of constants $C$ and a set of function symbols $F$ occurring in a predicate $p$, a type $\tau$ must either have a principal functor in $F$, a built-in type generalizing $C$, or be a disjunction of the previous types, that is, it should be self-contained with respect to the alphabet used in the predicate definitions. To summarize:

**Principle 3.** *Types are based on self-contained definitions.*

We will define later in the paper a *closure operation* which uses only the symbols defined in the predicate definition to build closed types for that predicate. Let us show an example of this principle:

**Example 4.** Consider the predicate $\text{add}$ in Peano arithmetic with the atom $z$ standing for zero:

$$\begin{align*}
\text{add}(z, X, X).
\text{add}(s(X), Y, s(Z)) : \neg\text{add}(X, Y, Z).
\end{align*}$$

Types which over-approximate the predicate model are:

$$\begin{align*}
\tau^1_{\text{add}} &= \text{atom} + s(\tau^1_{\text{add}}) \\
\tau^2_{\text{add}} &= \alpha \\
\tau^3_{\text{add}} &= \alpha + s(\tau^3_{\text{add}})
\end{align*}$$

Note that the type of the second and third arguments denotes a datatype with arbitrary constructors (an open datatype), because one of its elements is a type variable which may be instantiated by any type. The closure operation for this predicate will result in:

$$\begin{align*}
\tau^1_{\text{add}} &= \text{atom} + s(\tau^1_{\text{add}}) \\
\tau^2_{\text{add}} &= \text{atom} + s(\tau^2_{\text{add}}) \\
\tau^3_{\text{add}} &= \text{atom} + s(\tau^3_{\text{add}})
\end{align*}$$

Note that this last type could be a type declaration in type systems with explicitly type declarations for program functors [5, 6] as a way to guarantee type correctness according to the programmer’s intention. Using the assumptions about what is not a too-generous program, our closure operation gives an approximation to the programmer’s ‘intended’ types for a predicate which does not rely on type declarations, but it is entirely built from the program syntax. If the programmer explicitly defines its intention by declaring a type, this information can be used, but it is optional.
3 Regular Types

In this section, we briefly present some notions about regular types. Regular types have been widely used in the definition of type languages for logic programming [2, 10, 11, 14, 17, 19]. Regular types are types that can be described with a regular term grammar [12], and, informally, can be:

- type variables ($\alpha, \beta, \ldots$)
- type symbols (defined in a set of type rules) ($\tau, \sigma, \ldots$)
- type constants (1, $a$, $\ldots$)
- function symbols with arity $n$, applied to an $n$-tuple of regular types
- disjunctions of types (here denoted by the symbol $+$)

We call \textit{summands} to members of a disjunction of types. It is assumed that every type constant can be typed by a built-in type, for instance, 1 can be typed with the built-in type \texttt{int}. There is a set $T$ of type rules, each of them defining a type symbol. We can associate a logic program $\Phi_T$ to the type rules defined in $T$. For example, the logic program corresponding to the type $\tau = [\ ] + [a|\tau]$ is:

$\text{t}([\])$.
$\text{t}(X|R) :- a(X), \text{t}(R)$.

The interpretation of a type symbol according to $T$, $[\tau]_T$ is the set of terms occurring as arguments of the unary predicate $\tau$ in the least model $M_{\Phi_T}$ of the program $\Phi_T$.

Regular types correspond to tree automata [12], which is a class of languages where intersection, subset and unification are decidable.

Example 5. Let $\tau$ be a type symbol. The following is a type rule defining a list of integers with the list constructor ‘.’: $\tau = [\ ] + (\texttt{int}, \tau)$

Thus we will assume an infinite set of type variables ($\alpha, \beta, \gamma, \ldots$) and an infinite set of type constructors for each arity, represented by $f$ in the grammar below. A predicate type (\textit{pred type}) is a tuple of several term types. We shall use $\tau$, $\sigma$ or $\tau_p^i$ for term types (\textit{term types}), where $\tau_p^i$ will be used to represent the type for the $i$-th argument of predicate $p$. Term types may be monomorphic (\textit{mono type}) or polymorphic (\textit{poly type}) of the form $\tau(\alpha)$, where $\alpha$ are parameters (\textit{type var}). However, whenever it is indifferent whether the type is polymorphic or not, we will omit the variables for the sake of readability. We will assume that every constant in a logic program can be typed by a built-in type (\textit{built_in}). These built-in types are \texttt{nil}, \texttt{int}, \texttt{float}, \texttt{num}, \texttt{char}, \texttt{string}, and \texttt{atom}. The syntax of types is given by the following BNF grammar:

\[
\text{type} ::= \text{pred type} | \text{term type} | \text{group type} \\
\text{pred type} ::= \text{term type} \times \ldots \times \text{term type} \\
\text{term type} ::= \text{basic type} | \text{term type} + \text{term type}
\]
In this paper we assume that there is a previously defined type inference algorithm which infers regular types. Several type inference algorithms for logic programming were defined before (see for instance [2, 8, 11, 17, 19, 20, 3]). The properties of the type inference algorithm necessary to apply the closure operation are:

– returning regular types: this is necessary because our closure is defined for regular types;
– types returned by the algorithm must be sound with respect to a semantic definition: this property is important to guarantee that closed types are built from types which are semantically sound.

In this work, we are using the polymorphic type inference algorithm presented in [2, 16] as the previous step of the closure operation. Zobel’s type inference algorithm, as many type inference algorithms for logic programming, computes a conservative approximation of the program semantics. Very briefly, the algorithm follows bottom-up computing types for each clause in a predicate definition, and using a regular type unification algorithm [14] to build new types for terms which unify in the program. After this step, types inferred for each clause in a predicate definition become the parts of a new sum type for the predicate. This module of the algorithm is then repeated until a fixpoint is reached. Note that types inferred at the end of this type inference step may have no relation at all to a predicate’s intended types. This is the role of our closure module: to transform the inferred types into types which are closer to the intended types for each predicate.

4 Closed Types

Closed types are, intuitively, types that are constrained in some way. If a type is open, then it may type any term constructed with any type constructor, including type constructors not present in the program defining that predicate. This is exactly what we want to avoid with closed types. There are two main definitions that motivate closed types: unconstrained type variables and closed composite types. We will hereon name composite types to disjunction of types that have more than one summand and summands are basic types.

Definition 4 (Unconstrained Type Variable). A type variable \( \alpha \) is unconstrained with respect to a set of type definitions \( T \), notation unconstrained(\( \alpha, T \)), if and only if it occurs exactly once in \( T \).

Definition 5 (Closed Composite Type). A composite type \( \tau \) is a closed composite type, notation closedComposite(\( \tau \)), if and only if it has no type variables as summands.
The definition for closed types uses these two previous auxiliary definitions.

**Definition 6 (Closed Types).** A type $\tau$ is closed with respect to a set of type definitions $T$, notation $\text{closed}(\tau, T)$, if and only if the predicate defined as follows holds:

$$\text{closed}(\tau, T) = \begin{cases} 
\text{closedComposite}(\tau) & \text{if } \tau \text{ is a composite type} \\
\neg\text{unconstrained}(\tau, T) & \text{if } \tau \text{ is a type variable} \\
\text{True} & \text{if } \tau \text{ is basic but not a type variable}
\end{cases}$$

Informally, a closed type is either a closed composite type or, if it is a basic type, then that basic type cannot be an unconstrained type variable. The first case of this definition avoids open data types while the second case avoids unconstrained type variables which could be instantiated by the whole Herbrand universe.

**Example 6.** $\tau_1 = \alpha + f(\beta)$ is not a closed type with respect to any set of type definitions $T$, since it does not respect the first case of our definition.

$\tau_2 = \text{int} + f(\alpha)$ is a closed type with respect to any set $T$, since it does not have variables as summands.

## 5 Closure Operation

Assuming that there is a previous type inference algorithm which infers regular (open) types for a given logic program, we now define a closure operation which, given a set of regular types, closes them. This operation follows Principle 3 i.e. types are based on self-contained definitions.

We first define the *proper type domain* of a type $\tau$ with respect to a set of type definitions $T$ as the set of non-variable summands in $\tau$ itself and in the type definitions of all the types that share at least one type constructor with $\tau$.

A precise definition of the proper type domain of a type $\tau$ with respect to set of type definitions $T$, notation $\text{properType}(\tau, T)$, is that it is the set of types computed by the following function in pseudocode:

```plaintext
function properType(\tau, T)
    p = the set of non-variable summands of \tau;
    for each constructor c \in \tau do
        for \tau' \in T do
            if c \in \tau' then
                s = set of non-variable summands of \tau';
                p = p \cup s;
            end
        end
    end
    return p
end function
```
We then define the *proper variable domain* of a type variable $\alpha$ with respect to a set of type definitions $T$ as the union of all proper domains for the types whose definition includes $\alpha$ as a summand.

A precise definition of the proper variable domain of a type variable $\alpha$ with respect to a set of type definitions $T$, notation $\text{properVar}(\alpha, T)$, is that it is the set of types computed by the following function:

```plaintext
function properVar($\alpha, T$)
  $v = \emptyset$;
  for each $\tau \in T$ do
    if $\alpha$ is a summand of $\tau$ then
      $p = \text{properType}(\tau, T)$;
      $v = v \cup p$;
    end
  end
  return $v$
end function
```

Note that the definitions for the proper type domain and the proper variable domain are based on principles 2 and 3, since they are self-contained in the sense that the information used is defined in the lexical components of the program and closure will forbid unconstrained type variables.

Using these algorithms we can now define a closure operation given a set of type definitions $T$, notation $\text{closure}(T)$, as the set of types computed by the following function:

```plaintext
function closure($T$)
  $c = \emptyset$;
  for each $\tau \in T$ do
    if $\tau$ is not closed then
      for each $\alpha \in$ set of variables which are summands of $\tau$ do
        $p = \text{properVar}(\alpha, T)$;
        $s = \text{makeSum}(p)$;
        if $p \neq \emptyset \land \tau \neq \alpha$ then
          $c = c \cup \{\tau/\alpha\}$;
        end
      end
    else
      $c = c \cup \{\tau\}$;
    end
  end
  return $c$
end function
```

The auxiliary function $\text{makeSum}$ transforms a set of basic types into the sum of those basic types. Informally, the algorithm to compute the closure of a set of type definitions $T$ consists of the following steps:
– get all open types \( \tau \in T \) with respect to \( T \setminus \{ \tau \} \);
– for each variable \( \alpha \) that makes these types open, get their proper variable domain;
– substitute every occurrence of those variables as summands in the definitions in \( T \) by their proper variable domain;

Note that polymorphic occurrences of the variables, i.e. inside a type constructor, will not be substituted, since these variables are constrained by the constructor itself.

**Proposition 7.** Given a set of type definitions \( T \) the types defined in its closure, \( \text{closure}(T) \), are closed with respect to \( \text{closure}(T) \).

This stems from the fact that every variable is substituted by a new sum of types and this sum never contains a variable, therefore the resulting definition only contains variables inside type constructors, which means that the resulting types are closed.

If the proper variable domain of some type variable in a given set is empty, this indicates that we have no information to use for the closure. In these cases the variable is just “ignored” and the type is closed with itself because there is no hint on its intended type given by symbols used in the program.

**Example 7.** Let the predicate \( \text{gcd} \) calculate the greatest common divisor between two integers and be defined as follows:

\[
gcd(X, 0, X).
gcd(X, J, K) :- R \text{ is (} X \text{ mod } J), \ gcd(J, R, K).
\]

If we assume that the type inference algorithm attributes \( \text{int} \) to all variables involved in a “is” goal, then the open types obtained for this predicate are:

\[
\begin{align*}
\tau_{\text{gcd}}^1 &= \alpha + \text{int} \\
\tau_{\text{gcd}}^2 &= \text{int} \\
\tau_{\text{gcd}}^3 &= \alpha + \text{int}
\end{align*}
\]

Applying the closure to the set \( T = \{ \tau_{\text{gcd}}^1 = \alpha + \text{int}, \tau_{\text{gcd}}^2 = \text{int}, \tau_{\text{gcd}}^3 = \alpha + \text{int} \} \), we get the type \( \text{int} \times \text{int} \times \text{int} \) for the whole predicate. Following execution step-by-step:

– types \( \tau_{\text{gcd}}^1 \) and \( \tau_{\text{gcd}}^3 \) will be detected as open;
– the proper variable domain of \( \alpha \) will be calculated;
– the proper type domain of both \( \tau_{\text{gcd}}^1 \) and \( \tau_{\text{gcd}}^3 \) will be calculated;
– the result for the proper variable domain will be obtained (\( \text{int} \));
– after substituting the variable for the proper domain, the resulting set of type definitions will be \( T = \{ \tau_{\text{gcd}}^1 = \text{int}, \tau_{\text{gcd}}^2 = \text{int}, \tau_{\text{gcd}}^3 = \text{int} \} \).

A final note about closed types has to do with their relation to untyped programs. Note that closed types filter the set of admissible queries to a program as
the ones which are typed by the closed type. This has an immediate consequence which is that we may have a ground query for which the answer in an untyped version is “yes” and the answer in the same program but typed by a closed type is “no”. The example on the paper Introduction for the append predicate shows this: the query \texttt{append([],a,a)} has answer “yes” in the untyped version of append and “no” in the version typed by the closed type in the example. The opposite does not happen, i.e., if the answer to a query to a predicate typed by a closed type is “yes”, then the answer to the same query to the untyped version of the predicate still is “yes”. This holds because closed types are instances of an (open) type for the predicate which we assume it over-approximates the program semantics.

5.1 Closed Types and Program Debugging

Closed types, being less premissive that open regular types, may catch more program bugs. This is a pragmatic motivation of the use of closed types. Let us now give an illustrating example.

\textit{Example 8.} Let \texttt{max} be a predicate that finds the largest number of a list.

\begin{verbatim}
max([], Max, Max).
max([H|L], Max0, Max) :- Max0 < H, max(L, H, Max).
max([H|L], Max0, Max) :- max(L, Max0, Max).
\end{verbatim}

Sound open types for the predicate can be:

\begin{align*}
\tau^1_{\text{max}} &= \text{[ ]} + [\tau | \tau^1_{\text{max}}] \\
\tau^2_{\text{max}} &= \alpha + \text{num} + \beta \\
\tau^3_{\text{max}} &= \alpha + \gamma + \eta
\end{align*}

where, \( \tau = \text{num} + \delta \). It is clear to see that types \( \tau^2_{\text{max}} \) and \( \tau^3_{\text{max}} \) are open. The closed types returned by our closure step are the following:

\begin{align*}
\tau^1_{\text{max}} &= \text{[ ]} + [\text{num} | \tau^1_{\text{max}}] \\
\tau^2_{\text{max}} &= \text{num} \\
\tau^3_{\text{max}} &= \text{num}
\end{align*}

One common bug when defining this predicate can be defining the first clause as:

\begin{verbatim}
max([], Max, M).
\end{verbatim}

Now, closure fails in this case, since all the variables on \( \tau^3_{\text{max}} \) have an empty proper variable domain, and therefore that type becomes empty and thus the closure fails.

This is an example of how close types, being less permissive, avoid the debugging process at runtime, by catching more bugs at compile time.
5.2 Datatype-centric Programming

Programs in functional programming languages (such as Haskell and ML) and
in imperative and object-oriented languages (such as C and Java) are often
datatype-centric, in the sense that they are based on and make an intensive
use of algebraic datatypes. Usually it is considered that the same happens in
Prolog, using terms as a notation for datatype definitions. We argue that this
is not always the case due to the use of unconstrained logical variables as parts
of those datatype definitions. In these pathological cases the types we would get
for programs are open types. Closed types have an important role on the use
of a truly datatype-centric style of programming in logic programming. Let us
consider the following example:

Example 9. Let flatten be the standard Prolog predicate whose first argument
is a nested list of lists and the second is the flat version of that nested list, defined
as follows:

\[
\begin{align*}
\text{flatten}([], []). \\
\text{flatten}([L|R], Flat) :&- \\
& \text{flatten}(L, F1), \text{flatten}(R, F2), \text{append}(F1, F2, Flat). \\
\text{flatten}(L, [L]).
\end{align*}
\]

Open types for the predicate are:

\[
\begin{align*}
\tau^1_{\text{flatten}} &= [\tau^1_{\text{flatten}} | [\tau^1_{\text{flatten}}]] + [[] + \alpha] \\
\tau^2_{\text{flatten}} &= \tau^3_{\text{append}} + [[] + [\alpha]]
\end{align*}
\]

Note that the type for the first argument is open. The problem here is that
in the implicit datatype definition of the first argument of the predicate, single
elements of lists (here processed by the third clause of the predicate definition)
do not have an associated constructor which distinguishes them from any other
terms, such as lists.

This problem can be solved by changing the predicate definition as follows:

\[
\begin{align*}
\text{flatten}([], []). \\
\text{flatten}([L|R], Flat) :&- \\
& \text{flatten}(L, F1), \text{flatten}(R, F2), \text{append}(F1, F2, Flat). \\
\text{flatten}(\text{elem}(L), [\text{elem}(L)]).
\end{align*}
\]

Now, the predicate has the following closed types:

\[
\begin{align*}
\tau^1_{\text{flatten}} &= [\tau^1_{\text{flatten}} | [\tau^1_{\text{flatten}}]] + [[] + \text{elem}(\alpha)] \\
\tau^2_{\text{flatten}} &= \tau^3_{\text{append}} + [[] + [\text{elem}(\alpha)]]
\end{align*}
\]

Note that the \text{elem} functor in this predicate definition is playing the role of
a type constructor which identifies single elements in a datatype definition for
nested lists (such as in a data declaration in Haskell). This is enough to make
the implicit type definition of the first argument closed.
We are not advocating this second style of programming in Prolog, but we argue that if one wants to have safer programs, in the sense that bugs and errors are easier to catch at compile time, either we declare types, such as in Curry or Mercury, or we infer types with an extra closure operation after type inference which will make those types truly denote datatype definitions. If this is the case, then a more datatype-centric style of programming, with functors as type constructors in every case of an implicit datatype definition will avoid the extra closure operation in many cases, making the initial inferred types already closed.

6 Related Work

Since the early works on the area [10, 17, 20, 14, 1, 3], type systems for logic programming describe types as conservative approximations of the program semantics. Often, these approaches are based on the notion of regular types. Regular types can be written as logic programs, namely monadic logic programs [17], using unary-predicate programs to describe types in a natural way. Regular types returned by a type inference algorithm described in [2] are the input of our closure function, and as we have show in the paper they are, several times open.

Both type verification and type inference algorithms have been proposed before [2, 10, 20, 19, 22, 11]. These approaches differ on whether types are considered approximations of the success set of a logic program, or whether one wants to ensure that a type signature will be respected.

Mycroft and O'Keefe formulated a type system for Logic Programming [5], which Lakshman and Reddy later called Typed Prolog [6]. We can see some similarities between this type system and the one defined in [18]. In these systems, types of function symbols in the program are declared by the programmer. In some cases there are algorithms that reconstruct the type of the predicate having type declarations for function symbols as input. These systems are related to our notion of closed types because type declarations given by the programmer are, in many cases, functional types from closed types to closed types: they correspond to functions with datatypes as input.

Type systems have not been widely adopted by Prolog systems. Arguably, Ciao has been the major exception, through the support of parametric types in its declaration system [25]. Ciao also includes libraries for regular and Hindley-Milner types. More recently, there has been a revival of interest in Hindley-Milner types for mainstream Prolog systems, such as SWI-Prolog and YAP, where a new module was introduced for typechecking that allows for the mixture of typed and untyped code, with type declarations and run-time type checking [7]. A similar approach for XSB was proposed in Hadjichristodoulou’s gradual discovery of Hindley-Milner types [27], where programs need not a type declaration, but gradually become typed with similar types to the ones used recently in SWI-Prolog and YAP [29]. Our system can be easily added to these type system as a post-processing unit which checks if the types inferred or declared are closed and closes them if they are not.
Most of the work on type systems for programming languages has been influenced by the traditional definition of type systems for $\lambda$-calculus and functional programming [18, 28]. Both these type system and its evolution in modern functional languages, such as Haskell, have also been very influential in the design of type systems for functional-logic programming, such as Curry [23] and Mercury [26]. Types in these languages are basic types, datatypes or functional types between basic types and datatype definitions. Datatype definitions are usually closed by construction, i.e., they are declared by the programmer and cannot be open in the sense presented in this paper. These languages relate to closed types in two ways. Firstly, closed types can be easily translated to these datatype declarations and similar data definitions in typed functional programming languages, such as Haskell. Secondly, our closure algorithm can be integrated in a compilation module from an untyped logic programming language, such as Prolog, to a typed logic programming language where type information is automatically added by a type inference module followed by our closure operation. We did not explore this line of research but there is a clear relation with typed logic programming languages here that we want to explore in the future.

7 Final Remarks

Our definition of closed types corresponds to what we understand by not too over-generous programs and the results we got from the tests performed on several programs are what we intended. One of the advantages of closed types is that types become closer to the programmer’s intention (or the usual data structure definitions in a statically typed language) than open types.

We have implemented in Prolog a prototype of a type inference algorithm based on Zobel type inference for polymorphic regular types [2]. After the type inference stage we then have an implementation of the closure algorithm which closes types, and rejects programs whose types cannot be closed. Results for the tests we performed so far show that closed types matched the intended types most of the times. This strongly suggests that closed types retain the power of type declarations without having to declare types. Thus we argue that they are a first step towards having the best of both worlds: the type discipline of languages where the programmer has to declare types and the programming flexibility of untyped languages. We also argue that closed types are easy to understand and although they constrain the terms accepted by the program, we think that they are not too restrictive. We are now in the process of adding a type inference module to YAP [29] based on closed types.

Acknowledgments This work is partially funded by FCT within project Eleven POCI-01-0145-FEDER-016844, Project 9471 - Reforcar a Investigacao, o Desenvolvimento Tecnologico e a Inovacao (Project 9471-RIDTI), by project PTDC/EEI-CTP/3506/2014, and by Fundo Comunitario Europeu FEDER.
References


10. Eyal Yardeni, Thom W. Frühwirth, Ehud Y. Shapiro; Polymorphically Typed Logic Programs; Types in Logic Programming 63–90 (1992)


14. Philip W. Dart, Justin Zobel; A Regular Type Language for Logic Programs; Types in Logic Programming 157–187 (1992)

15. Lee Naish; Types and the Intended Meaning of Logic Programs; Types in Logic Programming 189–216 (1992)


17. Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, Eyal Yardeni; Logic Programs as Types for Logic Programs; Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS ’91) 300–309, Amsterdam, The Netherlands, July 15-18 (1991)
19. Nevin Heintze, Joxan Jaffar; Semantic Types for Logic Programs; Types in Logic Programming 141–155 (1992)
23. Michael Hanus; Functional Logic Programming: From Theory to Curry; Programming Logics - Essays in Memory of Harald Ganzinger 123–168 (2013)
28. Luís Damas, Robin Milner; Principal Type-Schemes for Functional Programs; Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages 207–212 (1982)
31. Mário Florido and Luís Damas; Types as Theories; Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming (1992)
Controlling LEGO® EV3 robots with Prolog (System description)

Sibylle Schwarz and Mario Wenzel

HTWK Leipzig, F-IMN, Postfach 301166, 04251 Leipzig
{sibylle.schwarz,mario.wenzel}@htwk-leipzig.de

Abstract. We present a method to control LEGO EV3 robots by Prolog programs. The connection between the robot and SWI-Prolog is established via ev3dev. The sensors and actors of the robot can be operated from Prolog programs using our collection of predefined predicates. We demonstrate our approach by some examples from our introductory robotics courses.

Keywords: logic programming, SWI-Prolog, robotics, LEGO EV3, ev3dev

1 Motivation

Intelligent control of autonomous robots has been an important goal of AI research from its very beginnig. Applications in mobile robotics are frequently used to motivate research and lectures on AI, planning and logic programming [7]. However, in many AI courses this motivation remains theoretical.

On the other hand, introductory robotics courses are evidently successful in growing young peoples interest in STEM topics like computer science, construction, physics and mechanics. LEGO® MINDSTORMS® is a flexible robotics platform that allows easy construction and programming of various robots for first steps in robotics with standard LEGO® pieces. These robots are frequently used in introductory courses to robotics for children and young students. Since it also allows advanced contructions with complex behaviour, it is also used in university courses and in research.

LEGO® provides a visual programming environment for controlling EV3 robots. Therefore, first programs for these robots are frequently written in this visual language. For several reasons, we even use the visual programming interface in robotics projects and competitions for our first year students. Instructions for many basic experiments in robotics like obstacle avoidance, line following, self-balancing, are avaliable online and in the literature.

The next step is usually towards imperative languages, like Java or a special C-like language NXC provided by LEGO®. Over the years, several platform-independent and open source APIs for imperative languages evolved.

Unfortunately, bindings to logic programming are rare. There have been successful approaches to connect Prolog and AI to its predecessors RCX [5] and NXT [6].
In this paper, we further develop this idea to a connection between SWI-Prolog and the EV3 hardware. We use the alternative operation system ev3dev, that conveniently wraps all communication details between actors, sensors, and the CPU. With the presented method, we provide a method to connect Prolog-based AI with LEGO® EV3 robots, both at a level accessible to non-experts in robotics and Prolog. In Section 3, we demonstrate our approach by some examples from our introductory robotics courses for students and children.

2 Prolog-API for LEGO®EV3

EV3 [9] is the most recent LEGO®MINDSTORMS® robotics platform. It consists of a programmable brick, motors and sensors that can be combined by standard LEGO® pieces to various robots.

Unlike its predecessors, EV3 has a powerful ARM9 CPU processor running Debian Linux. Hence advanced programs like a Prolog interpreter can be installed and run directly on the brick.

For LEGO®EV3 robots, the ev3dev framework [8] provides a convenient way to address sensors and actors via the file system. ev3dev is an open source OS for LEGO®MINDSTORMS®EV3 based on Debian Linux. It provides a low-level driver framework for controlling EV3 brick devices, sensors, and motors. Usually, ev3dev is run on the EV3 brick by booting from an SD card containing the ev3dev image.

In this paper, we describe how ev3dev can be used to control EV3 robots by Prolog programs. We limit ourselves to wrapping some of the ev3dev accessors into a Prolog library and currently do not work with an internal robot model that would allow for higher level abstractions.

2.1 Robot Configuration

We define the properties of our robot, e.g. which motors and sensors are attached to which port, as ground truths. This is not part of the library and is set by the user. This is also the configuration we use for all Braitenberg vehicles (see Section 3).

ev3_large_motor(portB).  ev3_large_motor(portC).
light_sensor(port2).     light_sensor(port3).
For all supported sensors and motors we give predicates that always fail. This is necessary because the user might not provide those predicates as ground truths and there is no mechanism yet for auto-detection of those peripherals.

\[
\begin{align*}
ev3\text{-large}\text{_motor}(_) & : = \text{false}. & ev3\text{-medium}\text{_motor}(_) & : = \text{false}. \\
nxt\text{_motor}(_) & : = \text{false}. & light\text{_sensor}(_) & : = \text{false}. \\
ultrasonic\text{_sensor}(_) & : = \text{false}.
\end{align*}
\]

Since in ev3dev communication with the sensors and motors of the robot is done via device files, their configuration consists of the composition of pathnames to the location of the corresponding files. If a large motor is attached to port B we find the files corresponding to the motor state in a directory \\
/sys/class/tacho-motor/motor<N> where N increases whenever a new motor is attached. To control the motor at port B we would be looking for the directory of that pattern whose `address` file has `outB` as its content.

We define a mapping of possible `address` values for internal use in addressing sensors or actors, and constants that we plan to expose the user.

\[
\begin{align*}
\text{port}\_\text{symbol}(\text{portA}, \text{'}outA\text{'}). & \\text{port}\_\text{symbol}(\text{portB}, \text{'}outB\text{'}). \\
\text{port}\_\text{symbol}(\text{portC}, \text{'}outC\text{'}). & \\text{port}\_\text{symbol}(\text{portD}, \text{'}outD\text{'}). \\
\text{port}\_\text{symbol}(\text{port1}, \text{'}in1\text{'}). & \\text{port}\_\text{symbol}(\text{port2}, \text{'}in2\text{'}). \\
\text{port}\_\text{symbol}(\text{port3}, \text{'}in3\text{'}). & \\text{port}\_\text{symbol}(\text{port4}, \text{'}in4\text{'}). \\
\end{align*}
\]

We find the corresponding directory using the `device\_path` predicate that gives the directory for the attached sensor or actor, depending which subsystem ev3dev uses to interact with them, where the interface files are located.

\[
\begin{align*}
tacho\_\text{motor}(M) & : = \\text{ev3}\_\text{large}\text{_motor}(M); \\text{ev3}\_\text{medium}\text{_motor}(M); \\text{nxt}\_\text{motor}(M). \\
\text{uart}\_\text{host}(\text{Port}) & : = \\text{light}\_\text{sensor}(\text{Port}); \\text{ultrasonic}\_\text{sensor}(\text{Port}). \\
\text{device}\_\text{path}(\text{Port}, \text{DevicePath}) & : = \\text{port}\_\text{symbol}(\text{Port}, \text{Symbol}), \\text{((tacho}\_\text{motor}(\text{Port}), \text{expand}(_{/}/\text{sys/class/tacho-motor/motor*/address'}, \text{AddressFile}) ) ; \\text{(uart}\_\text{host}(\text{Port}), \text{expand}(_{/}/\text{sys/class/lego-sensor/sensor*/address'}, \text{AddressFile}) )), \\text{file}\_\text{read}(\text{AddressFile}, \text{Content}), \text{Content} = \text{Symbol}, !, \\text{file}\_\text{directory}\_\text{name}(\text{AddressFile}, \text{DevicePath}).
\end{align*}
\]

2.2 Actor Control

ev3dev provides a number of virtual files that can be read and/or written to in order to control or read the actor or sensor. We give part of a directory listing to illustrate this:
To run a motor we need to write `run-forever` in its `command` file. The alternative commands would be `run-to-abs-pos`, `run-to-rel-pos`, `run-timed`, `stop`, and `reset`. Which commands are actually supported by the motor can be read from the `commands` file, that we do not parse yet. Then we need to write a target speed into the `speed_sp` file. Given a second implementation the target speed can also be read.

Since we currently only support the `run-forever` command, we set it to the corresponding value when we set the target speed `speed_sp`.

```prolog
speed_sp_file(Port, File) :-
    tacho_motor(Port), device_path(Port, Basepath),
    atomic_concat(Basepath, '/speed_sp', File).

command_file(Port, File) :-
    tacho_motor(Port), device_path(Port, Basepath),
    atomic_concat(Basepath, '/command', File).

command(M, C) :- command_file(M, F), file_write(F, C).

speed_sp(MotorPort, Speed) :- % evokes the action
    integer(Speed),
    ( tacho_motor(MotorPort),
      max_speed(MotorPort, MaxSpeed),!,
      MaxSpeed >= Speed, Speed >= -MaxSpeed,
      speed_sp_file(MotorPort, F), file_write(F, Speed),
      if(Speed == 0, command(MotorPort, 'stop'),
          command(MotorPort, 'run-forever'))
   ).

speed_sp(MotorPort, Speed) :- % reads the target speed
    var(Speed), ( tacho_motor(MotorPort),
      filename_motor_speed_sp(MotorPort, F), file_read(F, Speed)).
```
2.3 Reading and Writing Motor State

For the most part the predicates implemented so far correspond to some interface file in ev3dev. Those files can be either writable or readable or both. Given Prolog’s syntax we can use the same predicates for both operations.

\[
\text{equalize_motors} :- \text{speed} \text{sp} \text{(portB, Speed), speed} \text{sp} \text{(portC, Speed)}.
\]

Here the first call to \text{speed} \text{sp} is with an unbound variable \text{Speed} that cannot be used to write to the motor at port B so instead it is read. For the second call to \text{speed} \text{sp} the variable is bound and its content is written to \text{speed} \text{sp} of the motor at port C. Since reading and writing the motor speeds are side effects of the evaluation scheme, the order of evaluation is important and in this case apparent.

2.4 Sensor Modes and Values

The sensors are read in a similar fashion. The sensors supported by the sensor subsystem are located under /sys/class/lego-sensor/sensor<N>. Most sensors have different modes of operation. The light sensor can both detect ambient light as well as the amount of reflected light from the builtin LED. The available modes are in the \text{modes} file (that we currently do not evaluate) and the chosen mode needs to be written to the \text{mode} file. Then the value or values, depending on the mode, can be read from the \text{value0} to \text{value7} files. The read value is an integer and possibly needs to be adjusted by powers of ten depending on the value in the \text{decimals} file, which we also do not evaluate yet.

\[
\text{mode}(M, C) :- \text{mode} \text{file}(M, F), \text{file} \text{write}(F, C).
\]
\[
\text{value}(\text{Port}, \text{ValueNum}, \text{Value}) :-
\quad \text{value} \text{file}(\text{Port}, \text{ValueNum}, \text{File}), \text{file} \text{read}(\text{File}, \text{Value}).
\]
\[
\text{col_ambient}(\text{Port}, \text{Val}) :- \text{light} \text{sensor}(\text{Port}),
\quad \text{mode}(\text{Port}, \text{'COL-AMBIENT'}), \text{value}(\text{Port}, 0, \text{Val}).
\]

3 Examples

We describe some robotics experiments from our introductory robotics lectures and show how to implement robot configuration and control using our predicates.

3.1 Braitenberg Vehicles

The very first robots we use in our introductory robotics courses are two-wheeled Braitenberg vehicles (see [1], where all figures are from).
In Braitenberg vehicles, the connection between cognition and action is simple and direct. Every movement of the vehicle is an immediate reaction to the output value of a sensor. There is no intelligence involved. The values of (light or ultrasound) sensors directly determine the speed of the wheel motors. Because this simple construction can generate surprisingly complex behaviour, experiments with Braitenberg vehicles are suitable to draw interest in robotics.

**Braitenberg vehicle 1** More light produces faster movement. Less light produces slower movement. Darkness produces standstill. We only use the intensity given by one light sensor and apply it to both motors.

```prolog
braitenberg1a :- col_ambient(port2, Light),
   speed_sp(portB, Light), speed_sp(portC, Light),
   braitenberg1a.
```

**Braitenberg vehicle 2** The more light the faster the vehicle. The left light sensor corresponds to the left motor’s rotation and the right light sensor corresponds to the right motor’s rotation. This leads to the vehicle turning away from the light because the motor that is on the same side as the light is faster than the other one.

```prolog
braitenberg2a :- col_ambient(port2, Light),
   speed_sp(portB, Light), speed_sp(portC, Light),
   braitenberg2a.
```
braitenberg2 :-
  col_ambient(port2, LightR), speed_sp(portB, LightR),
  col_ambient(port3, LightL), speed_sp(portC, LightL),
  braitenberg2.

By exchanging the ports for either the sensors or the motors the vehicle
changes behaviour by turning towards the light since the motor on the darker
side rotates faster than the one on the light side. If the student is given a moving
light source, this is an especially funny behaviour since the student can be chased
by the robot or lead the robot around the room using the light source.

3.2 Obstacle Avoidance

Obstacle Avoidance is an introductory experiment to demonstrate intelligent
reactions of a robot to its environment. The robot detects obstacles using a for-
ward facing distance sensor attached to the robot. The robot should go forward
and once it senses an obstacle within a specified distance, turn until the sensor
shows that there is no longer an obstacle within that distance.

In [6], the author presents code for a similar experiment with LEGO®NXT
robots. Compared to that approach, our code is more compact because it does
not depend on any specific robot model.

ultrasonic_sensor(port1).
obstacle_avoidance :-
  ((us_dist_cm(port1, Dist), Dist > 50,
    speed_sp(portB, 50), speed_sp(portC, 50)
  ); speed_sp(portB, 0)), obstacle_avoidance.

Through Prolog’s evaluation scheme we can also deduce the port of the ultra-
sonic sensor given the configuration using the predicate ultrasonic_sensor(Port)
binding the port of some attached sensor to the variable Port.

4 Discussion

The original contribution of this paper is a Prolog binding for LEGO®EV3
robots. We demonstrate its application by several robotics experiments for be-
ginners.

The presented applications do not yet involve much AI but the reactive be-
avour of the Prolog programs is interesting in itself. Reading sensor values as
well as reading and writing motor actions are side effects. Examining the Prolog
control program and the resulting behaviour helps to understand how the rea-
soning process in the Prolog interpreter works. This is one reason, why we want
to use the presented method in our logic programming lectures.

Advanced experiments involving planning problems and other tasks that can
be solved by Prolog programs demonstrate the power of the logic programming
paradigm in robotics. In [5] and [2], advanced AI approaches are documented
for predecessors of the LEGO® MINDSTORMS® EV3 platform. Because of the restricted computing power in those earlier versions, plans are mostly generated outside the robot and then transferred to the robot for execution. The enhanced computing power of the EV3 robot allows the computation and modification of plans directly at the robot.

The next step is expanding the number of predicates to reach parity with the interface that is made available through ev3dev. Automatically detecting sensors from the available devices within their respective subsystems seems convenient as well.

We plan to integrate the presented bindings in robotics experiments in lab sessions on robotics, AI and logic programming. This requires a collection of Prolog configurations for special EV3 robot types appropriate to different tasks such as line following or picking objects with a claw. Next step is to develop a library of basic actions (such as, go 30 cm, turn 60 degree right) to define complex behaviour in a more abstract way.

With this ongoing work, we want to provide a convenient and easily accessible way to demonstrate the power of logic programming and AI in robot control.

References

8. ev3dev [http://www.ev3dev.org](http://www.ev3dev.org)
How I Teach Functional Programming

Johannes Waldmann

Fakultät IMN, HTWK Leipzig, D-04275 Leipzig

Abstract. I teach a course Advanced Programming for 4th semester bachelor students of computer science. In this note, I will explain my reasons for choosing the topics to teach, as well as their order, and presentation. In particular, I will show how I include automated exercises, using the Leipzig autotool system.

1 Motivation

In my course Advanced Programming, I aim to show mathematical models (say, the lambda calculus) as well as their concrete realization first in its “pure” form in a functional programming language (Haskell [Mar10]), then also in some of today’s multi-paradigm languages (imperative and object-oriented, actually mostly class-based) that students already known from their programming courses in their first and second semester. My course is named “advanced” because it introduces functional programming (but it’s not advanced functional programming).

I will explain this in some detail now. Motivation (this section) and discussion will be somewhat opinionated, starting with the following paragraph. I still think it will be clear on what facts my opinion is based, and I hope that at least the facts (examples and exercises) are re-usable by other academic teachers of functional programming. The slides for the most recent instance of this lecture are at https://www.imn.htwk-leipzig.de/~waldmann/edu/ss17/fop/folien/, and online exercises can be tried here: https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=238

When teaching programming, the focus should really be on how to solve the task at hand (given an algorithm, produce executable code) using available tools (programming languages, libraries), and to understand the fundamentals that these tools are based on. We should not commit to teaching random quirks of programming languages of the day — although it can be instructive to compare languages, and follow their design process.

It is interesting to note that major “advances” in “mainstream” languages largely consist of the introduction of ideas that are well-known from functional programming, for decades. Java 5 (2004): generic polymorphism, first in ML (1975); Java 8 (2014): lambda expressions, first in LISP (1958); C# LINQ (2007): (higher order functions and) monads, first in Haskell (1990). Before these languages admitted functional programming, there was a whole cottage industry for simulating higher-order functions as “design patterns” [GHJV95]. And of course
JavaScript (1995) is exactly LISP (1958) — with the superficial change that lots of parentheses had been replaced with lots of braces, to fool the unsuspecting C programmer.

2 Topics

The course contains these topics, which are presented in the order given, except for those in parentheses, which will be delayed by a few weeks.

– first-order data:
  • model: trees over a signature,
  • Haskell realization: algebraic data types
  • (object-oriented simulation: the composite design pattern)
– first order programs:
  • model: term rewriting, equational reasoning
  • Haskell realization: oriented equations and pattern matching
– higher-order data and programs
  • model: lambda calculus
  • pure realizations: actual lambda expressions,
  • (alternative realizations: the strategy design pattern, functional interfaces in Java 8)
– application: patterns for systematic recursion
  • model: algebra over a signature
  • Haskell realization: fold
  • simulation: the visitor design pattern
– restricted polymorphism
  • Haskell: type classes,
  • simulation: Java: interfaces, bounds on type parameters
  • killer application: type-directed generation of test cases
– evaluation on demand, in particular, for infinite streams
  • model: (non) strictness of functions, lazy evaluation,
  • simulation: the iterator design pattern
– higher order functions for stream processing
– functional reactive programming
  • model: behaviours (time-dependent values) and events,
  • alternative: the observer design pattern

The course is mandatory for B. Sc. students of computer science (Informatik and Medieninformatik) in the 4th semester. It consists of one lecture, and one lab class, per week. I’ve been teaching it for roughly five years now, in its present form. Details change. The latest addition is functional reactive programming.
3 Exercises

The course also includes homework. Some exercises require proofs (well, let’s say, they require argumentation) and will be discussed in class. Others are of a more mechanical nature, so they can be graded automatically (and still be discussed in class). For automation, I use the Leipzig autotool system [Wal17a].

Automated exercises come in two variants:

- with domain-specific syntax and semantics: e.g., for term rewriting and lambda calculus,
- with Haskell syntax and semantics: students fill in holes in a Haskell program, such that predefined tests are valid.

For the first variant, autotool runs an evaluation in a problem-specific monad that would typically print a lot of information along the way. For the second variant, autotool evaluates an actual Haskell expression, so there is no tracing. Students are encouraged to evaluate expressions from the program on their own, in a ghci session.

4 First Order Data and Programs

In the lecture, tree-like data is described first, and computation on that data (via pattern matching, case) next. I try to avoid built-in data types for numbers (or later, lists). When I need numbers, I write the data declaration for Peano numerals.

4.1 Exercise on Algebraic Data Types

The mathematical model for trees is terms over a signature. This is known from the course Modeling in the first semester. Students learn the Haskell representation, and solve exercises like this one:

- Instance: a set of data declarations, a type $T$, a number $n$
- Solution: $n$ distinct elements of type $T$

Example instance (this is the full text of the exercise):

```haskell
module Blueprint where
import qualified Data.Set as S
-- imported from Prelude:
-- data Bool = False | True
data C = R | G | B deriving (Eq, Ord, Show)
data T = X C | Y Bool Bool deriving (Eq, Ord, Show)
solution :: S.Set T
solution = S.fromList undefined
test :: Bool
test = S.size solution == 7
```
This is a pattern for a Haskell program text. The student is to replace `undefined` with an expression, such that `test` evaluates to `True`.

Example solution (only the relevant lines are shown):

```haskell
solution :: S.Set T
solution = S.fromList [ X R, X G, X B,
                      , Y False False, Y False True, Y True False, Y True True ]
```

For recursive data types, we can set a lower bound on the cardinality:

```haskell
data C = R | G | B deriving (Eq, Ord, Show)
data D = U | V C deriving (Eq, Ord, Show)
data S = P | Q D S deriving (Eq, Ord, Show)
solution :: S.Set S
solution = S.fromList undefined
test :: Bool
test = S.size solution >= 7
```

With all “complete the code” exercises, a design goal is to have them self-contained. The student can load the problem statement in a `ghci` session, and it should be correct syntactically and statically. This works since `undefined` has any type. The student can then change the source text, and evaluate `test`, or any other expression. It seems unavoidable that the code will contain items that the student cannot understand from the lecture alone (at this point). In this example, this applies to `deriving (Eq, Ord, Show)`, and the use of `Data.Set`.

### 4.2 Exercise on Term Rewriting

The mathematical model for processing tree-like data is *term rewriting* [BN98]. The following type of exercise helps to get a basic understanding of this model of computation.

– Instance: a term rewriting system $R$ over signature $\Sigma$, terms $s$ and $t$ over $\Sigma$;
– Solution: a sequence of $R$-rewrite steps that transforms $s$ to $t$, where a step is given by
  * (number of) rule to apply,
  * position of application (where position is sequence of natural numbers),
  * substitution for variables in rule.

This definition is copied verbatim from the definition of the rewrite relation $s \rightarrow_R t : \exists (l,r) \in R, p \in \text{Pos}(s), \sigma \in \text{Var} \rightarrow \text{Term}(\Sigma) : s[p] = l\sigma \land t = s[p := r\sigma]$ that was given in the lecture.

Example instance:

```haskell
for the system TRS
{ variables = [ x, y, z]
  , rules = [ f (f (x, y), z) -> f (x, f (y, z))
           , f (x, f (y, z)) -> f (f (x, y), z) ] }
give a sequence of steps
from \( f ( f ( f (a , b ), f (c , d )), e ) \)
to \( f (a , f (f (b , c ), f (d , e ))) \)

Example solution (attempt):

\[
( f ( f ( f (a , b ), f (c , d )), e ) , \begin{array}{l}
\text{[ Step { rule_number = 0 , position = [ 0, 1 ] , substitution = listToFM}
\end{array}
\begin{array}{l}
\text{[ ( x, f (a , b )), ( y, f (c , d )), ( z, e ) ] } ] )
\end{array}
\]

Example output of autotool for above input (slightly edited)

apply step Step { rule_number = 0, ... }
to term
\( f ( f ( f (a , b ), f (c , d )), e ) \)
the rule number 0
is \( f ( f (x, y), z) \rightarrow f (x, f (y, z)) \)
the subterm at position [ 0, 1]
is \( f (c , d ) \)
the instantiated lhs is
\( f ( f (a , b ), f (c , d )), e ) \)
agrees with subterm at position?
No.

This exercise type allows some modifications (not all are implemented currently):

– minimal, maximal length of sequence is given
– start term not given, target term not given, or given by a constraint (e.g., a basic term, a normal form)
– admissible steps restricted by some strategy (e.g., outermost)

It is important to note that this is an exercise on (a model of) programming, but the task is not “guess the program” but “describe the execution of a given program” (by giving its execution steps in detail).

### 4.3 Exercises on First Order Programs

The main point is case distinctions on algebraic data types. In the lecture, I emphasize that it is best if a set of patterns in a case distinction is complete (it covers all values) and disjoint, and this is achieved easily by writing one pattern for each constructor of the data type of the discriminant. I strongly recommend to write these patterns in order of declaration, e.g., Zero before Successor, and Nil before Cons (but this needs polymorphism, which happens only later).

Here is an example for an exercise that I use in class: Given

```plaintext
data Bool = False | True
data T = F T | G T T T | C
```
answer for each of the following expressions:

- is it syntactically correct
- is statically correct
- what is its result (its dynamic semantics)
- is the pattern match complete? disjoint?

1. case False of { True -> C }
2. case False of { C -> True }
3. case False of { False -> F F }
4. case G (F C) C (F C) of { G x y z -> F z }
5. case F C of { F (F x) -> False }
6. case F C of { F (F x) -> y }
7. case F C of { F x -> False ; True -> False }
8. case True of { False -> C ; True -> F C }
9. case True of { False -> C ; False -> F C }
10. case C of { G x y z -> False; F x -> False; C -> True }

For self-study exercises, I use “complete this Haskell code”, cf. Subsection 4.1.

- instance: a pattern for Haskell program (undefined can be replaced by arbitrary expression) that contains a definition of an expression test::Bool
- solution: a program that is an instance of the pattern such that test evaluates to True.

The specification can be given by a concrete test case, but it is much better to give it as a property: a function \( p : D \rightarrow \text{Bool} \) that encodes \( \forall x \in D : p(x) \). Students should learn from the start to use property-based testing, e.g., smallcheck [Che13], and later they can also learn how automatic enumeration of test cases works (with Haskell type classes).

Care must be taken to not reveal the answer when writing the property. As an example, the task is to implement the min operation for Peano numbers, and we can specify it using a correct implementation of addition

import Prelude hiding (min)

data N = Z | S N deriving (Show , Eq)

min :: N -> N -> N
min x y = undefined

spec1 = \ x y -> min x y == min y x
spec2 = \ x y -> min (plus x y) x == x

plus :: N -> N -> N
plus x y = case x of { Z -> y ; S x' -> S (plus x' y) }
4.4 A Side Remark on Automated Testing

I think that smallcheck is too complicated already, as its implementation uses LogicT, which cannot be explained easily. For that reason, I am inclined to switch to leancheck [Mat16] because they essentially have

```haskell
class Serial a where series :: [[a]]
```

which can be explained later, cf. Subsection 6.3. To get instances of Serial for user-defined data types, I can write down an instance

```haskell
instance Monad m => Serial m N where series = cons0 Z \/ cons1 S
```
or I can have them derived

```haskell
data N = Z | S N deriving (Eq, Generic) ; instance Serial m N
```

Both involve magic, i.e., concepts (Monads, Generic) that the student cannot grasp at this point.

Also I found that smallcheck uses IO too often in its API. To work around this, I need (some lines of) boilerplate, and copy it in each exercise, like this:

```haskell
test :: Bool
test = and [ null $ failures 10 1000 $ spec1
            , null $ failures 10 1000 $ spec2 ]
-- | first f failures from t testcases for property p
failures f t p = take f $ filter ( \ x -> not $ p x )
            $ take t $ do d <- [ 0 .. ] ; list d series
```

5 Polymorphism

A main motivation for polymorphic data (that is, type constructors), are container types (lists, trees). Again, I prefer to write all data declarations, and not use types from the Prelude. Students should definitely not get the impression that lists are somehow intrinsic to Haskell [Wal17b].

5.1 Exercise on polymorphic types

- instance: set of data declarations, some of them for type constructors, a number n, a type T
- solution: a set of n expressions of type T with distinct values

Example instance:
{- using these types and type constructors from Prelude: 
data () = ()
data Bool = False | True
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
-}

data C = R | G | B deriving (Eq, Ord, Show)
data Pair a b = Pair a b deriving (Eq, Ord, Show)

solution :: S.Set (Either (Pair Bool (Maybe ())) (Maybe (Maybe C)))
solution = S.fromList undefined

test :: Bool
test = S.size solution >= 9

5.2 Exercise on polymorphic functions

They are of the type “complete this Haskell program”, and earlier remarks apply.
Now that have polymorphism, we can use it to emphasize some extra points:

We can specify a polymorphic type for what the student should write

reverse :: List a -> List a
reverse xs = undefined

and give a monomorphic test case

reverse (Cons True (Cons False Nil)) == Cons False (Cons True Nil)

The polymorphic type will prevent the student from writing
everse xs = Cons False (Cons True Nil)

This makes it harder to cheat. Of course, one test is not enough, think of

reverse xs = append (tail xs) (Cons (head xs) Nil)

Then we can add another property like

prop xs = reverse (reverse xs) == xs

and it seems then we reached the threshold of “sufficient specification” where
the intended solution is smaller than any cheating “solution”.

6 Higher Order Data and Functions

The main motivation for higher order features is that they allow to abstract over
programs, as a higher order function can be seen as a schema for a program. The
underlying mathematical model is the lambda calculus [Bar84]. Of course, in the
lecture I just use the basic computational model, and don’t do any “theory”. I
mention termination and confluence.
6.1 Exercises on Lambda Calculus

This exercise is similar to the one on term rewriting (Subsection 4.2).

– instance: lambda terms $s,t$, possibly some restrictions
– solution: sequence of $\alpha$ and $\beta$ steps that transforms $s$ to $t$, and conforms to restrictions.

Example instance:

give a derivation that transforms $(\lambda \ x \ y \ . \ y \ x) \ (y \ y) \ (\lambda \ x \ . \ x)$ to $y \ y$

Here, I use the standard abbreviations for nested lambdas, and nested application. To be absolutely sure, autotool will display the abstract syntax tree.

Example solution (attempt):

```
[ Step { position = [ 0 ]
  , action = Reduce
    { formal = x , body = \ y -> y x , argument = y y } } ]
```

Example output trace (slightly edited):

Current step is Step { position = [ 0], ... }
Subterm at position [ 0] ist
$(\lambda \ x \ y -> y \ x) \ (y \ y)$
Error: when replacing x with y y:
in \ y -> y x ,
a free occurrence of a variable would become bound

In a correct solution, this step should be prefixed with

```
Step { position = [ 0,0,0]
  , action = Rename { from = y , to = z } }
```

The necessity of local renamings is also discussed in the lecture. It may appear as a technical shortcoming of the calculus, but similar problems will appear in any (programming) language that has local names, when the program is modified (refactored) by expanding function definitions. This can be shown by comparing the following Java snippets (that can be executed in jshe11 of JDK 9):

```java
int x = 3;
int f(int y)
{ return x + y; }
int g(int x)
{ return (x + (x+8)); }
// g(5) => 18
```

As a side remark, I also show to the students that JavaScript [TWB17] meanwhile does almost behave like a real programming language. There is really not much remaining difference (well, except for lack of static typing) between
6.2 Exercises on Fold

A prime motivation for higher-order functions is that they can be used to express recursion schemes over recursive algebraic data types. The point is that “instantiating the schema” is just function application, because the schema is a function, and it needs to be higher-order for this to work.

In the lecture I use the phrase “when a fold is applied, each constructor gets replaced by a corresponding function”, and from that, it is quite clear that each algebraic data type has exactly one recursion schema, and its type and implementation can be derived mechanically. I do this for Peano numbers, lists, trees; but it is also a nice exercise to have students construct and discuss the “fold” for `Bool`, `Maybe`, and `Either`. After this, I have students look up in https://www.haskell.org/hoogle/ whether functions of that type are already in the standard library. Of course they are. This shows the value of static types for documentation.

Here is a programming exercise example: for Peano numbers, `fold` and `plus` are given, `minus` should be implemented.

data `N = Z | S N` deriving (Eq, Show)

fold :: b -> (b -> b) -> (N -> b)
fold z s x = case x of
  Z -> z
  S x' -> s (fold z s x')

plus :: N -> N -> N
plus x y = fold y S x

minus :: N -> N -> N
minus x y = (fold undefined undefined) y

spec1 :: (N,N) -> Bool
spec1 = \( (x,y) \) \( minus \ (plus x y) y == x \)

spec2 :: (N,N) -> Bool
spec2 = \( (x,y) \) \( minus x \ (plus x y) == Z \)

In the lecture, I describe how to solve this kind of exercise “write function `f` as fold”. The student should draw an example input tree (for the argument of the fold) and write at each node the required result of `f` on that subtree. Then they can read off test cases for the unknown arguments of fold. With the above
example, let $x = S (S (S Z))$ and consider $\text{minus} \ x \ (S (S Z))$. According to specification, the result is $S Z$. Subtrees to be evaluated are $\text{minus} \ x \ (S Z)$ with result $S (S Z)$ and $\text{minus} \ x \ Z$ with result $S (S (S Z))$.

So if $\text{minus} \ x \ y = \text{fold} \ z \ s \ y$, then we see that $n = x$ and, for example, $s (S (S (S Z))) = S (S Z)$. So quite clearly $s$ is the predecessor function, with a modification ($\text{pre} \ Z = Z$).

Now I also make a point of showing that some functions are not folds. For instance, if we would have a presentation $\text{pre} = \text{fold} \ z \ s$, then on the one hand $Z = \text{pre} (S Z) = s (\text{pre} Z) = s Z$ but on the other hand we have $S Z = \text{pre} (S (S Z)) = s (\text{pre} (S Z)) = s Z$, which is a contradiction, since $s$ is a function. Note that this reasoning only works for pure functional programs.

This leads the way for an extension of the above exercise: $\text{pre}$ can be defined as the projection of a helper function, which is a fold.

```haskell
pre :: N -> N
pre x = case pre' x of
  Pair y z -> z

spec3 :: N -> Bool
spec3 = \ x -> pre (S x) == x

data Pair a b = Pair a b deriving (Eq, Show)

pre' :: N -> Pair N N
pre' = fold undefined (\ x -> case x of Pair y z -> undefined)

spec4 :: N -> Bool
spec4 = \ x -> pre' (S x) == Pair (S x) x
```

### 6.3 Type Classes

There are cases where full generic polymorphism does not capture the intention of the programmer. For instance, the type of $\text{sort}$ cannot be $\text{sort} :: \text{List} \ a \rightarrow \text{List} \ a$ because we need to be able to compare elements. So it is $\text{sort} :: \text{Ord} \ a \Rightarrow \text{List} \ a \rightarrow \text{List} \ a$

I mention the standard type classes $\text{Show}$, $\text{Eq}$, $\text{Ord}$, $\text{Ord}$.

The “killer example” for type classes is automated type-directed generation of arguments for property tests. The essential classes (for leancheck) are

```haskell
check :: Testable a => a -> IO ()
class Testable a where
  results :: a -> [[String,Bool]]
instance Testable Bool where
```
results p = [([],p)]
instance (Listable a, Testable b)
  => Testable (a -> b) where ...
class Listable a where tiers :: [[a]]
instance Listable Int where ...

If an expression check \( \lambda x y . x + y == y + x :: \text{Int} \) is type-checked, the compiler notes that
\( \lambda x y . x + y == y + x :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Bool}) \)
and then goes on to prove that

\( \text{instance Testable (Int \rightarrow (Int \rightarrow \text{Bool})}) \)

can be derived from the axioms given above. This is a nice showcase of formal deduction put to very practical use.

I also compare to means of restricting polymorphism in Java. The situation is a bit messed up because of Java's historical baggage. For instance, \texttt{equals} is in class \texttt{Object} and thus has the wrong type, and it needs Eclipse’s “derive (hashCode and) equals” (from the “source” menu) to implement it correctly.

But — Haskell has some historical baggage as well: notice the unnatural appearance of \( :: \text{Int} \) in the example above. Of course the natural place for this type declaration would be right where \( x \) is declared:

\( \lambda (x :: \text{Int}) (y :: \text{Int}) . x + y == y + x \)

If I try this, ghc tells me that this is illegal (wat?) and that it can only be enabled with \texttt{ScopedTypeVariables}. WAT? Where’s the type variable?

7 Strictness, Laziness, Streams

There are two motivations for demand-driven evaluation. One is that it may help to save work. The other, more important, is that it is needed for modularity of programming [Hug89]. The classical example is \texttt{if-then-else} as a function, which will not work as expected with strict evaluation, since it forces the values of both branches.

7.1 Strictness

In lecture, I give the usual definition that \( f \) is strict in a specific argument position iff \( f(\ldots, \bot, \ldots) = \bot \). This definition can be applied directly in \texttt{ghci}:

\texttt{False && undefined -- does not raise an exception}

This short-cut evaluation of Boolean connectives also shows that other languages have some non-strict operations, but only as special cases.

I go on to discuss reasons for evaluation. Where does the “demand” in “on-demand” evaluation come from? Without giving a full account of the dynamic
semantics of Haskell, I teach the principle that: if the value of a case expression is demanded, then the value of its discriminant will be demanded, to the extent that is necessary for deciding which pattern matches, and then the value of the chosen branch is demanded.

Students can use this to prove that

\[ f :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \]
\[ f \ x \ y = \text{case } y \ of \ { \text{False} \rightarrow x ; \text{True} \rightarrow y } \]

is strict in the second argument, and they can evaluate \( f \ \text{undefined} \ True \) to prove that it's not strict in the first. Another question of this type is: analyze strictness of

\[ g :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \]
\[ g \ x \ y \ z = \text{case } (\text{case } y \ of \ { \text{False} \rightarrow x ; \text{True} \rightarrow z } \ of \ { \text{False} \rightarrow x \text{ True} \rightarrow \text{False} } \]

and, assuming standard definitions of Boolean connectives,

\[ f \ x \ y \ z = \text{case } y \ \&\& \ z \ of \ { \text{False} \rightarrow \text{case } x \ |\| \ y \ of \ { \text{False} \rightarrow z \text{ True} \rightarrow \text{False} } \text{ True} \rightarrow y } \]

Actually, it would be nice to have automation for this kind of exercise.

### 7.2 Streams

In particular, laziness is helpful for separating the processing of data streams into producer, transformer, and consumer. Often, the transformer is written using higher order functions (map, bind), and the consumer is a fold.

This is the only place in the lecture where students actually use Haskell’s built-in list type and operations. In the lecture, I emphasize the difference between \texttt{foldr} (the “right” fold) and \texttt{foldl}. A nice exercise is: write a function

\[ \text{fromBits} :: [\text{Bool}] \rightarrow \text{Integer} \]
\[ \text{fromBits} [\text{True,False,False,True,False}] = 18 \]

with one of the folds. Is it \texttt{foldl}, or is it \texttt{foldr}? Prove that one of these works, and that the other one does indeed not.

I also compare to the corresponding stream processing libraries LINQ for C# and \texttt{java.util.stream} for Java 8. I will not go into detail here, and just list the correspondence:
7.3 Behaviours and Events

To conclude the lecture with a “real life” application of (ideas from) functional programming, I briefly discuss functional-reactive programming [EH97].

The approach is to give a mathematical model of behaviours as values that depend on the time, and events, which can be thought of as behaviours that are piece-wise constant, and can be represented as streams of pairs of time and value, with increasing time.

With this model, interactive software can be written in a modular way, by using combinators for behaviours and events. This is in contrast to the method of event processing via the observer pattern, which quickly leads to callback hell.

In class, I use threepenny-gui [Apf13]. One of this library’s examples is

```haskell
-- dollar <- UI.input ; euro <- UI.input
getBody window #+ [ column [ grid [[string "Dollar:" , element dollar]
  ,[string "Euro:" , element euro ] ] ]

-- euroIn <- stepper "0" $ UI.valueChange euro
dollarIn <- stepper "0" $ UI.valueChange dollar

-- let rate = 0.7 :: Double
withString f = maybe "-" (printf "%.2f") . fmap f . readMaybe

dollarOut = withString (/ rate) <$> euroIn
euroOut = withString (* rate) <$> dollarIn

-- element euro # sink value euroOut
element dollar # sink value dollarOut
```

This uses combinators for behaviours and events like

```haskell
-- stepper :: MonadIO m => a -> Event a -> m (Behavior a)

-- and general combinators like

(<$>) :: Functor f => (a -> b) -> f a -> f b

-- The point is that the Functor abstraction is standard (not specific to this library) and makes it easier to understand and to use. Behaviours also instantiate Applicative, so we can use

(<<*) :: Applicative f => f (a -> b) -> f a -> f b

-- to combine them. A quick exercise for the above code is to add another output element that shows the concatenation of dollar and euro behaviours. This can be done with
```
I admit that this is on the border from introductory to advanced functional programming, and a full explanation is outside the scope of the lecture. On the other hand, this may just be a gut reaction by the seasoned Haskell programmer (“they destroyed Haskell with Applicative, Foldable, and whatnot”) that sees his familiar environment (e.g., the type of foldl) change.

8 Discussion

Regarding evolution of programming languages, I find it amusing to observe current developments towards, and also against, static typing.

Static types document design decisions in a way that can be checked by machine. So it detects software errors at compile time, when they are cheaper to repair (instead of runtime), and it allows for more efficient execution (by the removal of runtime type checks). So, alongside higher order functions and generic polymorphism mentioned above - there can be no doubt that static typing is the correct thing to do, and thus, the right thing to teach. In Dijkstra’s words, “It is not the task of the University to offer what society asks for, but to give what society needs.” [Dij00]. I claim that society needs static typing.

But what do market forces in the software industry think? On the one hand, there are serious proposals to introduce gradual type systems [Sav14] for untyped languages (which are marketed as “dynamic”). On the other hand, there are proposals to open up JVM and C# for “dynamics”. I think all of this is mostly marketing, and it should not influence teaching.

Let us hope that TypeScript [typ17] (or MyPy [Leh17]) wins before departments succumb and throw out statically typed languages from introductory courses.

References


Author Index

Abreu, Salvador 3
Achuthan, Krishnahsree 19
Barbosa, João 268
Bayot, Roy 187
Brass, Stefan 171
Bry, François 155
Carnaz, Gonçalo 187
Costa, Vítor Santos 268
Dageförde, Jan C. 76
Fang, Min 107
Florido, Mário 268
Frühwirth, Thom 35, 60
Furfaro, Angelo 217
Gall, Daniel 60
Gonçalves, Teresa 187
Hanus, Michael 233
Hofstedt, Petra 91
Hölldobler, Steffen 139
Jayaraman, Bharat 19
Kalkus, Jona 242
Kannimoola, Jinesh M 19
Körner, Philipp 252
Krings, Sebastian 252
Kuchen, Herbert 76
Leutgeb, Lorenz 123
Liu, Ke 91
Loeffler, Sven 91
Mörbitz, Richard 139
Nogatz, Falco 242
Nogueira, Vitor Beires 187
Oberschweiber, Jonas 233
Pedro, Vasco 3
Quaresma, Paulo 187
Richter, Frank 60
Roque, Pedro 3
Saccà, Domenico 217
Saldanha, Emmanuelle-Anna Dietz 139
Schwarz, Sibylle 284
Seipel, Dietmar 242
Tikovsky, Jan Rasmus 201
Tompits, Hans 107
Waldmann, Johannes 292
Weinzierl, Antonius 123
Wenzel, Mario 284
Wolf, Armin 50