RWTH Aachen

Department of Computer Science
Technical Report

WFLP'04
13th International Workshop on Functional and (Constraint) Logic Programming

Herbert Kuchen (Editor)

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2004-05

RWTH Aachen · Department of Computer Science · June 2004

The publications of the Department of Computer Science of RWTH Aachen (*Aachen University of Technology*) are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Preface

This volume contains the proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP), which took place on June 1 at Aachen, Germany, as part of RDP'04, the Federated Conference on Rewriting, Deduction, and Programming. WFLP is a platform for the exchange of ideas between researchers interested in functional programming, (constraint) logic programming, as well as their integration. It promotes 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 languages and related areas.

WFLP has some considerable tradition already. The previous WFLP editions took place at Valencia, Spain, 2003, Grado, Italy, 2002, Kiel, Germany, 2001, Benicassim, Spain, 2000, Grenoble, France, 1999, Bad Honnef, Germany, 1998, Schwarzenberg, Germany, 1997, Marburg, Germany, 1996, Schwarzenberg, Germany, 1995 Schwarzenberg, Germany, 1994, Rattenberg, Germany, 1993, and Karlsruhe, Germany, 1992.

WFLP'04 covers all areas of functional and (constraint) logic programming, including but not limited to:

- Language Design: modules and type systems, multi-paradigm languages, concurrency and distribution, objects
- Foundations: formal semantics, rewriting and narrowing, non-monotonic reasoning, dynamics, type theory
- Implementation: abstract machines, parallelism, compile-time and run-time optimizations, interfacing with external languages
- Transformation and Analysis: abstract interpretation, specialization, partial evaluation, program transformation, meta-programming
- Software Engineering: design patterns, specification, verification and validation, debugging, test generation
- Integration of Paradigms: integration of declarative programming with other paradigms such as imperative, object-oriented, concurrent, and real-time programming
- Applications: declarative programming in education and industry, domain-specific languages, visual/graphical user interfaces, embeded systems, WWW applications, knowledge representation and machine learning, deductive databases, advanced programming environments and tools.

Nine papers have been submitted to WFLP'04 and considered by the program committee. After some careful peer reviewing all of them have been accepted for presentation at the workshop.

I would like to thank all the people who contributed to the suceess of WFLP'04, in particular the authors for their presentations, the members of the program committee and the reviewers for their careful work and their valuable hints to the authors, enabling them to improve their presentations. My special thanks go to the organizers of RDP'04, in particular to Jürgen Giesl, the RDP'04 conference chair, for their continuous support and the perfect organization.

Münster, Germany, May 1,

Herbert Kuchen

Program Committee

María Alpuente	Technical University of Valencia	Spain
Sergio Antoy	Portland State University	USA
Manuel Chakravarty	UNSW Sydney	Australia
Olaf Chitil	University of Kent	UK
Rachid Echahed	Institut IMAG	France
Moreno Falaschi	Università di Udine	Italy
Michael Hanus	CAU Kiel	Germany
Petra Hofstedt	TU Berlin	Germany
Zhenjiang Hu	University of Tokyo	Japan
Tetsuo Ida	University of Tsukuba	Japan
Herbert Kuchen (Chair)	University of Muenster	Germany
Francisco Javier Lopez-Fraguas	Universidad Complutense de Madrid	Spain
Juan José Moreno-Navarro	Universidad Politécnica de Madrid	Spain
Germán Vidal	Technical University of Valencia	Spain

Additional Referees

Gianluca Amato, Demis Ballis, Rafael Caballero Roldan, Santiago Escobar, Mircea Marin, Claudio Ochoa, Wendelin Serwe, Sean Seefried, Josep Silva.

Contents

TeaBag: A Functional Logic Language Debugger*

Sergio Antoy Stephen Johnson

Computer Science Department, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
{antoy,stephenj}@cs.pdx.edu

Abstract We describe a debugger for functional logic computations. The debugger is an accessory of a virtual machine currently under development. A distinctive feature of this machine is its operational completeness of computations, which places novel demands on a debugger. We give an overview of the debugger's features, in particular the handling of non-determinism, the ability to control non-deterministic steps, to remove context information, to toggle eager evaluation, and to set breakpoints on both functions and terms. We briefly describe the debugger's architecture and its interaction with the associated virtual machine. Finally, we describe a short debugging session of a defective program to show in action debugger features and window screenshots.

1 Introduction

Functional logic programming joins in a single programming paradigm characterizing features of functional and logic programming. There are a number of languages with this aim, e.g.,Curry [23], Escher [26], Le Fun [2], Life [1], Mercury [36], NUE-Prolog [29], Oz [35] and Toy [27], to name a few. These languages support user-defined functions and the subsequent evaluation of expressions involving these functions. Debugging functional computations of this kind is a non-trivial, but well-studied problem [5,14,15,18,31,32,37,38,40,42]. These languages also support the use of logic variables. Debugging programs with the combination of user-defined functions and logic variables is much more challenging for reasons that will be discussed shortly.

Indeed, programming with the combination of user-defined functions and logic variables is the subject of active research even for its most fundamental aspects, e.g., the formulation of both adequate semantics and efficient implementations. A significant problem of combining functions and logic variables is what to do when the execution of a program leads to the evaluation of a functional expression containing uninstantiated logic variables. This problem is solved by either residuation or narrowing [19]. *Residuation* delays the evaluation by transferring control to some other portion of the program in hopes that the variables will be instantiated by a predicate so that the evaluation of the functional expression can continue. *Narrowing*, instead, guesses instantiations of variables which allow the evaluation to continue. Thus, the result of evaluating by narrowing an expression produces both the value of the expression, generalizing a functional computation, and a substitution of some variables of the expression, generalizing a logic computation. The details of this computation are quite technical and outside the scope of this discussion. Examples will be provided in the next section.

Narrowing introduces non-determinism in the sense that distinct instantiations of a variable in an expression may be equally plausible and different instantiations may lead to different values. This suggests to allow functions (including constants seen as functions of zero arguments) that for the same arguments return different results. Obviously, these "things" are not functions in the mathematical sense, but they are defined and used as ordinary functions in a program. The semantics of a functional logic program is often formulated by seeing the program as a first-order rewrite system. Higher-order and partially applied functions are eliminated by a transformation referred to

* This research has been supported in part by the NSF grants CCR-0110496 and CCR-0218224.

as *firstification* [8,41]. Then, the execution of a program consists in the evaluation by narrowing of an expression using the rules of the rewrite system.

The characteristics discussed above pretty much shape a debugger for a functional logic language. The debugger has the features generally found in tracing debuggers for functional languages. It shows evaluation steps as reductions. In the case of a functional logic language, rather than the β-reductions of the λ calculus, these reductions are narrowing steps of a rewrite system—for first-order, variable-free expressions the difference between the two is small. In addition, the debugger must handle logic variables and non-determinism. How this is done depends on what a functional logic language provides. Our work is centered on Curry [23] and on an implementation of Curry, referred to as the *FLVM*, currently under development [7]. Curry offers both narrowing and residuation and the *FLVM* offers a complete implementation of non-determinism. How these characteristics affect a debugger will be discussed at length in the following sections.

Section 2 contains background information on Curry and the *FLVM*. Section 3 presents an overview of the significant features of our debugger. Section 4 sketches the architecture of the debugger and how it interacts with the *FLVM*. Section 5 shows an example of a debugging session. Section 6 discusses related work. Section 7 offers our conclusion.

2 Background

Curry provides built-in types, such as numbers and characters; user-defined algebraic data types; functions, including higher-order and non-deterministic ones, defined by pattern matching; lazy evaluation; logic variables; and built-in search. The syntax is Haskell-like. An example of a complete program follows. The numbers to the left are not part of the program. They are used for reference purposes only.

```
 1  data Color = red | white | blue

 2  mono _ = []
 3  mono c = c : mono c

 4  solve flag | flag =:= x ++ white:y ++ red:z
 5             = solve (x ++ red:y ++ white:z)
 6             where x,y,z free
 7  solve flag | flag =:= x ++ blue:y ++ red:z
 8             = solve (x ++ red:y ++ blue:z)
 9             where x,y,z free
10  solve flag | flag =:= x ++ blue:y ++ white:z
11             = solve (x ++ white:y ++ blue:z)
12             where x,y,z free
13  solve flag | flag =:= mono red ++ mono white ++ mono blue
14             = flag
```

Line 1 defines the type `Color` whose instances are three constants. Lines 2 and 3 define a non-deterministic function, `mono`, that takes an argument (of type `Color`) and returns a list whose elements are all equal to the argument. The textual order of the rules is irrelevant. Lists of any length can be returned. The remaining lines define a function, `solve`, that "solves" the *Dutch National Flag* problem in the spirit of [16], i.e., by swapping pebbles out of place.

The function `solve` is defined by conditional rules of the form:

$$f\ t_1 \ldots t_n \ \texttt{|}\ c\ \texttt{=}\ e \quad \texttt{where}\ vs\ \texttt{free}$$

The conditions are *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 evaluate to unifiable data terms. Free variables, introduced by the `where` clause, in a condition may be instantiated by narrowing steps, if this is useful to satisfy the condition.

In contrast to other declarative programming languages, e.g., Haskell, where the first matching rule is applied, in Curry all matching (to be more precise, unifiable) rules are non-deterministically applied to support complete computations. This enables the definition of non-deterministic functions, such as `mono` and `solve`, which may have more than one result on a given input. As an example of solving constraints, consider the evaluation of the following expression:

```
solve [white,red,blue,white]
```

Both the first and third rule of `solve` can be fired, because the conditions of these two rules are satisfiable. For example, the first condition holds if `x` $=$ `[]`, `y` $=$ `[]` and `z` $=$ `[blue,white]`. These instantiations of `x`, `y` and `z` are computed by the evaluation of the constraint. With these instantiations, the expression is rewritten to `solve [red,white,blue,white]`.

A crucial design decision of the implementation of the language is how to handle the fact that *two or more* rules are applicable to the same expression. One common strategy is to select one rule and delay the application of the others until the selected rule yields either a result or a failure. This is a simple strategy adopted, e.g., by some implementations of Curry [20,28] and other functional logic languages [27]. But it is unsatisfactory because if the application of the selected rule leads to a non-terminating computation no other rule that could yield a result is ever applied.

Another strategy is to fork the computation for every applicable rule and to execute fairly and independently all the results. This is the strategy adopted by the *FLVM*. This design decision is more satisfactory because it ensures the operational completeness of the language implementation. However, it also introduces novel problems for a debugger, since the trace of a computation is no longer the traditional linear sequence of steps, but it has a tree-like structure. A distinctive feature of our debugger is the handling of this structure.

3 Features Overview

In this section we present some characterizing features of our debugger, called TeaBag (The Errors And Bugs Are Gone!). These features are realized by several interactions with a computation. A synopsis of these features follow: *computation structure* is a window that visualizes the non-deterministic steps of a computation; *choice control* is an option for the early elimination of undesired non-deterministic steps; *context hiding* is an option for displaying only a subterm of the term being evaluated and/or only steps that affect this subterm; *eager evaluation* is an option to eagerly evaluate and/or replace a subterm of a term; *runtime debugging* is a debugging mode that supports non-terminating computations and runtime selection of non-deterministic steps; *breakpoints* is an option to set breakpoints not only on functions, but also on terms; *highlighting* is the use of colors and other visual clues to ease understanding.

There are several classes of debugging tools for declarative languages. This subject will be further discussed in Section 6. To understand some of the following features, we recall the difference between a tracer and a runtime debugger. These terms are not formally defined and our descriptions are only a subjective point of view to aid comprehension. A tracer executes a computation

and when the computation terminates it displays some representation of the computation, e.g., the computation steps. A runtime debugger executes a computation and if some events occur it displays information about these events. The events generally include the termination of the computation, runtime errors, and the invocation of certain functions selected by the user. A runtime debugger can provide useful information about non-terminating and/or failing computations. It can also be helpful when debugging code that interacts with the outside world, e.g., the program directly paints to the screen or uses a socket. However, a runtime debugger generally provides less detailed information about ordinary computations.

3.1 Computation Structure

A computation is the set of the narrowing or rewriting steps performed on the term being evaluated. Computations in deterministic languages are a linear sequences of steps. In a non-deterministic language a computation is a tree sometimes called the narrowing space. The narrowing strategy executed by the *FLVM* is essentially an implementation of the *inductively sequential narrowing strategy* [6] with some adjustments to support residuation. In this framework, narrexes and possibly redexes can have more than one replacement. When this happens, a trace forks into several paths—one for each replacement. In other words, a computation has the structure of a tree and a trace is a path in this tree.

TeaBag provides a view of the tree structure of a computation. This view does not show all the rewriting and narrowing steps of the computation. It just shows a tree in which a branch represents a non-deterministic step and a leaf represents the endpoint of a trace, which is a term totally or partially evaluated. In this view, a sequence of deterministic steps is shown simply as an arc from a parent to a child in the tree. This view highlights the current path through the tree that the user is looking at in the trace browser. The trace browser, discussed in section3.2, shows all the rewriting and narrowing steps of a trace.

Having a computation structure is very important to understand how a result is obtained. Without the computation structure it is difficult to know where a rewriting or narrowing step fits into the overall computation. The computation structure lets the user know which non-deterministic steps were made to get to any rewriting or narrowing step in a trace. In deterministic computations, where traces are linear, this contextual information can be obtained with just a counter, but this is impossible in non-deterministic computations. An example of the computation structure is in figure3.

A variable is displayed with both its source code name v and a unique internally generated number n in the format $v|n$. The name aids the user in relating steps of the computation to the source code. Since different variables may have the same name because of either recursion or the scoping rules, the unique number allows the user to distinguish different variables with same name.

3.2 Trace Browser

The trace browser shows the rewriting and narrowing steps for the selected path in the computation structure. There are two ways to view the trace. The first way is as a table of all rewriting and narrowing steps. This view is convenient for getting a “birds eye” view of the trace. However, it is not suitable for examining individual rewriting or narrowing steps of the trace. The second way to view a trace is by examining each rewriting and narrowing step. The user can choose to view one or two terms of the trace at a time. Each term in the trace is obtained from a rewriting or narrowing step on the previous term. The trace browser includes buttons to move to the next, previous, first, and last steps. The user can also select a particular step number to jump to.

The trace browser interacts with the computation structure. The node or edge in the computation structure corresponding to the current step in the trace browser is highlighted. When a non-deterministic step is displayed in the trace browser the user is given the option of which branch to follow. The selected path is highlighted in the computation structure. The user can also select a path in the computation structure and the trace browser will be updated to display that path. An example of the trace browser is in figure 1.

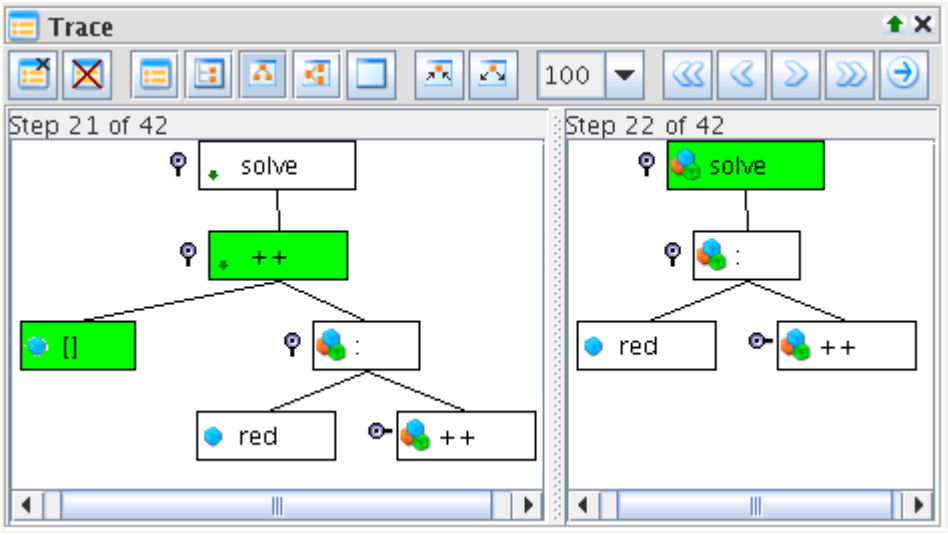


Figure 1. Example of Trace Browser. The terms are displayed as trees. The user can expand and collapse subtrees. Nodes in the tree are symbols and branches are arguments.

3.3 Choice Control

TeaBag includes computation management to control subcomputations originating from non-deterministic steps made during runtime debugging. When a computation executes a non-deterministic step, the *FLVM* evaluates fairly and independently all the results of this step. This is essential for ensuring the operational completeness of a computation. This view allows the user to kill, pause, and activate the subcomputation of any individual result. Often, the user is interested only in a subset of all the choices of a non-deterministic step. Since there can be an exponential growth of non-deterministic steps, being able to pause and kill subcomputations toward the beginning of a computation can greatly reduce the total number of non-deterministic steps made. This makes it easier for the user to debug computations that would normally produce too many steps to examine.

3.4 Context Hiding

Even for small programs, the sheer volume of data to be displayed and analyzed for an execution may become a serious obstacle to debugging. TeaBag alleviates this problem with two features intended to suppress unwanted information.

Term Size Lazy evaluation has a propensity for creating large terms during a computation. Large terms are not displayed easily and they make it hard to find subterms of interest. Often, the programmer is interested in examining a subterm nested somewhere in a large term. To assist the user

in focusing on this term, TeaBag can be instructed to display a subterm of a term of a computation. The subterm to be displayed is selected by the user. An option, allows the user to select only the redex or narrex of each step, i.e., to eliminate the portion of a term above the subterm replaced by a step. Also, TeaBag will expand terms only as much as is needed to display redex, narrex, and created positions, i.e. to eliminate the portion of the term below the parts of the term that were replaced. The user has the option to further expand the term to see more. Hiding the context of a term makes finding pertinent information easier for the user.

Trace Length Likewise, the number of steps in a trace can be very large. Many times the user will only want to look at a subset of the trace steps, in particular at all the steps that affect a particular subterm of the term being evaluated. For example, this may be convenient for terms rooted by a function thought to be defective. TeaBag lets the user choose which terms to trace. The trace for that term will only have the rewriting and narrowing steps performed on that term or one of its subterms. This feature limits the number of rewriting and narrowing steps that the user needs to look at. This makes it possible for the user to examine traces that would normally be too long to look at.

The programmer may want to look at a portion of a trace after thousands or millions of steps since the beginning of a computation. Displaying or even recording all the steps of a computation can be very time consuming or even infeasible. TeaBag lets the user set breakpoints so that no step is displayed or recorded until the breakpoint. The user can set breakpoints in a program and choose to display only the steps that are executed on a subterm between breakpoints or until the subterm is in normal form.

3.5 Eager Evaluation

In a lazy language, the arguments of a function application $f\ t_1\ \ldots\ t_n$ are evaluated, as the name says, lazily. For example, if t_i is needed to compute the application of f, it will be evaluated to a constructor head normal form. Then steps may be executed on other terms unrelated to f and t_i, but it is possible that either t_i or some other argument of f will need to be further evaluated to compute the application of f. In short, the arguments of f may be computed in stages. This back and forth switching between arguments may occur repeatedly. It may be difficult for the programmer to understand the behavior of f until some of its arguments are sufficiently evaluated, but it is time consuming and distracting to interleave the evaluations of these arguments with the evaluation of other unrelated terms.

On demand eager evaluation is a feature that lets the programmer override the default lazy evaluation of a term. In this context, eager evaluation means evaluation to normal form. Any term displayed in a window can be interactively selected. By default, the result of eagerly evaluating a term is only displayed and does not replace the term. This lets the user see what the term evaluates to without changing the lazy behavior of the program. An option, allows the user to replace a term with its eagerly evaluated result. This is another device to compress the information displayed to the user. Obviously, an attempt to eagerly evaluate a term may result in a non-terminating computation even for a trace that terminates.

3.6 Runtime Debugging

TeaBag is not just merely a tracer. It is also a runtime debugger. TeaBag allows a programmer to see the rewriting and narrowing steps of a computation at runtime. The unique feature ofTeaBag's

runtime debugging environment is that it interacts with the tracer. The tracer will reflect the computation(s) the programmer activated during runtime. It will also compress in a single step the optional eager evaluation of a term. Choice Control, the feature described in Section 3.3, is available during runtime debugging and can effect what steps are generated for a trace. The runtime debugging environment is shown in figure 2.

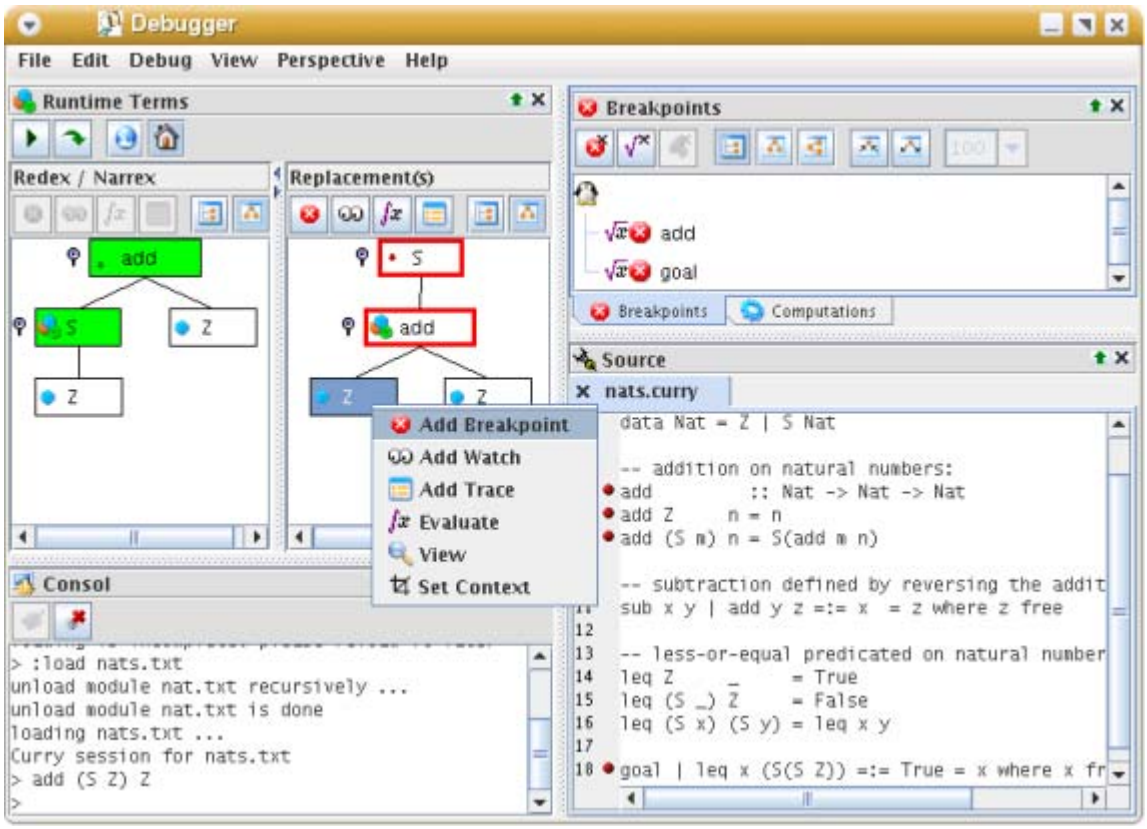


Figure 2. Runtime Debugging Environment

3.7 Breakpoints

TeaBag lets the programmer set a breakpoint on a function during runtime debugging. Whenever a rewrite rule defining that function is applied, the *FLVM* will be paused and the rule application will be displayed. This is convenient to debug non-terminating computations. TeaBag also lets the programmer attach a breakpoint to an *individual* term during runtime debugging. When that one term is replaced, the *FLVM* will be paused and the step from which the replacement originates will be displayed. This is convenient when the programmer does not understand when or why a term is evaluated. To the best of our knowledge this is the only debugger that allows breakpoints to be set on individual terms.

3.8 Highlighting

Highlighting uses colors, icons, and other visual clues as aids to understand the large amounts of displayed information.

Term Replacement When a rewriting or narrowing step is displayed both the redex pattern and created positions are highlighted. This information is intended to speed up the perception of how a rewriting or narrowing step changes a term. It also enables the viewer to determine which of the possibly many rewrite rules of a function has been applied in the step.

Variable Binding When a variable is bound, both the variable and its instantiation are highlighted. This makes it easier to detect that a step involves a binding and to determine both the variable being bound by the step and the step's substitution.

Source Code When available, the source code executed to perform a step is highlighted. Information about the source code is optionally added by the compiler to the generated bytecode. Providing source code highlighting makes it easy for users to correlate the rewriting or narrowing steps with their code. Being able to relate information displayed by the debugger to source code is considered important [38] since ultimately a bug in a program will be fixed by changing the source code.

Non-Deterministic Choice Selection Source Code Highlighting When the user selects a non-deterministic step to follow in the trace browser, the source code for that selection is highlighted if it is available. For example, in figure 4 the non-deterministic choice selection in the trace browser corresponds to the third rewrite rule for `solve` as highlighted in the source code.

4 Architecture

In this section, we give a few details on the architecture of our debugger. A significant aspect of our architecture is that the debugger interface is entirely separated from the *FLVM*. The debugger is implemented in Java. Java is a friendly, portable language with excellent graphical libraries. The *FLVM* is implemented in Java as well, but this may change in the future. The efficiency of the *FLVM* is obviously a concern and the size of its code is small, thus a conversion to a different language is feasible.

A potential problem of most debuggers is scalability. Generally, one must consider both large programs and programs that execute a large number of steps. In our case, one should also consider programs with a large degree of non-determinism. Often, scalability is in conflict with both providing detailed information and presenting information in a form visually rich, e.g., by means of colors, fonts, and options. We have chosen to provide detailed, visually rich information and have also implemented several features, described in Section 3.4, which should help in debugging realistic programs.

Top Level Architecture The debugger interface is decoupled from the *FLVM* by running both in separate processes and communicating over sockets. This allows the debugger to work with any virtual machine that implements the socket interface. Having the *FLVM* in a separate process makes it easy for the debugger to kill and restart it which is especially useful when the *FLVM* executes a non-terminating computation.

Debug Events The *FLVM* communicates with the debugger by sending it debug events over a socket. The debugger has a thread listening for these events. When data becomes available on this socket the debugger parsers the event and dispatches it to the event thread for processing.

Debug Commands The *FLVM* understands both user commands, such as "`:load`" and "`:quit`" to respectively load a module and terminate an execution, and debugging commands, such as the request to eagerly evaluate a term. User commands are given interactively to the command line interpreter. Debugging commands for the *FLVM* have a textual representation that could be typed in by a user. This allows the *FLVM* to redirect the socket of communication to standard input and run as if there was a user typing in the commands. When a program is executed under the debugger, the debugger acts as a proxy for the *FLVM* for all user input.

Tracing The information for a trace is written to a file during runtime. The trace browser reads data from that file to display the steps in the trace window. Since, the user can choose which terms to trace, not all steps of a computation are recorded to this file. Whenever a term is set to be traced, the *FLVM* creates a chain of responsibility structure [17] among its subterms via a listener. Then, when a subterm is replaced, it propagates this information up the chain of responsibility. Any term along this chain that has a trace set on it fires a trace step event to the debugger. When the debugger gets the trace step event, it writes the event to a file. In an attempt to make the changes to the *FLVM* as simple as possible the debugger handles the files associated with tracing. However, marshaling and unmarshaling the event is time consuming. One optimization we foresee is moving the file handling to the *FLVM* and having it write the trace steps directly rather than through the debugger.

To minimize file sizes only the first step of the file contains the entire term. Subsequent steps contain the difference from the previous term represented as a position and replacement. To get a step from the file the first term is parsed out. Then for each step up to the desired one the given position in the term is replaced with the replacement. Since this can be time consuming (it can take as long as the entire computation) the files are broken up so that they contain at most 50 steps.

The execution of a non-deterministic step fires a non-deterministic trace step event to the debugger. The debugger creates new traces for each of the possible replacements. A non-deterministic trace step is just a collection of traces.

Breakpoints Each time the *FLVM* replaces a term it checks if either the term or the symbol (a function) at the root of the term has a breakpoint set on it. If it does then the *FLVM* fires a breakpoint event to the debugger, suspends the thread that evaluates terms, and wakes up the thread that reads input from the user.

Eager Evaluation To perform eager evaluation of a term a new evaluation thread to work on a deep copy of the term to be evaluated is created. Creating the deep copy makes sure that there are no shared terms between the copy and any other term in the *FLVM*. This is necessary to preserve the lazy evaluation. When the newly created evaluation thread finishes evaluating the term to normal form an evaluation event is fired to the debugger. The *FLVM* holds on to the term to evaluate and its replacement until it knows if the user has selected to replace the term or not.

5 Example

The following example demonstrates the computation structure of TeaBag. Consider the *Dutch National Flag* program of Section 2 where lines 10-12 are replaced by:

```
10    solve flag | flag =:= blue:y ++ white:z
11               = solve (white:y ++ blue:z)
12               where y,z free
```

i.e., the prefix, `x`, of a blue-white pair of pebbles out of place has been forgotten. When `solve [white,red,blue,white]` is run using the above program the result is a failure. To find this bug we first generated a trace of `solve [white,red,blue,white]`. Part of the structure for this trace is shown in figure 3. We decided to follow the path through the computation structure that

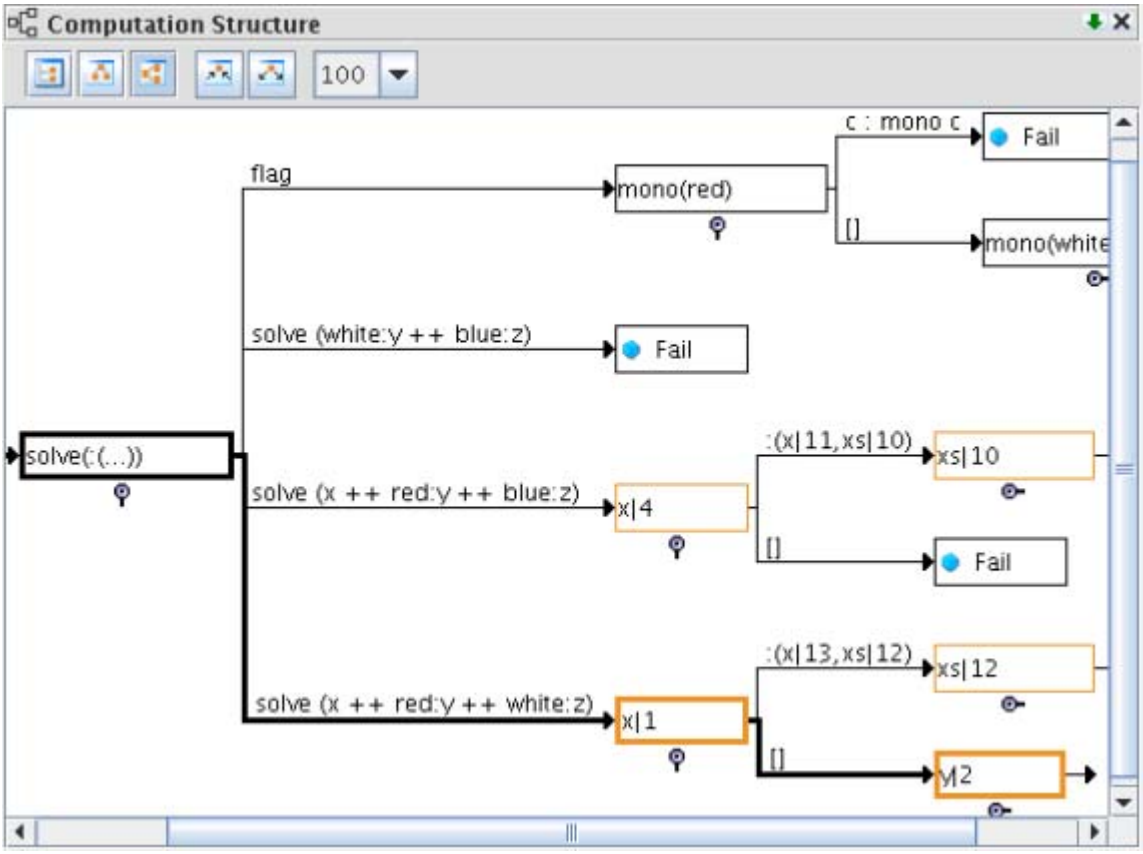


Figure 3. Computation Structure for Buggy Dutch National Flag Program

we thought should have led to a solution. Since the choices along this path should have led to a solution, examining the rewriting and narrowing steps on this path will tell us where the bug is located. In this example we realized that to get a solution the first and third rules of `solve` would need to be executed. The first rule should swap `red` and `white` and the third rule should swap `blue` and `white`. Either order of applying these rules should led to a solution. We arbitrarily decided to look at the path that is generated from applying the first rule and then applying the third rule. In order for the first rule to swap `red` and `white` it must find bindings for the free variables `x`, `y`, and `z` that satisfy `flag =:= x ++ white:y ++ red:z`. Since `flag` is `[white,red,blue,white]` binding `x` to `[]`, `y` to `[]`, and `z` to `[blue,white]` will work. We used this information to find the path thought the computation structure for applying the first rule.

There were two ways we could have found this trace path in the computation structure. We could have stepped through the trace in the trace browser one step at the time. Then when a non-deterministic step was made we would have been prompted to pick a branch to follow. By selecting the appropriate branches we would have followed the trace corresponding to the path in the computation structure that we wanted. Because there is more contextual information, this method works well when the correct choices at each non-deterministic step are difficult to determine. The second

way to find a trace path in the computation structure is to follow branches through the tree. Each node in the tree has one branch for each possible replacement. By knowing what the replacements should be, the correct branch at each node can be selected. Thus, this method works well when the correct choices at each non-deterministic step are known. We chose the second option since we knew which choices we wanted to examine. The first branch we followed was the one for swapping `red` and `white`. The next branch in the computation structure was for binding the variable `x`. One of the branches was for binding `x` to the empty list and the other branch for the non-empty list. The same was true of `y`. So we chose the empty list for both. We saw that there was no choice to be made for `z` since our choices for `x` and `y` forced `z` to be bound to `[blue,white]`. The next choice we had to make was for `solve(:(...))`.

At this point we needed to see what the term for the trace looked like to see if `red` and `white` were actually swapped like we thought they should have been. We were expecting that the term would be `solve [red,white,blue,white]`. To check this, we right clicked on the node in the computation structure for `solve(:(...))` and selected "*Move trace to this step*." This updated the trace browser to show the trace along the path we have chosen so far and to display the step for this choice as the current step. Figure 4 shows the trace at this point. The upper left corner is the trace step for picking a non-deterministic choice for `solve(:(...))`, the upper right corner is the source code with the code for the choice in the trace browser highlighted, and the lower panel shows the computation structure with the current path through the trace highlighted. Since we were interested in whether or not `red` and `white` were actually swapped we moved the trace back one step. This step showed that `red` and `white` had been swapped. The term for this step was `solve(red : [] ++ [white,blue,white])`.

Now `blue` and `white` must be swapped to get a solution. So we continued along the path we had followed so far choosing the path in the computation structure for swapping `blue` and `white`. We noticed immediately that this path leads to a failure as can be seen in figure 4. This caused us to think that the bug for this program was somewhere between choosing to swap `blue` and `white` and the failure. So we stepped through the trace one step at a time in the trace browser starting with the step for choosing to swap `blue` and `white`. After looking at five trace steps we noticed that for the condition to evaluate to a success, `[red,...]` must be equal to `[blue,...]`. Obviously, this can never happen since `red` can not be equal to `blue`. With this information we then went back in the trace to see why `blue` must be equal to `red`. We went back to the previous choice step to examine how the condition was created. Here we noticed that the condition is `[] ++ (red : [] ++ [white,blue,white]) =:= [blue,y,white,z]`. At this point we realized that there is no way for `red` to match anything on the right hand side since there is no free variable for it. So we added a free variable to this rule giving us the code presented in Section2.

We could have also found this bug by looking at the path in the computation structure that corresponds to applying the third rule of `solve` and then the first rule of `solve`. If we had chosen this path then we would have found the bug much faster since the path for applying the third rule of `solve` immediately leads to a failure as can be seen in figure 3.

6 Related Work

Functional logic languages borrow ideas from both functional and logic languages. Functional logic language debuggers are no different. They borrow ideas from debugging functional languages and from debugging logic languages. Four different debugging techniques have been applied to debugging functional logic languages. The first one is tracing. Tracing is a debugging technique used for debugging functional programs [14,15,38,42]. Tracers show each step of a computation to the pro-

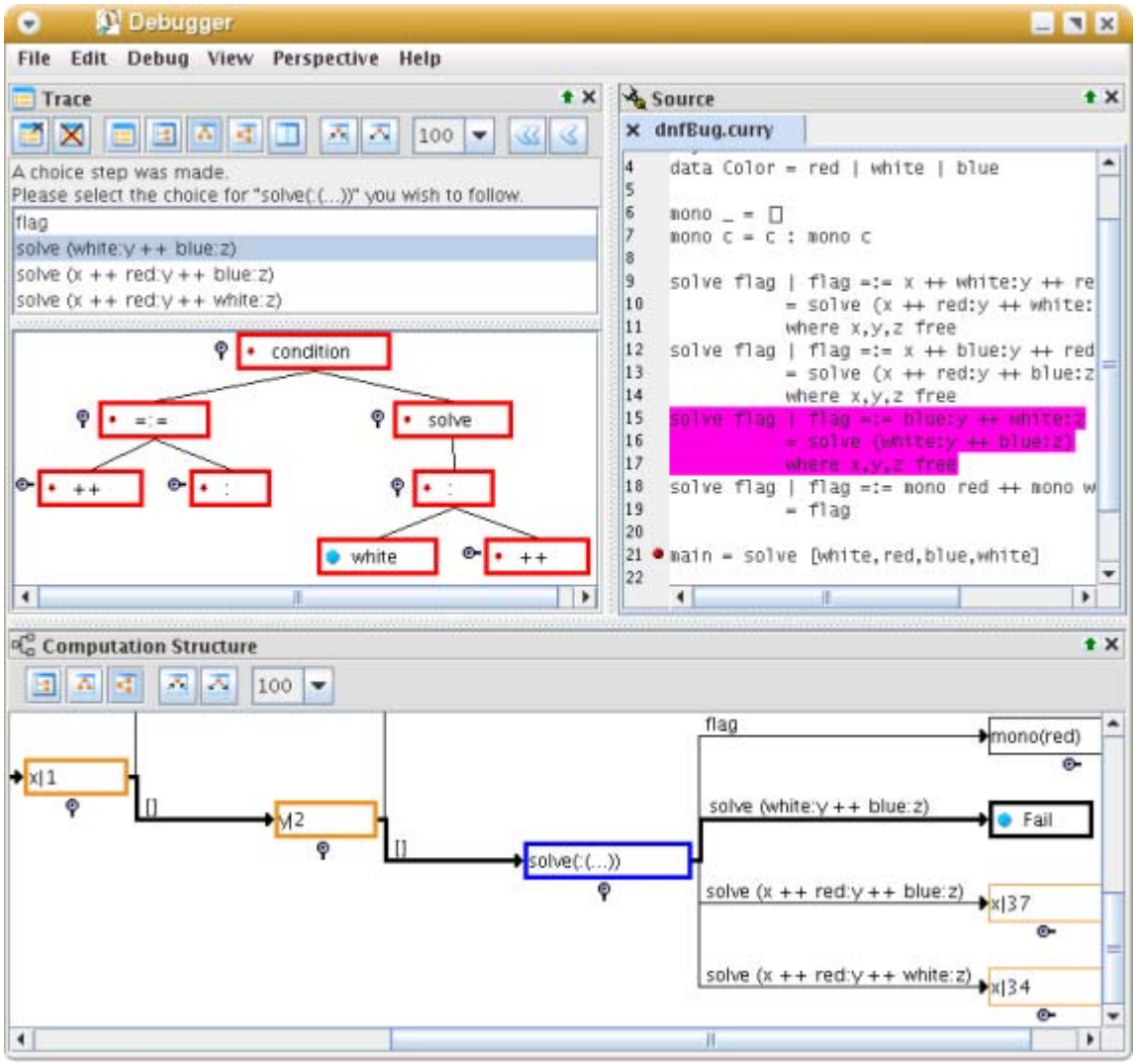


Figure 4. Trace of Buggy Dutch National Flag Program

grammer. Typically these steps are reductions, though they do not have to be. For example, tracing has been used to trace the redex trail of Haskell programs [38,14]. Likewise, CIDER [22] is a functional logic debugger for Curry based on tracing. Trace steps in CIDER are narrowing or rewriting steps. A specialized form of tracing called box-oriented debugging is the second type of functional logic debugging. Box-oriented debugging was first developed by Byrd [11] for debugging Prolog programs. Box-oriented debuggers trace goals in logic programs. Box-oriented debugging has been extended to functional logic languages by Hanus and Josephs [21] and by Arenas-Sánchez and Gil-Luezas [9]. The third type of functional logic debugger is observation debugging. This idea was first developed by Gill [18] for Haskell. It was latter incorporated into the Haskell tracer called Hat [40]. Observational debugging works by letting the programmer see the intermediate data structures that are passed between functions. Recently, Braßel, et al. extended this idea to functional logic languages by handling non-deterministic search, logical variables, concurrency, and constraints [10]. The final type of functional logic debugger is algorithmic. This idea was initially proposed by Shapiro [34] for debugging Prolog programs. The idea of algorithmic debuggers has been used in functional [31,32,37], logic [25,34,39], and functional logic debuggers [3,4,12,13,30]. Algorithmic

debuggers work by using the declarative semantics of the program. An oracle, typically the user, is asked questions about the intended meaning of their program in an automated way until the debugger can point out where the bug is located. Some examples of algorithmic debuggers for functional logic languages are Buggy [3], an accessory of the Münster compiler [12], and the debugger in the Toy system [13].

At first glance it may seems surprising that almost all functional logic language debuggers are algorithmic. Why haven't more debugging ideas from the functional and logic communities be explored in functional logic languages? We believe the reason for this is that algorithmic debuggers have been proven to work in both functional and logic languages so it is only natural that they would also work in functional logic languages. Most of the other debugging schemes for functional and logic languages have only been shown to work in their respective family of languages. Thus directly using one of those schemes for debugging functional logic languages will only debug "half" of the language. For example, CIDER [22] contains a tracer of rewriting and narrowing steps for debugging. This tracer works fine for tracing deterministic programs. However, it becomes difficult to use in non-deterministic programs. It shows the trace of non-determinism as a deterministic backtracking step which can be difficult to follow [11]. The tracer in CIDER is not as effective on non-deterministic programs as it is on deterministic programs.

For a functional logic debugger to be useful it has to be able to deal with both deterministic and non-deterministic features that real programs use [12]. We have applied this principle to tracing rewriting and narrowing steps in functional logic programs. We created a functional logic tracer that can be used to trace both deterministic and non-deterministic programs. To do this we borrowed the traditional tracing of reduction steps idea from functional programming [42] and combined it with the structure of the search space [33]. We believe that our approach is the first attempt to exploit this combination to trace the steps in a computation.

TeaBag is a debugger for Curry. There are three other debuggers for Curry: Münster [12], COOSy [10], and CIDER [22]. Münster is a compiler for Curry that contains a declarative debugger of wrong answers. TeaBag and Münster take different approaches to debugging Curry. Münster uses the declarative semantics of the program for debugging it. TeaBag uses the runtime narrowing and rewriting steps. Münster systematically asks the user questions until it can deduce where the bug is located. TeaBag, on the other hand, lets the user investigate how their program is being executed to find the bug. Given these differences Münster and TeaBag should be viewed as complementary, rather than competing, debuggers. Like Münster, COOSy takes a different approach to debugging from TeaBag. COOSy is an observational debugger. Thus COOSy lets the user view the values of expressions. To handle the non-deterministic aspects of functional logic programs COOSy extended Gill's observational debugging idea [18] to handle non-deterministic search, logical variables, concurrency, and constraints. Like TeaBag, COOSy extended a functional language debugging idea to handle all aspects of functional logic languages. Alternate non-deterministic choices in COOSy are shown in a group and the bindings of logic variables are displayed. TeaBag is much more like CIDER in that both of them use tracing for their debugger. CIDER is an IDE for Curry that contains a debugger which uses tracing to debug Curry. However, CIDER does not provide context hiding, highlighting, or a trace structure suitable for debugging non-deterministic programs. Thus CIDER is more difficult to use than TeaBag for debugging large programs and non-deterministic programs. While the sole focus of TeaBag is debugging, CIDER focuses on program development of which debugging is just one aspect. Thus CIDER includes analysis, editing, and compilation tools which are not in TeaBag.

TeaBag has a unique place in the current landscape of debuggers for functional logic languages. It is the only tracer of narrowing steps we are aware of that truly handles both deterministic and

non-deterministic features found in real functional logic programs. For more detailed information, source code, and to download TeaBag refer to [24].

7 Conclusion

We have presented, TeaBag, a debugger for functional logic computations. TeaBag has been developed as an accessory of the *FLVM*, a virtual machine intended for the execution ofCurry programs. A distinctive characteristic of this machine is its operational completeness. This means that the strategy for the execution of non-deterministic steps is concurrency, rather than backtracking. This strategy poses novel demands on a debugger.

Our debugger has both typical features of functional and logic debuggers, specifically features found in tracers and/or runtime debuggers, and novel features for displaying and managing non-determinism. In addition to standard features such as context elimination, highlighting and breakpoints on functions and terms, the user can view the non-deterministic steps of a computation and display only traces that make certain user-selected steps. To our knowledge, this is the first debugger with this capability.

References

1. H. Aït-Kaci. An overview of LIFE. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.
3. M. Alpuente and F. Correa. Buggy user's manual. http://www.dsic.upv.es/users/elp/buggy/.
4. M. Alpuente, F. Correa, and M. Falaschi. A debugging scheme for functional logic programs. In *Proc. of the 10th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, 2001.
5. M. Alpuente, F. Correa, and M. Falaschi. A declarative debugging scheme for functional programs. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2002)*, 2001.
6. S. Antoy. Optimal non-deterministic functional logic computations. In *6th Int'l Conf. on Algebraic and Logic Programming (ALP'97)*, volume 1298, pages 16–30, Southampton, UK, 9 1997. Springer LNCS.
7. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. Architecture of a virtual machine for functional logic computations, October 2003. Preliminary manuscript, available at `http://www.cs.pdx.edu/~antoy/homepage/publications.html`.
8. S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722, pages 335–350, Tsukuba, Japan, 11 1999. Springer LNCS.
9. P. Arenas-Sánchez and A. Gil-Luezas. A debugging model for lazy functional logic languages. Technical Report DIA 94/6, 1994.
10. B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing functional logic computations. In *Proceedings of the 6th Internationl Symposium on Practical Aspects of Declarative Languages*, Dallas, Texas, USA, June 2004. To appear.
11. L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tarnlund, editor, *Proceedings of the Logic Programming Workshop*, pages 127–138, 1980.
12. R. Caballero and W. Lux. Declarative debugging for encapsulated search. In Marco Comini and Moreno Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier, 2002.
13. R. Caballero and M. Rodríguez-Artalejo. A declarative debugging system for lazy functional logic programs. In Michael Hanus, editor, *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier, 2002.
14. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 176–193, Aachen, Germany, September 2000.
15. O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In Ricardo Pena and Thomas Arts, editors, *Implementation of Functional Languages: 14th International Workshop, IFL 2002*, LNCS 2670, pages 165–181, March 2003. Madrid, Spain, 16–18 September 2002.
16. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2001.
18. A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop. Technical report of the University of Nottingham.*, 2000.
19. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
20. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs`, 2003.
21. M. Hanus and B. Josephs. A debugging model for functional logic programs. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 28–43. Springer LNCS 714, 1993.
22. M. Hanus and J. Koj. An integrated development environment for declarative multi-paradigm programming. In *Proc. of the International Workshop on Logic Programming Environments (WLPE'01)*, pages 1–14, Paphos (Cyprus), 2001. Also available from the Computing Research Repository (CoRR) at `http://arXiv.org/abs/cs.PL/0111039`.
23. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
24. S. Johnson. TeaBag: A debugger for Curry. Master's thesis, Portland State University, expected September 2004. Available at `http://redstar.cs.pdx.edu/~stephenj/teabag/`.
25. G. Kókai, L. Harmath, and T. Gyimóthy. Algorithmic debugging and testing of Prolog programs. In *Proceedings of ICLP '97 The Fourteenth International Conference on Logic Programming, Eighth Workshop on Logic Programming Environments*, July 1997.
26. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.
27. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
28. W. Lux and H. Kuchen. An efficient abstract machine for Curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 — Annual meeting of the German Computer Science Society (GI)*, pages 390–399. Springer Verlag, 1999.
29. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
30. L. Naish and T. Barbour. A declarative debugger for a logical-functional language. In Graham Forsyth and Moonis Ali, editors, *Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems — Invited and Additional Papers*, volume 2, pages 91–99, Melbourne, June 1995. DSTO General Document 51.
31. H. Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, November 2001.
32. B. Pope. Buddha: A declarative debugger for Haskell. Technical report, University of Melbourne, 1998.
33. C. Schulte. Oz explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.
34. E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.
35. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.
36. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
37. J. Sparud and H. Nilsson. The architecture of a debugger for lazy functional langauges. In Mireille Ducassé, editor, *Proceedings of AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995. IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, Cedex, France.
38. J. Sparud and C. Runciman. Tracing lazy functional computations using Redex Trails. In *PLILP*, pages 291–308, 1997.
39. A. Thompson and L. Naish. A guide to the NU-Prolog Debugging Environment. Technical Report 96/38, Department of Computer Science, University of Melbourne, Melbourne, Australia, August 1997.
40. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: A new Hat. In *Proceedings of the Haskell Workshop 2001*, Firenze, Italy, 2001. Final version to appear in ENTCS.
41. D.H.D. Warren. Higher-order extensions to Prolog: Are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.
42. R. Watson. *Tracing Lazy Evaluation by Program Transformations*. PhD thesis, School of Multimedia and Information Technology, Southern Cross University, 1997.

Constraint Solving for Generating Glass-Box Test Cases

Christoph Lembeck[1], Rafael Caballero[2⋆], Roger A. Müller[1], and Herbert Kuchen[1]

[1] Department of Information Systems, University of Münster, Germany
christoph.lembeck@wi.uni-muenster.de
romu@wi.uni-muenster.de
kuchen@uni-muenster.de
[2] Dpto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain
rafa@sip.ucm.es

Abstract Glass-box testing tries to cover paths through the tested code, based on a given criterion such as def-use chain coverage. When generating glass-box test cases manually, the user is likely to overlook some def-use chains. Moreover it is difficult to find suitable test cases which cause certain def-use chains to be passed. We have developed a tool which automatically generates a system of test cases for a piece of Java byte code, which ensures that all def-use chains are covered. The tool consists of a symbolic Java virtual machine (SJVM) and a system of dedicated constraint solvers. The SJVM uses the constraint solvers in order to determine which branches the symbolic execution needs to consider. A backtracking mechanism is applied in case that several branches remain feasible. Thus we have applied implementation techniques known from functional logic and constraint programming to handle the considered applications problems.

1 Introduction

As quality standards for software systems are permanently increasing, software testing becomes a more and more important part of the software development process. Since software testing is time consuming and costly and tests have to be repeated several times, much effort has been invested in the automation of testing. As a result of these endeavors many regression test suites have been implemented. Even though these regression test suits assist a developer performing already known tests on his code, the generation of new test cases remains up to the user.

In the literature, testing is divided into black-box testing and glass-box testing. While black-box testing derives test cases from the specification of the user requirements, glass-box testing is based solely on the code and is hence adapted exactly to the way a given problem will be solved by the program. Here, we will focus on glass-box testing.

Glass-box testing is particularly suited to unit testing and to testing of algorithmically challenging parts of software systems. Unfortunately generating an exhaustive set of test cases based on a given coverage criterion as the def-use chain coverage is then challenging and error-prone, too.

In order to simplify glass-box testing, we have developed a tool that automates the generation of (mostly complete) sets of test cases satisfying a selectable testing criterion. Each test case consists of a set of constraints that describe the values for the input parameters leading to the pass through the test case in a general way, a concrete instance of values satisfying these constraints, a description of the path that will be taken during the runtime of the test, and the symbolic and the real result that will be expected at the end of the run. The testing criterion discussed in this paper will be the def-use chain coverage, which roughly ensures that all values defined somewhere in the program do not cause problems at places, where they are used. However the tool is also able to support other coverage criteria like statement coverage, branch coverage, or condition coverage and offers an open interface for the integration of further coverage criteria. Our tool is implemented in Java.

In order to identify the paths that may be followed by a real Java virtual machine [Su03] as precisely as possible, we have designed our tool as a symbolic Java virtual machine that processes

⋆ Author partially supported by the Spanish CICYT (project TIC2002-01167 'MELODIAS').

Java byte code. The difference of our SJVM to an original Java Virtual Machine [LY99] is that the value of a variable is not just a numerical value but an expression on some input variables. Prior to the symbolic execution, our tool analyzes the byte code in order to determine all def-use chains. During the symbolic execution, the SJVM generates at each branching instruction (e.g. a conditional jump) a constraint corresponding to the branch taken and propagates it to a constraint solver. If the overall system of constraints has no more solutions, or if all definitions and uses in the considered branch have been processed, the SJVM backtracks to the latest branching instruction and continues with an alternative branch, much like the Warren Abstract Machine known from Prolog [Wa83]. When the end of a path has been reached without any conflicts, it is up to the constraint solver to calculate one particular solution of the collected set of constraints, and to generate a corresponding test case. This can be added to a regression test suite.

As already mentioned, the main difficulty of the approach above is checking, whether the collected constraints remain solvable, or if they are already contradicting each other. For this purpose we have implemented a dedicated constraint solver, which is integrated into our SJVM and is connected to the symbolic execution engine by a special constraint solver manager that administrates the gathering of new constraints and facilitates an incremental growth and solving of the constraints (see Fig. 1). The SJVM has been presented in [ML03] and will not be explained in detail here. We will rather focus on the system of constraint solvers.

This paper is structured as follows. The tasks and the functionality of the constraint solver manager are described in Section 2. Section 3 explains how linear constraints will be handled, while Section 4 shows the treatment of non-linear constraints. Extracts of our experimental results and some known limitations of our approach will be discussed in Section 5. Related work and our conclusions can be found in Section 6 and 7, respectively.

2 The Constraint Solver Manager

The constraint solver manager (CSM) acts as an interface between the symbolic execution engine of the SJVM and the different constraint solvers we have implemented to handle different kinds of constraints. Since the constraints produced by the execution engine arrive incrementally at the CSM, it has to store each of them for further calculations. As the backtracking mechanism of our tool also guarantees that the latest constraints added to the system are the first that will be removed again, the CSM needs a constraint stack to maintain them.

Moreover, the CSM analyzes the constraints and transforms them to some kind of normal form. Additionally, it selects the most appropriate constraint solver for each system of constraints and distributes each constraint to the corresponding constraint solver, in case that the overall system of constraints consisted of several independent subsystems. At present only a small set of constraint solving algorithms are implemented, but the CSM has an easily implementable interface for the integration of further, more powerful algorithms.

One step of the normalization of constraints is that inequalities of the form $exp_1 \neq exp_2$ are transformed to an easier manageable system of inequalities $exp_1 < exp_2 \vee exp_1 > exp_2$. Moreover, fractions are removed by expanding and extending the constraints as shown in Figure2. Here it is important to notice that for each denominator it has to be examined, in which situations it will have the value 0, because then the occurrence of an `ArithmeticExcpetion` has to be considered (in Figure 2 this is represented by constraint (1)). Since multiplying inequalities with negative factors lets the comparison operator switch its direction, the original constraint has to be split into two new systems of constraints (line (2) and (3) of the example; all connected by $\vee$). These three systems

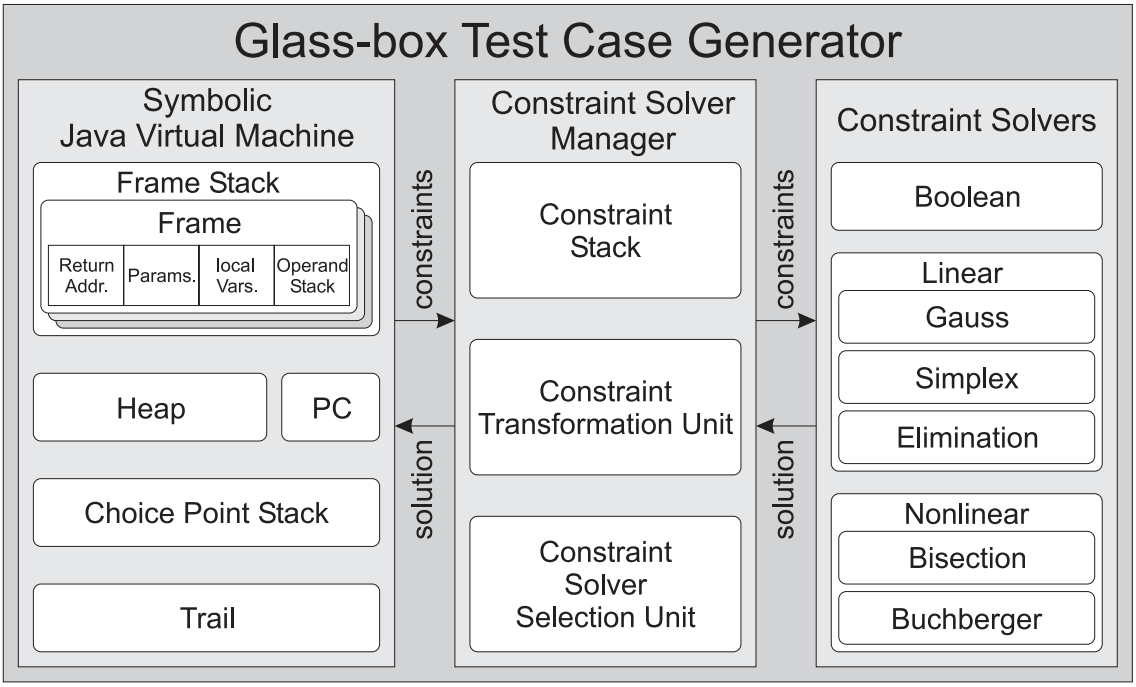


Figure 1. Symbolic Java Virtual Machine

$$\frac{1}{1+x} \geq a \quad \rightsquigarrow \quad \bigvee \begin{cases} 1+x=0 & (1) \\ 1 \geq a+ax \ \wedge \ 1+x>0 & (2) \\ 1 \leq a+ax \ \wedge \ 1+x<0 & (3) \end{cases}$$

Figure 2. Removal of fractions for $\frac{1}{1+x} \geq a$

will be successively considered using a backtracking mechanism. Thus, one system at a time will be passed on to an appropriate constraint solver.

Removing divisions also implies a consideration of the data types of the involved variables. While the types `float` and `double` introduce few difficulties, integer divisions (for the Java types `byte`, `char`, `short`, `int`, and `long`) have to be replaced in a special way. For example the equation $a/5 = 4$ containing the `double` variable a will be transformed to $a = 20.0$, while the same equation with an integer division has to be transformed to the set of constraints $a = 20 + r \wedge r \geq 0 \wedge r \leq 4$. Possible values for a will then be 20, 21, 22, 23, and 24.

The resulting system of purely polynomial equations and inequalities is then analyzed, in order to check whether they can be broken up into smaller, independently solvable systems. This is an interesting question, because maybe some parts of the system then can be handled by the more efficient linear constraint solvers, while only those parts that are really depending on nonlinear equations are passed on to the more complex nonlinear solvers. The next benefit of this approach concerns the ability of handling incrementally growing systems of constraints. When a new equation or inequality arrives, it has to be checked, which constraints are depending on the variables contained in the new equation or inequality and only for these constraints a new solution has to be computed. All the other constraints that are not depending on the variables of the new equation are left untouched and their solution calculated before can be reused. Because calculating solutions for complex systems of equations and inequalities can be very time consuming, the CSM firstly verifies if the existing solution of the previous calculations satisfies the new equation too. Only if

the old solution does not fit any longer, the computation of a new tuple of values will be initiated. Nevertheless the new constraint has to be added to the constraint stack in both cases.

The next and maybe most important job of the CSM is the analysis of the constraints and the choice of the best constraint solver for the given problem. For this purpose the constraints are divided up into several categories, each having a dedicated constraint solver.

Pure boolean constraints consisting only of the boolean constants `true` and `false`, boolean variables and the boolean Java operations `&` (and), `|` (or), `!` (not), `^` (xor), and the comparison operators == (equal) and `!=` (not equal) are solved by a simple backtracking mechanism known from (functional) logic programming languages like Prolog [SS94] or Curry [HK95].

The most important decision criterion of the arithmetic constraints is the linearity or nonlinearity of the system. Further criteria are the existence of weak ($\leq$, $\geq$) or strict ($<$, $>$) inequalities or of variables with integer data types (`byte`, `char`, `short`, `int`, or `long`).

Linear equations consisting only of floating-point data types (`float` and `double`) can easily be solved using Gaussian elimination, while inequalities can be solved using a variable elimination solver, parts of the simplex algorithm [BJ90] or a combination of them. The simplex algorithm can be enhanced by a branch-and-bound add-on or total enumeration, which enables the solution of linear (mixed) integer problems [BJ90,GN72]. These solvers are already included in the current version of our test tool and have proven to be reliable.

To explain the functionality of the whole test tool and especially of some of the mentioned solvers, the analysis of the short Java method `gcd` that calculates the greatest common divisor of two positive integer numbers will be used as a running example. The source code of the Java method is shown in Figure 3.

```
public static double gcd (double a, double b){
  while (a > 0){
    if (a < b){
      double h = a;
      a = b;
      b = h;
    }
    a = a % b;
  }
  return b;
}
```

Figure 3. Java source code of the greatest common divisor method

Since our tool operates on the byte code due to several reasons (e.g. efficiency and support of different programming languages (Java 1.4, Java 1.5, Pizza)) the source code of Figure3 has to be compiled to the form presented in Figure 4. Starting from that byte code our test tool offers the developer the choice of different testing criteria. In our example we decided to select the def-use chain coverage. Def-use chains [Be90] are defined as follows:

$$def(S) := \{X | \text{ instruction } S \text{ (re)defines } X\}$$
$$use(S) := \{X | \text{ instruction } S \text{ uses } X\}$$

Then $[X, S, S']$ is a ***def-use chain*** for variable X, if $X \in def(S) \cap use(S')$ but $X \notin def(S'')$ for all instructions S'' passed on some path from S to S'. We will indicate S and S' by the corresponding

```
 0 goto 15                    11 dload_0   // a
 1 dload_0   // a             12 dload_1   // b
 2 dload_1   // b             13 drem      // a%b
 3 dcmpg                      14 dstore_0 // a=a%b
 4 ifge 11   // a<b?          15 dload_0  // a
 5 dload_0   // a             16 dconst_0 // 0
 6 dstore_2  // h=a;          17 dcmpl
 7 dload_1   // b             18 ifgt 1   // a>0?
 8 dstore_0  // a=b;          19 dload_1  // b
 9 dload_2   // h             20 dreturn  // return b
10 dsotre_1  // b=h;
```

Figure 4. Java byte code of the greatest common divisor method

byte code line numbers. Taking into account that according to the Virtual Machine Specification [LY99] the SJVM puts the input parameters of the `gcd`-method on top of the methods local operand stack as variables 0 and 1 (which is annotated with VM in the def-use chains), the method contains the following set of def-use chains: [a,VM,1], [a,VM,5], [a,VM,11], [a,VM,15], [a,8,11], [a,14,1], [a,14,5], [a,14,11], [a,14,15], [b,VM,2], [b,VM,7], [b,VM,12], [b,VM,19], [b,10,2], [b,10,7], [b,10,12], [b,10,19], and [h,6,9].

To cover all these def-use chains several test cases have to be generated, each causing a different path through the code to be taken and requiring special values as input for the method. The computation of these values will be discussed in the following sections.

3 Solving Linear Constraints

Even though solving linear constraints seems to be one of the easier manageable problems during the generation of test cases and lots of algorithms concerning this subject have been discussed in the literature, many of the known algorithms may be applied just to special subsets of the problems that appear in practice. While Gaussian elimination is working fine on systems of linear equations, the first phase of the Two-Phase Simplex Algorithm (leading to an initial basic feasible solution) [BJ90] are utilized for systems of linear equations and weak inequalities ($\leq$, $\geq$). Although some variations of the simplex algorithm exists that allow e.g. the additional handling of negative variables, the treatment of strict inequalities ($<$, $>$) is not sufficiently supported. For this reason we added an additional, more flexible algorithm to our test tool based on the Fourier-Motzkin elimination procedure [DE73,Ap03]. This elimination procedure iteratively removes a selected variable by isolating it in each inequality, combining these new inequalities and replacing the whole system of inequalities by a new system that does not contain the selected variable. It has been shown that the new system has a solution if and only if the original system has a solution, too (the two systems are also called equisatisfiable). This approach will be repeated until a system of inequalities remains that contains a set of restrictions to only one variable, for which a solution can easily be found. Substituting the solution for the last variable into the previously build systems leads then iteratively to a complete solution of the whole original problem.

More detailed the approach works as follows. Firstly all equations have to be removed from the system of equations and inequalities. This can be done by isolating an arbitrary variable of the first equation that is contained in the system of constraints. This variable can now be substituted in all equations and inequations by the right hand side of the transformed equation without varying the solution space of the remaining variables. This step will be repeated until the constraints are

free from any equations. If the original equations contain both integer and noninteger variables it is recommendable to remove the noninteger variables first.

Then the m inequalities with n variables have to be transformed to a system of inequalities of the form shown in (1). The operator $\odot$ stands for one of the operators $\leq$ and $<$ in the following:

$$\begin{array}{c} a_{11}x_1 + \cdots + a_{1n}x_n \odot b_1 \\ \vdots \qquad\qquad \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n \odot b_m \end{array} \tag{1}$$

Choosing x_1 as the first variable to eliminate lets us split the set $I = \{1, ..., m\}$ of indices for the inequalities into $I = I_- \cup I_0 \cup I_+$:

$$\begin{array}{l} I_- = \{i \in I | a_{i1} < 0\} \\ I_0 = \{j \in I | a_{j1} = 0\} \\ I_+ = \{k \in I | a_{k1} > 0\} \end{array} \tag{2}$$

The classification of the inequalities into the sets I_- and I_+ assists us to transform the inequalities of system (1) into the form (3):

$$\begin{array}{l} a_{j1}^{-1}b_j - a_{j1}^{-1}a_{j2}x_2 - \cdots - a_{j1}^{-1}a_{jn}x_n \odot x_1 \quad \forall j \in I_- \\ x_1 \odot a_{i1}^{-1}b_i - a_{i1}^{-1}a_{i2}x_2 - \cdots - a_{i1}^{-1}a_{in}x_n \quad \forall i \in I_+ \end{array} \tag{3}$$

Combining each of the inequalities with the indices in I_- with each of the inequalities with indices in I_+ by the isolated variable x_1 using the transitivity of the $\leq$ and $<$ relations leads to the new system (4):

$$\begin{array}{ll} \sum_{j=2}^{n}(a_{k1}^{-1}a_{kj} - a_{l1}^{-1}a_{lj})x_j \odot a_{k1}^{-1}b_k - a_{l1}^{-1}b_l & \forall k \in I_+, l \in I_- \\ \sum_{j=2}^{n} a_{ij}x_j \odot b_i & \forall i \in I_0 \end{array} \tag{4}$$

It is obvious that the new system consists only of $n-1$ variables and $|I_0| + |I_-| \cdot |I_+|$ inequalities. This step has to be repeated until only one variable is left or some constraint becomes obviously contradictory. Here it is also recommendable to begin eliminating the noninteger variables so that after some eliminations a system of inequalities remains that does only contain integer variables[1]. Together with the integer equations noted before, the generated weak inequalities over integer variables may now be solved by the branch and bound extended simplex algorithm or total enumeration.

Although the number of inequalities grows in the worst case exponentially in the number of variables, in our application this is no problem, since it is usually no problem to find a variable for which one of the sets I_+ or I_- and thus the total number of inequalities in the next step become relatively small. Note that if I_- or I_+ are empty, it is trivial to find a solution. Additionally the number of input parameters of typical Java methods tends to be small.

Taking our example of the `gcd`-method described earlier, the Fourier-Motzkin elimination will be used by the CSM for situations, where the SJVM wants to know, what branches of the `if`-instruction in the byte code on line 4 are still reachable, after the "true branch" of the conditional jump instruction on line 18 has been taken positive. Here the simple inequalities $a > 0 \wedge a < b$ will have to be handled. Since this is a trivial problem for the elimination algorithm, we make the example a little bit more interesting by adding the conditions $a \leq 100$ and $b \leq 100$.

[1] This mixture of possibly strict and weak inequalities can then easily be transformed into a system of weak inequalities by subtracting the greatest common divisor of all coefficients and the right hand side of each inequality from their right hand sides.

The set of inequalities gathered so far by the CSM will then be:

$$\begin{array}{lll} a > 0 & \rightsquigarrow \quad -a < 0 & (1) \\ a < b & \rightsquigarrow a - b < 0 & (2) \\ a \leq 100 & \rightsquigarrow \quad a \leq 100 & (3) \\ b \leq 100 & \rightsquigarrow \quad b \leq 100 & (4) \end{array}$$

Selecting a as the next variable to eliminate lets us build the three sets of indices $L = \{1\}$, $I_0 = \{4\}$, and $I_+ = \{2, 3\}$. Preparing the inequalities (1), (2), and (3) for the later elimination of a leads to:

$$\begin{array}{l} 0 < a \\ \quad a < b \\ \quad a \leq 100 \end{array}$$

Combining them pairwise and adding the inequalities of I_0 produces the $1+1 \cdot 2 = 3$ inequalities with b as the only contained variable:

$$0 < b \;\; 0 < 100 \;\; b \leq 100$$

The constraint solver is now able to detect that there are no conflicting constraints and that the value of the variable b has to be selected from the interval $]0; 100]$. Selecting $b = 50$ randomly and inserting this value into the original system results in:

$$\begin{array}{ll} -a < 0 & a < 50 \\ a \leq 100 & 50 \leq 100 \end{array}$$

Thus the interval for the variable a is $]0; 50[$. The information, the CSM can pass on to the symbolic execution engine of the SJVM is that the considered path of the symbolic execution can still be reached and possible values for a test case leading to that path can e.g. be $a = 40$ and $b = 50$.

4 The Nonlinear Constraint Solver

For nonlinear constraints we offer a symbolic approach based on the Buchberger algorithm [BW93] [ML03] and a numerical approach. The Buchberger algorithm has the disadvantage that it is doubly exponential and that there is no systematic approach to find one concrete solution which could be used to generate a test case in case that there are infinitely many solutions. In the present paper we will focus on our numeric approach, a bisection algorithm, which successively divides the search space looking for solutions of the considered system of polynomial equations and inequalities.

Returning to our running example, the `gcd`-method, let us consider the constraints gathered when for the first three if-instructions of the byte code the true-branch was selected and for the following two if-instructions the conditions were assumed to be false. This leads to a test case covering the def-use chains [a,VM,1], [a,VM,11], [a,VM,15], [a,8,11], [a,14,1], [a,14,5], [a,14,15], [b,VM,2], [b,VM,7], [b,VM,12], [b,10,12], [b,10,19], and [t,6,9]. The result of the symbolic execution in that case is ($a \bmod b$) and the gathered constraints are:

$$\begin{array}{l} a > 0 \;\; a \bmod b > 0 \;\; b \bmod (a \bmod b) \leq 0 \\ a \geq b \;\; a \bmod b < b \end{array}$$

Because many algorithms can not handle constraints containing modulo operations directly, the disturbing $\bmod$-operations will be eliminated first by inserting additional variables:

$$\begin{array}{lll} a > 0 & x > 0 & y \leq 0 \\ a \geq b & a = mb + x & b = nx + y \\ m \geq 0 & x \geq 0 & x \leq b - 1 \\ n \geq 0 & y \geq 0 & y \leq x - 1 \end{array}$$

Here it is easy to see, that y has to be zero and that the whole problem is obviously nonlinear. Substituting b, the equation for the variable a will be transformed to $a = m \cdot n \cdot x + m \cdot y + x$, which is a polynomial equation of third degree and thus finding solutions for it is a nontrivial problem.

The constraints in our setting are conjunctions of *atomic polynomial constraints*. Disjunctions are not considered because the solver manager handles them via backtracking. Each atomic polynomial constraint has the form $p \lozenge 0$, where p is a polynomial and $\lozenge \in \{=, <\}$. Notice that for the further reasoning it is not necessary to include weak inequalities, since they can be decomposed in the disjunction of an equality and a strict inequality, and that constraints of the form $p > 0$ can be replaced by $-p < 0$.

After transforming the example we get:

$$\begin{array}{lll} -a < 0 & -n < 0 & b - nx = 0 \\ b - a < 0 & a - mb - x = 0 & x - b + 1 < 0 \\ -m < 0 & -x < 0 & 1 - x < 0 \end{array}$$

Each variable in the constraint has some predefined domain, represented by the different types used in Java for representing numbers (`byte`, `char`, `short`, `int`, `long`, `float`, `double`). Given a variable x we represent the set of values of its domain by $dom(x)$. In our example we would expect a, b, and x to be `double` values and, m and n `int` values.

Let $\overline{x}_n$ be the number of different variables in a constraint φ, $\varepsilon \in \mathbb{R}_+$, and $\overline{c}_n \in \mathbb{R}^n$. Then we say:

- $\overline{c}_n$ *ε-satisfies* an atomic constraint of the form $p < 0$ if $p(\overline{c}_n) < -\varepsilon$ and each $c_i \in dom(x_i)$ for $i = 1 \ldots n$.
- $\overline{c}_n$ *ε-satisfies* an atomic constraint of the form $p = 0$ if $-\varepsilon \leq p(\overline{c}_n) \leq \varepsilon$ and each $c_i \in dom(x_i)$ for $i = 1 \ldots n$.
- $\overline{c}_n$ *ε-satisfies* a constraint φ if it ε-satisfies all the atomic constraints in φ.
- $\overline{c}_n$ *satisfies* a constraint φ if it 0-satisfies all the atomic constraints in φ.

The constraint solver aims at finding some values $\overline{c}_n$ ε-satisfying the constraint φ for some small enough ε, $0 < \varepsilon < 1$. However, this problem is quite involved.

Fortunately, we are not interested in finding *all* the solutions of the system. For our purposes it is enough to find one solution, if such a solution exists, to obtain a test case covering the considered def-use chains. With this aim our solver uses a numerical technique based on a generalized *bisection algorithm* [KS99], which has proved to be reliable for isolating real solutions of multivariate polynomial systems. The algorithm, adapted to our requirements, considers some initial n-dimensional rectangle D, where the number n of variables in the given constraint φ, and some error bound ε,

and checks if there is any point $\overline{q}_n \in D$ which ε-satisfies φ. It can be summarized as follows:

Solve(D,φ,ε):

I = {D}
While I is not empty and no solution has been found
    Select a rectangle S of I.
    I = I - $\{S\}$
    if there exists $\overline{q}_n \in S$ such that $q_i \in dom(x_i)$ for $i = 1 \ldots n$ then
        if $\overline{q}_n$ ε-satisfies φ then a solution has been found ($\overline{q}_n$)
        else if mightBeSatisfied(S, φ, ε) then
            split S in smaller rectangles $\{S_1, \ldots, S_k\}$
            I = I $\cup \{S_1, \ldots, S_k\}$

As it can be seen in this description, the algorithm first checks whether there is any point $\overline{q}_n$ in the rectangle such that $q_i \in dom(x_i)$ for all $i = 1 \ldots n$. If such a point does not exist then the rectangle can be discarded. Otherwise one of these points is singled out to check if it ε-satisfies the constraint. In this case we have obviously found a solution. Otherwise the algorithm uses a test, represented by the function *mightBeSatisfied* to determine if the rectangle could still contain some point ε-satisfying φ. If this happens the rectangle is split into k smaller rectangles (in our current implementation $k = 3$), which will be considered in turn. The process continues until some solution is found or the set of rectangles is empty, meaning that no solution has been found. Both the termination property and the correctness of the algorithm depend on the test *mightBeSatisfied*, which:

1. Returns *no* if it can be ensured that actually there is no point in the rectangle that can ε-satisfy the constraint. In this way the correctness is ensured, since no solution can be skipped.
2. Returns *no* too, if the size of the rectangle S is smaller than our predefined ε in each dimension and if any $\overline{q}_n \in S$ with $q_i \in dom(x_i)$ for $i = 1 \ldots n$ does not ε-satisfy φ. This ensures the termination of the algorithm if no valid solution will be found within the given precision.

To meet the first requirement we must devise some test *cannotBeSatisfied(S,φ,ε)* that can ensure, in certain cases, that the constraint cannot be ε-satisfied in some rectangle S. Then obviously *mightBeSatisfied* can be defined as:

mightBeSatisfied(S,φ,ε):

if cannotBeSatisfied(S,φ,ε) $\Rightarrow$ then return *no*
else return *yes*

A constraint φ cannot be ε-satisfied if some of its atomic constraints $p \lozenge 0$ cannot be ε-satisfied. And this can be checked since p is a polynomial and hence a continuous function:

cannotBeSatisfied(S,$p \Diamond 0$*,*ε*)*:

Let $m, M \in \mathbb{R}^n$ be two numbers such that $m < p(\overline{q}_n) < M$ for all $\overline{q}_n \in S$
if $\Diamond$ is '=' and $m \cdot M > 0$ and $|m|, |M| > \varepsilon$ then return *yes*
else ($\Diamond$ is $'<'$)
    if $m > -\varepsilon$ then return *yes*
        else return *no*

i.e. a constraint of the form $p = 0$ cannot be ε-satisfied in S when the upper and the lower bounds of the polynomial have the same sign (condition $m \cdot M > 0$) and both are greater that ε in absolute value. The same occurs if the constraint is of the form $p < 0$ and the lower bound is greater than $-\varepsilon$. If neither of these cases occurs then the constraint still might be ε-satisfied in S. To determine the upper and the lower bounds of a polynomial p in the rectangle S we use the *Bernstein expansion* of the polynomial since this representation has the following well-known property [GG99]:

Let p be a polynomial with n variables and p_B its Bernstein expansion. Let m, M be the minimum and the maximum respectively of the coefficients of p_B. Then $m < p(\overline{q}_n) < M$ for all $\overline{q}_m \in [0,1]^n$, where $[0,1]^n$ represents the n-dimensional cube $\underbrace{[0,1] \times \cdots \times [0,1]}_{n}$

Then the steps to find the values m, M used in the definition of the function $cannotBeSatisfied(S, p \Diamond 0, \varepsilon)$ are the following:

1. Define a n-dimensional homotecy $h : [0,1]^n \to S$ converting $[0,1]^n$ into S.
2. Compute the Bernstein expansion p_B of the composition $p \cdot h$, which is also a polynomial.
3. Obtain the maximum coefficient of p_B as value M and the minimum coefficient as value m.

Checking that the lower (respectively the upper) bounds of $p \circ h$ in $[0,1]^n$ is the lower (respectively the upper) bounds of p in S is straightforward. Moreover this technique has the following nice property:

Let p be a polynomial with n variables and $\varepsilon \in \mathbb{R}$, $0 < \varepsilon < 1$. Then for small rectangles S, the composition $p \circ h$, with h an homotecy such that $h : [0,1]^n \to S$, verifies that the minimum m and the maximum M coefficients of its Bernstein expansion $(p \circ h)_B$ are such that $M - m < \varepsilon$.

This ensures the termination property of the algorithm, since every rectangle such that $M - m < \varepsilon$ either contains a solution or can be discarded. However our current implementation also includes a time-out to ensure that reaching this limit is not too costly. Of course if the time-out is triggered then the technique cannot ensure if there is any solution to the system. But in practice this occurs only with very complex constraints (i.e. polynomials of very high degree). For instance given the system at the beginning of the section, the solver gets a solution after a few milliseconds.

5 Experimental Results and Restrictions

As we have seen in the previous sections, our constraint solvers are able to handle linear and nonlinear constraints and combinations of them. Interestingly, many of the common algorithms and data structures known from the literature get by with simple boolean and linear constraint solvers and the nonlinear solvers will rarely be used.

Algorithm	Characteristics of constraints			
	linear	nonlinear	floating-point	integer
Ackermann	√			√
Binary search	√			√
Bubble Sort	√			√
Bresenham	√			√
Factorial	√			√
Gaussian Elimination	√			√
Greatest Common Divisor	√	√		√
Histogram	√			√
Dijkstra Shortest Path	√			√
Logarithm		√	√	
Matrix Multiplication	√			√
McCarthy	√			√
Pattern Matching	√			√
Sin		√	√	
Sqrt		√	√	
StoogeSort	√			√

Table 1. Characteristics of constraints for several algorithms

This phenomenon is caused in the fact that although the results of many algorithms can involve rather complex calculations, the decisions, which guide the control flow, are mostly very simple. Taking for example factorial: the results for the factorials of the first natural numbers are $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, and $5! = 120$. Admittedly the corresponding calculations are surely nonlinear, but a look at the definition $factorial(n) = \prod_{i=1}^{n} i$ shows that a simple loop over the natural numbers from 1 to n is sufficient. The constraints the CSM has to solve during test case generation for the factorial method are $1 \leq n$, $2 \leq n$, $3 \leq n$, etc., because only the value of the control variable i and the value of the input variable n are responsible for the termination of the loop, whereas the value of the variable i is known in each iteration.

A similar behavior can be observed for every algorithm that is based on simple loops or even several nested loops having dependent or independent termination conditions. Algorithms in this category are vector and matrix multiplication, several sorting algorithms such as bubble sort, naive string pattern matching algorithms, and even the recursive Ackermann function.

The constraints become a little bit more complicated, if methods are performing nonlinear computations and their results are used to decide on the termination of the algorithm. Then the nonlinear computations will become part of the constraints. This problem can often be found in numerical algorithms like the computation of different Taylor series to approximate trigonometric functions, logarithms, roots, or the exponential function. Here often the value of a previously calculated estimation is compared to the current result to check whether more approximations are needed.

At a first glance this may quickly lead to a big problem, since in some calculations the exponents of the symbolically calculated intermediate results grow by each iteration of the calculation. But in order to generate complete sets of test cases for simple testing criteria like the commonly used def-use chain coverage we typically require only a small number of passes through loops. Thus the complexity of the constraints we gathered was mostly relatively small and hence manageable in a short amount of time. In particular our tool is able to generate the complete sets of test cases regarding the def-use chain coverage for the algorithms mentioned in Table1 in a few seconds. This table shows what kinds of constraints occur in the example applications we have considered.

Although the prototype of our test case generator has proven to be applicable to many algorithms, there are still a few problems we have to solve. For instance, our tool does not yet offer a

constraint solver that can manage constraints containing bit operations corresponding to the byte code instructions `ishl`, `ishr`, `iushr`, `lshl`, `lshr`, and `lushr`, which are produced by Java's shift operators `<<`, `>>`, and `<<<` on expressions of the types `int` and `long`. Since these operations are lacking in the usual algebra, workarounds have to be invented to deal with them. Fortunately, they rarely occur in application software and are typically only found in system software such as device drivers.

As mentioned already, very few iterations of each loop are typically sufficient to cover all def-use-chains. In cases where the number of iterations or recursive calls is very large, e.g. since it has been fixed by a large constant (note that this is bad programming style), the symbolic computation might take too long. Pragmatically, our tool will stop with a corresponding error message in that case. Note that due to the halting problem, it is in general impossible to determine the number of required iterations.

On the other hand our approach is vulnerable to rounding errors that come along with calculations on floating point values. Since our computations are not identical to that of the concrete JVM, the rounding errors in our tool and in the concrete JVM computation may differ. Note that it is equally bad to be less precise that the concrete JVM computation than to be more precise. The different rounding behavior may cause the symbolic and concrete computation to diverge, which would affect the correctness of our approach. Thus after generating the test cases, we will compare the paths taken by the symbolic and concrete computations. If they diverge, we will (try to) adapt the responsible input parameter, say from 3.9999 to 4.0. In case, where this does not work, our tool has to give up with a corresponding error message. After all, our tool tries to be helpful in almost all cases. In the few remaining cases, the user has to find suitable test cases on his own. In particular, this might happen in numerically instable calculations, which should be avoided anyway. In any case, we can guarantee that only correct test cases are generated. To summarize, our tool is correct but not necessarily complete (among others due to the halting problem).

As we already mentioned, we actually have only a prototypically implementation of our test tool and some features are not entirely realized. E.g. the symbolic execution engine of our tool is not yet able to handle all of the features of the Java language. First of all we have to consider the multithreading capability in this context, which may cause a nondeterministic behavior of Java programs and thus makes it hard to generate test cases that satisfy a certain coverage criterion in a real virtual machine. Secondly it is allowed to integrate functions and applications into Java programs that are not written in the Java language by using the Java Native Interface [Li99]. Java methods based on native method calls or methods invoking those native methods are at the moment not analyzable by our test tool.

6 Related Work

Many approaches to software testing have been proposed (see e.g. [Ed99] for an overview), but only a few can directly be related to our approach.

Gupta, Mathur and Soffa [GM00] have proposed an approach that uses a branch selection algorithm that generates input data which exercises a selected branch in a program. However their approach features no virtual machine and is strictly numerical. Gotlieb, Botella and Rueher [GB98] have proposed a limited constraint-based approach for a sub-set of the C-language that can identify paths that cover every instruction in the program. Their approach featured neither an exchangeable testing criterion, nor did the test tool contain several, automatically chosen constraint solvers. Korel [Ko96] has presented an approach to generate test data by performing a data dependence analysis in Turbo Pascal programs and using minimization techniques to discover suitable input values. The

constraint-based approach by Offutt and DeMillo [DO91] is based on their unusual mutation analysis test criterion, but comprehends just a naive constraint solver. An early approach to test-data generation was proposed by Ramamoorthy, Ho, and Chen [RH76], who transformed Fortran-code to a basic form and tried to solve the constraints describing the sought paths through the program by forward substitution. Finally, Lapierre et al. [LM99] implemented a tool that tried to solve the path-describing constraints by using mixed-integer linear programming for C programs.

7 Conclusions and Future Work

In this paper we have presented a tool for the automatized generation of glass-box test cases for Java methods using a combination of a symbolic Java virtual machine and a dedicated constraint solver. We pointed out that a simple usage of already known constraint solvers or constraint solving algorithms is not sufficient for the considered application, since the constraints that have to be managed occur not in normalized forms but have to be transformed by a dedicated constraint solver manager before passing them to the various solvers. We also discussed the influence of the different Java primitive types on the way, the constraints will be treated. We have additionally shown how the constraint solver manager handles incrementally arriving constraints.

Our system of constraint solvers contains in particular a solver using the simplex algorithm, a Fourier-Motzkin elimination solver for linear constraints, and a bisection solver for nonlinear ones. We are not aware of any other test case generator, which provides this powerful combination of solvers.

The experiences with our prototype have shown a few possible enhancements and some topics that will have to be improved in future releases, e.g. the handling of threads and the integration of native method calls.

References

Ap03. K.R. Apt. Principle of Constraint Programming. Cambridge University Press, Cambridge, UK, 2003.

Be90. B. Beizer. Software Testing Techniques. Van Nostr. Reinhold, 1990.

BJ90. M.S. Bazaraa, J.J. Jarvis, H.D. Sherali. Linear Programming and Network Flows. John Wiley & Sons, New York, NY, 1990.

BS90. M.S. Bazaraa, H.D. Sherali, C.M. Shetty. Nonlinear Programming, Theory and Algorithms. John Wiley & Sons, New York, NY, 1993.

Bu85. B. Buchberger. Grobner Bases: An Algorithmic Method in Polynomial Ideal Theory. *In Multidimensional Systems Theory*, D. Reidel Publishing Company, Dordrecht, Holland, 1985.

BW93. T. Becker, V. Weispfenning. Gröbner Bases, A Computational Approach to Computer Algebra. Springer, New York, NY, 1993.

DE73. G.D. Danzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288-297, 1973.

De03. R. Dechter. Constraint Processing. Morgan Kaufmann Publishers, San Francisco, 2003.

DO91. R.A. DeMillo and A.J. Offutt. Constraint-Based Automatic Test Data Generation. *In IEEE Transactions on Software Engineering, Vol.17, No.9*, September 1991.

Ed99. J. Edvardsson. A survey on Automatic Test Data Generation. *In Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21-28. ECSEL, October 1999.

GB98. A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation using Constraint Solving Techniques. *In ISSTA '98, Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, March 1998.

GG99. J. Garloff and B. Graf. Solving Strict Polynomial Inequalities by Bernstein Expansion *I*n The Use of Symbolic Methods in Control System Analysis and Design, N. Munro, Ed., The Institution of Electrical Engineers (IEE), London, pp. 339-352 (1999).

GM00. N. Gupta, A.P. Mathur, and M.L.Soffa. Generating Test Data for Branch Coverage. 15th IEEE International Conference on Automated Software Engineering (ASE'00), 2000.

GN72. R.S. Garfinkel, G.L. Nehmhauser. Integer Programming. John Wiley & Sons, New York, NY, 1972.

HK95. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. *In Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, 1995.

Ko96. B. Korel. Automated Test Data Generation for Programs with Procedures. *In Proceedings of the 1996 International Symposium on Software Testing and Analysis, San Diego, CA, USA. Software Engineering Notes 21(3)*, May 1996.

KS99. T. Kutsia and J. Schicho. Numerical Solving of Constraints of Multivariate Polynomial Strict Inequalities. RISC technical report 99-31. Available at: *ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1999/99-31.ps.gz.*

Li99. S. Liang. The Java Native Interface, Programmer's Guide and Specification. Addison-Wesley, Reading, Massachusetts, 1999.

LG97. Q. Li, Y. Guo, T. Ida, J. Darlington. The minimised geometric Buchberger algorithm: an optimal algebraic algorithm for integer programming. *International Conference on Symbolic and Algebraic Computation.* 331 - 338, 1997.

LM99. S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem, and P. Tonella. Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees. Int. Conf. on Software Maintenance 1999.

LY99. T. Lindholm and F. Yellin. The Java(TM) Virtual Machine Specification, Second Edition. Addison-Wesley, 1999.

ML03. R.A. Müller, C. Lembeck, and H. Kuchen. GlassTT - A Symbolic Java Virtual Machine using Constraint Solving Techniques for Glass-Box Test Case Generation, Technical Report 102, University of Münster, 2003.

RH76. C.V. Ramamoorthy, S.-B.F. Ho, and W.T. Chen. On the Automated Generation of Program Test Data. *In IEEE Transactions on Software Engineering, Vol. SE-2, No.4*, December 1976.

SS94. L. Sterling and E. Shapiro. The Art of Prolog, Second Edition. The MIT Press, 1994.

Su03. Sun Microsystems. Java 2 Platform, 2003. http://java.sun.com/j2se/.

Wa83. D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

A logical approach to the verification of functional-logic programs

José Miguel Cleva, Javier Leach and Francisco J. López-Fraguas *

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
jcleva@sip.ucm.es, fraguas@sip.ucm.es, leach@sip.ucm.es

Abstract. We address in this paper the question of how to verify properties of functional logic programs like those of Curry or Toy. The main problem to face is that equational reasoning is not valid for this purpose, due to the possible presence of non-deterministic functions with call-time choice semantics. We develop some logical conceptual tools providing sound reasoning mechanisms for such kind of programs, in particular for proving properties valid in the initial model of a program. We show how *CRWL*, a well known logical framework for functional logic programming, can be easily mapped into logic programming, and we use this mapping as a starting point of our work. We explore then how to prove properties of the resulting logic programming translation by means of different existing interactive proof assistants, and give some initial proposals trying to overcome the limitations of the approach, both in terms of efficiency and theoretical strength.

1 Introduction

One distinguished feature of modern functional logic languages like Curry [18] or Toy [20] is that programs are constructor based rewrite systems allowed to be non-terminating and non-confluent. Semantically this leads to the presence of non-strict and non-deterministic functions, which have been shown quite useful for practical declarative programming.

However, non-determinism makes equational reasoning non valid for reasoning about programs. The *CRWL* framework [13,14] - which is the theoretical basis of our work - gives a well-established alternative logic for functional logic programming (FLP). In *CRWL* the semantics of a program is given by its possible reductions, expressed by means of a reducibility relation $e \to t$ between evaluable expressions and constructor terms, which are the sensible kind of result of computations. CRWL provides a proof calculus prescribing which reduction statements $e \to t$ hold for a given program[1]. Programs have initial models, which are commonly accepted as the natural candidates to be intended models of programs.

CRWL has been extended with success to cope with many other features relevant to productive programming: HO, objects, subsorts, algebraic datatypes, constraints and failure. See [29] for a recent survey of the *CRWL*-approach to FLP. Here we restrict ourselves to first order programs.

Verification of properties of logic and functional programs have been frequently studied [28,27,17]. We do not know of many results in the FLP setting. The work of Padawitz [24,25] in equational logic programming constitute a serious effort, both at the theoretical and the practical level. In Padawitz functions are deterministic and with strict semantics. There is some other work contemplating the issue of FLP program properties from a specific point of view. This includes different topics about declarative debugging [7,8,1], abstract interpretation [6] or abstract diagnosis [2].

* The authors are partially supported by the Spanish project TIC2002-01167 'MELODIAS'.

[1] CRWL considers also a different kind of semantic statements, called joinability statements, which are useful for a good treatment of strict equality, a matter which we do not consider here.

The goal of our paper is to develop a logical basis from which quite general properties of FLP programs (like those of Curry or Toy) can be formulated and proved. The main lines of our approach can be summarized in advance as follows:

- Programs are *CRWL*-programs and the properties of interest are those valid in the initial model of a given program P, expressed as first order logic (FOL) formulas with reduction ($\rightarrow$) as relation symbol.
- The *CRWL*-semantics of P is expressed by means of a FOL theory, which actually is a logic program P_L, whose least model corresponds closely to the *CRWL*-initial model of P.
- We can prove properties valid in those models by FOL deduction from a FOL theory consisting of the completion of P_L extended with inductive axioms. The set of provable valid properties can be enhanced by refining this theory, in particular by embedding in it some meta-theory about *CRWL*-derivations.

The remainder of the paper is organized as follows. The next section presents some preliminaries about *CRWL*. In section 3 we draw a parallel FOL theory P_L – a logic program indeed – for any given *CRWL*-program P such that *CRWL*-deducibility from P corresponds to FOL-logical consequence from P_L. In section 4, in order to prove properties of the initial model of a sample *CRWL*-program P, we translate the inductive extension of the completion of P_L into several existing interactive proof assistants. In section 5 we introduce a variant of the logic program P_L emulating *CRWL*, where the derivation trees for statements $e \rightarrow t$ are explicit. Finally, section 6 summarizes some conclusions. Due to lack of space, proofs are omitted.

2 Preliminaries: *CRWL* programs and their logical semantics

We assume a signature $\Sigma = DC_\Sigma \cup FS_\Sigma$ where $DC_\Sigma = \bigcup_{n \in \mathbb{N}} DC^n_\Sigma$ is a set of *constructor* symbols and $FS_\Sigma = \bigcup_{n \in \mathbb{N}} FS^n_\Sigma$ is a set of *function* symbols, all of them with associated arity and such that $DC_\Sigma \cap FS_\Sigma = \emptyset$. We also assume a countable set $\mathcal{V}$ of *variable* symbols. We write Exp_Σ for the set of (total) *expressions* built up with Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset $CTerm_\Sigma$ of (total) constructor terms or (total) *c-terms*, which only make use of DC_Σ and $\mathcal{V}$. The subindex Σ will usually be omitted. Expressions intend to represent possibly reducible expressions, while c-terms represent not further reducible data values.

The signature $\Sigma_\perp$ results of extending Σ with the new constant (0-arity constructor) $\perp$, that plays the role of the undefined value. The sets $Exp_\perp$ and $CTerm_\perp$ of (partial) expressions and (partial) c-terms respectively are built up using $\Sigma_\perp$. Partial c-terms represent the result of partially evaluated expressions; thus, they can be seen as approximations to the value of expressions. A partial c-term is called *ground* if it does not contain any variable.

As usual notation we will write $X, Y, Z, ...$ for variables, c, d for constructor symbols, f, g for functions, e for expressions and s, t for c-terms. In all cases, primes (') and subindices can be used. Expressions can be compared by the *approximation ordering* $\sqsubseteq$, defined as the least partial ordering verifying: $\perp \sqsubseteq e$ and $e_1 \sqsubseteq e'_1 \wedge \ldots \wedge e_n \sqsubseteq e'_n \Rightarrow h(e_1, \ldots, e_n) \sqsubseteq h(e'_1, \ldots, e'_n)$, for $h \in DC^n \cup FS^n$.

We will use the sets of substitutions $CSubst = \{\theta : \mathcal{V} \rightarrow CTerm\}$ and $CSubst_\perp = \{\theta : \mathcal{V} \rightarrow CTerm_\perp\}$. We write $e\theta$ for the result of applying θ to e.

In the next sections we will need some familiar notions about first order logic and logic programming (see e.g. [11,5] for standard references). We will use $\varphi, \varphi', \ldots$ for FOL-formulas

Table 1. Rules for *CRWL*-provability

(BT) Bottom	$\dfrac{}{e \to \bot}$ for $e \in Exp_\bot$
(DC) Decomposition	$\dfrac{e_1 \to t_1 \ldots e_n \to t_n}{c(e_1,...,e_n) \to c(t_1,...,t_n)}$ $c \in DC^n, \quad t_i \in CTerm_\bot, \quad n \geq 0$
(FR) Function reduction	$\dfrac{e_1 \to t_1 \ldots e_n \to t_n \quad e \to t}{f(e_1,...,e_n) \to t}$ if $t \not\equiv \bot, f(t_1,...,t_n) \to e \in [P]_\bot$

and the standard notation $T \models \varphi$, $I \models \varphi$ for logical consequence from a FOL theory (i.e., set of formulas) T and validity in a given interpretation I. We write also $I \models T$ to indicate that I is a model of T.

2.1 The Proof Calculus for CRWL

In [13,14] programs are made of conditional rules, where conditions are conjunctions of joinability (or strict equality) conditions. Since we are not dealing here with strict equality as a specific, built-in construct, and it is known [4,30] that in programs like ours (following constructor discipline) conditions can be replaced by semantically equivalent *if-then* expressions, we consider here programs with non-conditional rules.

So, in this work a *CRWL-program* $\mathcal{P}$ is a finite set of rewrite rules of the form $f(t_1,...,t_n) \to e$ where $f \in FS^n$, $(t_1,...,t_n)$ is a linear tuple (each variable in it occurs only once) of c-terms, and e is an expression. Notice that e can contain variables not occurring in $f(t_1,...,t_n)$. We write $\mathcal{P}_f$ for the set of defining rules of f in $\mathcal{P}$.

From a given program $\mathcal{P}$, the proof calculus for *CRWL* can derive *reduction* or *approximation statements* of the form $e \to t$, with $e \in Exp_\bot$ and $t \in CTerm_\bot$. The intended meaning of such statement is that e can be reduced to t, where reduction may be done by applying rewriting rules of $\mathcal{P}$ or by replacing subterms of e by $\bot$. If $e \to t$ can be derived, t represents one of the possible values of the denotation of e.

When using a function rule R to derive statements, the calculus uses the so called *c-instances* of R, defined as $[R]_\bot = \{R\theta | \theta \in CSubst_\bot\}$. We write$[P]_\bot$ for the set of c-instances of all the rules of a program P. Parameter passing in function calls are expressed by means of these c-instances in the proof calculus.

Table 1 shows the proof calculus for *CRWL*. We write $\mathcal{P} \vdash_{CRWL} \varphi$ for expressing that the statement φ is provable from the program $\mathcal{P}$ with respect to this calculus. The rule (FR) allows to use c-instances of program rules to prove approximations. Thesen c-instances may contain $\bot$ and by rule (BT) any expression can be reduced to $\bot$. This reflects a non-strict semantics, allowing non-terminating programs to be meaningful.

A distinguished feature of *CRWL* (shared by concrete systems like Curry or Toy) is that programs can be non-confluent, defining thus *non-deterministic functions*. As a typical example, consider the program (called *Coin* for future references) in Fig.1, which assumes the constructors 0 and s for natural numbers.

Notice that *coin* is a non-deterministic function, for which the previous calculus can derive the statements $coin \to 0$ and $coin \to s(0)$. The use of c-instances in rule (FR) instead of general instances corresponds to *call time choice* semantics for non-determinism [19,13,14]). In the example, it is possible to build a *CRWL*-proof for $double(coin) \to 0$ and

$0 + Y \to Y$	$coin \to 0$
$s(X) + Y \to s(X + Y)$	$coin \to s(0)$
$double(X) \to X + X$	

Fig. 1. CRWL sample program *Coin*

also for $double(coin) \to s(s(0))$, but not for $double(coin) \to s(0)$. This semantic choice is not a caprice of *CRWL*. Call-time choice is related to *sharing*, a well known operational technique considered essential for the effective implementation of lazy funcional languages like Haskell. Existing FLP languages like Curry or Toy also use sharing and call-time choice semantics. The above described behaviour for the reduction of $double(coin)$ corresponds exactly with what happens in those systems. Run-time choice, an alternative semantics for non-determinism with which $double(coin)$ can be reduced also to $s(0)$ is investigated for the FLP setting in [3].

From the point of view of verifying properties of FLP programs, non-determinism and call-time choice semantics have the unpleasant consequence that equational reasoning is not valid for CRWL-programs. In the previous example, if the rules for *coin* were understood as the equalities $coin = 0$ and $coin = s(0)$, then we could deduce $0 = s(0)$, which is not intended. Call-time choice implies that not only equational reasoning, but also ordinary rewriting is invalid since, from the point of view of rewriting, the rule $double(X) \to X + X$ should be applicable to *any* X, and not only to c-terms. Hence, we would have $double(coin) \to coin + coin$, and from this, $double(coin) \to s(0)$, which is not valid with call-time choice. A remark about the *CRWL*-calculus presented here, with respect to the original in [13,14]: in addition to the above mentioned elimination of joinability statements, we have also dropped the so called *restricted reflexivity rule*:

$$\text{(RR)}\ \frac{}{X \to X} \qquad X \in \mathcal{V}$$

At the end of this section we argue the advantages of having done so. But we first discuss the relation between both calculi. Inside this discussion, let us call *CRWL* the calculus of table 1, and $CRWL_{RR}$ the proof calculus with the rule (RR). Within $CRWL_{RR}$ we can prove, for instance, $0 + X \to X$ and all its c-instances while in *CRWL* only the ground c-instances $0 + t \to t$, for any ground partial c-term t. The next result precises the relation between both calculi:

Proposition 1. *Let P be a CRWL-program. Then:*
(i) $P \vdash_{CRWL} e \to t \Rightarrow P \vdash_{CRWL_{RR}} e \to t$
(ii) $P \vdash_{CRWL_{RR}} e \to t \Rightarrow P \vdash_{CRWL} e' \to t'$*, for all ground c-instances* $e' \to t'$ *of* $e \to t$

With respect to models the situation is the following. In $CRWL_{RR}$ Herbrand models of programs have as support a Herbrand universe of partial c-terms with variables [13,14], and every program P has a least Herbrand model M_{RR_P} which is technically a *free* model, while with *CRWL* as it has been presented here we must use the ordinary Herbrand universe of ground c-terms, and it can be shown that every program P has a least Herbrand model M_P which is an initial model. Least models verify:

Proposition 2. *For any CRWL-program P,*
(i) $P \vdash_{CRWL_{RR}} e \to t \Leftrightarrow M_{RR_P} \models e \to t$
(ii) $P \vdash_{CRWL} e \to t \Leftrightarrow M_P \models e \to t$*, for any ground* $e \to t$
(iii) $M_{RR_P} \models e \to t \Rightarrow M_P \models \forall(e \to t)$*, where* $\forall\varphi$ *indicates the universal closure of* φ

We believe that, in some sense, M_P is more natural than M_{RR_P} as intended model whose properties are to be formally verified. For instance, in the *Coin* example above, the property $\varphi \equiv \forall E, T.(E \to T \Rightarrow E + 0 \to T)$, which is intuitively a true property about addition and reduction, is in fact valid in M_P, but not in M_{RR_P}, because with RR we can *CRWL*-prove $X \to X$ (and then $M_{RR_P} \models X \to X$), but not $X + 0 \to X$ (and then $M_{RR_P} \not\models X + 0 \to X$).

3 CRWL as a logic program

In this section we will map *CRWL* into first order logic (FOL). We assume the reader is familiar with standard notions of FOL (see e.g.[11]) and logic programming (see e.g. [5]). We want to associate to a given CRWL-program P a FOL theory P_L such that *CRWL*-deducibility from P corresponds to FOL-logical consequence from P_L. The theory P_L will be indeed a logic program, and we will use this logic program to prove properties of the original CRWL program as stated by the results given in this section.

Consider a *CRWL* program P with signature $\Sigma = DC \cup FS$. The logic program P_L associated with P is made of the following clauses (written as implications $l \Leftarrow C_1 \wedge \ldots \wedge C_n$, $n \geq 0$) defining the relation $\to$:

$\bot \to \bot$
For every $c \in DC$:
  $c(E_1, \ldots, E_n) \to \bot$
  $c(E_1, \ldots, E_n) \to c(T_1, \ldots, T_n) \Leftarrow E_1 \to T_1 \wedge \ldots \wedge E_n \to T_n$
For every $f \in FS$:
  $f(E_1, \ldots, E_n) \to \bot$
  For every rule $f(t_1, \ldots, t_n) = e \in P$:
  $f(E_1, \ldots, E_n) \to T \Leftarrow E_1 \to t_1 \wedge \ldots \wedge E_n \to t_n \wedge e \to T$

Since P_L is a logic program, we may consider for it standard notions, like that of the *completion* of P_L [5], $Comp(P_L)$. The following are well known results about logic programs:

Proposition 3. *Let P be a CRWL-program and P_L its associated logic program. Then:*
(i) $Comp(P_L) \models P_L$
(ii) *There exists a least Herbrand model M_{P_L} of P_L, which is also the least model of $Comp(P_L)$.*
(iii) *If $e \to t$ is ground, then $P_L \models e \to t \Leftrightarrow M_{P_L} \models e \to t$*

There is a close relation between a CWRL-program P and its associated P_L, as given by the following result:

Proposition 4. *Let P be a CRWL-program and P_L its corresponding logic program. Then, for any expression e and term t,*
(i) $P_L \models e \to t \Leftrightarrow P \vdash_{\text{CRWL}} e \to t$.
(ii) $Comp(P_L) \models e \not\to t \Rightarrow P \nvdash_{\text{CRWL}} e \to t$ *(where $e \not\to t$ stands for $\neg(e \to t)$).*

We are interested in properties which are expressible as FOL formulas φ over the relation $\to$. In this sense, we consider the following FOL theories:

$T_{P_L} = \{\varphi \mid P_L \models \varphi\}$
$T_{Comp(P_L)} = \{\varphi \mid Comp(P_L) \models \varphi\}$
$T_{M_P} = \{\varphi \mid M_{P_L} \models \varphi\}$

We are mainly interested in the properties valid in M_{P_L}, that is, in T_{M_P}. But since M_{P_L} is a model of P_L and $Comp(P_L)$, we have $T_{P_L} \subseteq T_{Comp(P_L)} \subseteq T_{M_P}$, which means that in practice we can use P_L or $Comp(P_L)$ to obtain properties of M_{P_L} by FOL deduction.

Of course, T_{P_L} is a rather poor approximation to T_{M_P}. We find in $T_{Comp(P_L)}$ more interesting properties, in particular related to impossible reductions from a given expression. For instance, in the *Coin* example we have $Comp(Coin_L) \models double(coin) \not\rightarrow s(0)$, where $e \not\rightarrow t$ stands for $\neg(e \rightarrow t)$.

There are nevertheless many interesting properties of M_{P_L} which are not deducible from $Comp(P_L)$, in particular many inductive properties. In order to cope with (some of) these properties within the framework of FOL deduction, we consider the *inductive extension* of the completion.

Definition 1 (Inductive extension). Let P be a *CRWL* program and consider its completion $Comp(P_L)$. The inductive extension of the completion, $CompInd(P_L)$, results of adding to $Comp(P_L)$ the following axioms for the structural induction scheme:

For every formula φ with one free variable:

$\ldots \wedge \varphi(a) \wedge \ldots \wedge \varphi(g) \wedge \ldots \wedge$

$\ldots \wedge \forall x_1, \ldots, x_n.(\varphi(x_1) \wedge \ldots \wedge \varphi(x_n) \Rightarrow \varphi(c(\bar{x}))) \wedge \ldots \wedge$

$\ldots \wedge \forall x_1, \ldots, x_n.(\varphi(x_1) \wedge \ldots \wedge \varphi(x_n) \Rightarrow \varphi(f(\bar{x}))) \wedge \ldots$

$\Rightarrow \forall x.\varphi(x)$

where a, g range over DC^0 and FS^0, and c, f over DC^n and FS^n $(n > 0)$.

All these FOL axioms for induction are valid in M_{P_L}, and then $M_{P_L} \models CompInd(P_L)$. $CompInd(P_L)$ is powerful enough for proving many interesting properties of M_{P_L}. One example of formula valid in M_{P_L} that can be proved from $CompInd(P_L)$ but not from $Comp(P_L)$ is the above mentioned formula $\forall E, T.(E \rightarrow T \Rightarrow E + 0 \rightarrow T)$.

Let us discuss now how good is $CompInd(P_L)$ as axiomatization of M_{P_L}. If we call $T_{CompInd(P_L)} = \{\varphi \mid CompInd(P_L) \models \varphi\}$, we have the following chain of FOL theories:

$$T_{P_L} \subseteq T_{Comp(P_L)} \subseteq T_{CompInd(P_L)} \subseteq T_{M_P}$$

where we know that the first two inclusions are strict. It is easy to give examples showing that also $T_{CompInd(P_L)} \subseteq T_{M_P}$ is a strict inclusion (we start Sect. 5 with some of such examples). But note that this is an old known limitation of formalizations which come back to Gödel uncompleteness results. Since P_L, $Comp(P_L)$ and $CompInd(P_L)$ are recursive, T_{P_L}, $T_{Comp(P_L)}$ and $T_{CompInd(P_L)}$ are all recursively enumerable, while T_{M_P} is not, except for some very simple P.

4 Translation into some existing frameworks

In this section we put in practice the ideas introduced in the last section: to prove properties of the initial model of a *CRWL*-program, use the inductive extension of the completion of its associated logic program, and perform FOL deduction.

To this purpose, we have translated into several existing interactive proof assistants the inductive extension of the completion of some *CRWL*-programs. Actually, since all the used systems include induction as a built-in reasoning mechanism, it suffices to translate the completion.

To guide the discussion in this section, we use in all cases the program *Coin* in Fig. 1. and consider for it the following very simple properties:

($\mathbf{P}_1$) $double(coin) \rightarrow 0$: This formula is in fact a consequence of $Coin_L$.
($\mathbf{P}_2$) $double(coin) \nrightarrow s(0)$: This formula is in fact a consequence of $Comp(P_L)$.
($\mathbf{P}_3$) $\forall X, Y, T.(term(X) \wedge term(Y) \wedge X + Y \rightarrow T \Rightarrow Y + X \rightarrow T)$: This is an inductive property deducible from $CompInd(P_L)$, but not from $Comp(P_L)$. We make use of the auxiliary predicate *term* - defined in the natural way- to recognize if an expression is indeed a constructor term.

We have used ITP [10], LPTP [31] and Isabelle [23] as proof assistants. Different reasons are behind the choice of each one of these systems: our interest in ITP is explained by the relative proximity (see [26]) of *CRWL* and rewriting logic [22], the underlying logic of ITP; we expect LPTP to be useful for our purposes, because we translate *CRWL* into logic programming and LPTP is a specific tool for proving properties of logic programs; finally, Isabelle is a general purpose and widely used powerful proof assistant.
The ITP prover [10]: The ITP tool is designed to prove properties of the initial model of an equational specification written in Maude [9]. As it has been explained from the very beginning in this work, it would be unsound to introduce in ITP a *CRWL* program as an equational specification, because of the semantics of *CRWL* . Instead, we must specify the reduction relation $\rightarrow$ by means of equations giving the value *true* or $false$. In figure 2 part of this specification is shown. As it can be seen, the possible reductions are split by the rules that can be applied at this moment. The condition in the rules giving the value $false$ is, in consequence, the negation of the disjunction of the conditions of the rules giving *true*. To specify universal quantification we need to use new constants, which are denoted as C$*$.

```
op _->_ : Expression Expression -> Bool .
op _+_ : Expression  Expression -> Expression [ctor] .
op double : Expresison -> Expression [ctor] .
op coin : -> Expression [ctor] .
...
ceq (X + Y) -> T = true if eq(T, bottom) [label sumbot] .
ceq (X + Y) -> T = true if ((X -> 0) and (Y -> T)) .
ceq (X + Y) -> T = true if ((X -> s(T1)) and (s(T1 + Y) -> T)) [label sumI] .
ceq (X + Y) -> T = false if ((not eq(T, bottom)) and (not ((X -> 0) and (Y -> T)))
                           and (not ((X -> s(Z*)) and ((s(Z* + Y) ->T))))) [label redmas] .

ceq double(X) -> T = true if eq(T, bottom) [label doublebot] .
ceq double(X) -> T = true if ((X -> T1) and ((T1 + T1) -> T)) [label pdob] .
ceq double(X) -> T = false if ((not eq(T, bottom)) and (not (((X -> Y*)
                            and ((Y* + Y*) -> T))))) [label nredd] .

ceq coin -> T = true if eq(T, bottom) .
ceq coin -> T = true if (0 -> T) .
ceq coin -> T = true if (s(0) -> T) .
ceq coin -> T = false if ((not eq(T, bottom)) and (not(0 -> T)) and
(not (s(0) -> T))) .
...
```

Fig. 2. Part of Maude specification for *Coin*

Using this specification we obtain a perfect control on the nondeterministic reduction possibilities and therefore on the call-time choice semantics, but there is also a loss of automation when using the theorem prover tool.

We have tested this system with the three simple properties already mentioned. The property $\mathbf{P}_1$ is easily proved using this tool, but not automatically, as one would desire. This is because we need to make explicit which of the possible reductions of *coin* is adequate to instantiate the existential variable `T1` which appears in the rule for *double*. In ITP, in general, rules having new variables on their right hand side cannot be applied automatically, and the user must apply the rule manually by making explicit the rule instance which is interesting to apply. When dealing with negative properties like $\mathbf{P}_2$, it is needed an application of a rule for false reductions. Such rules cannot be applied neither automatically nor manually because of the introduction of the variables $C*$. Therefore, we need to prove lemmas specifying the condition with universally quantified variables. Many of this lemmas introduce numerous impossible cases increasing the length of the proof. Non-determinism of the reductions of expressions bring supplementary complexity because all possible ways to obtain the result are explored. Large proofs like that of $\mathbf{P}_3$ evolve into a chain of implications. This chain of implications is not directly treated as the tool does not have methods for reasoning on logical formulas. For example, to prove $e \to t = true \Rightarrow e' \to t' = true$ we do not simplify $e \to t$ to $e' \to t'$ because this cannot be done by any rewriting rule. Therefore we split the proof into two different modules, one using $e' \to t'$ and another using $e' \not\to t'$. The first one is the original one adding the implication step as assumption and therefore simulating the next step of the chain of implications. For the second one we have to prove, using a new lemma, the impossibility of such an assumption. When reasoning on the chain of implications we also introduce many negative proofs increasing the complexity. The successive steps of the proof are not automatic because they use internal assumptions of the module.

The LPTP prover [31]: LPTP is a theorem prover for success, failure and termination properties of Prolog programs. To use this tool we only have to translate a logic program expressing *CRWL* properties into a Prolog program. LPTP automatically generates the inductive completion of the program. One of the advantages of using this tool is that, being LPTP a prover for Prolog properties, the introduction of non-determinism does not cause as many problems as in ITP. Therefore, proving $\mathbf{P}_1$ is simpler with LPTP.

Testing the second and third properties LPTP has as many problems as ITP. First, there are too many possibilities in the reduction relation for negative or universally quantified properties. Second, the proof simplifies the goal adding the corresponding assumptions to the theory. This causes a growing on the number of variables. For properties as simple as those introduced here the system generates a complex proof of more than one thousand lines.

Isabelle [23]: Isabelle/HOL is a theorem prover where specifications and validations are considered on Higher-Order logic. In this case we specify the system as an inductive set for the least model of the logic program. In such a least model we can prove positive and negative facts about the reduction relation and also inductive properties of it. In figure 3 appears part of the theory on which the results are proved.

Isabelle provides methods to reason on logic formulas, relations and sets. Using these methods the property $\mathbf{P}_1$ was proved automatically. Negative properties like $\mathbf{P}_2$ require reasoning on the completion. This can be done using axioms for inductive sets. Similarly as in the other systems, the different ways to derive the same term in *CRWL* introduce many repeated facts to be proved. On the other hand it is not difficult to prove known facts of this calculus such as transitivity of the reduction relation. Inductive properties like $\mathbf{P}_3$ can

```
theory Arrows = Main:

datatype exp = bottom | zero | s exp | coin | sum exp exp | double exp

consts arrow :: "(exp * exp) set"
inductive arrow
intros
bt [intro]: "(x, bottom) : arrow"
dczero [intro]: "(zero, zero) : arrow"
dcs [intro]: "(x, t):arrow ==> (s x, s t):arrow"
fcoin1 [intro]: "(zero, t):arrow ==> (coin, t):arrow"
fcoin2 [intro]: "(s(zero), t):arrow ==> (coin, t):arrow"
sum1 [intro]: "[|(x, zero):arrow ; (y, t):arrow|] ==> (sum x y,t):arrow"
sum2 [intro]: "[|(x, s(t1)):arrow ; (y,t2):arrow ; (s(sum t1 t2),t):arrow|]
               ==> (sum x y , t):arrow"
double [intro]: "[|(x, t1):arrow ; (sum t1 t1,t):arrow|] ==> (double(x), t):arrow"
...
```

Fig. 3. Part of Isabelle specification for *Coin*

be expressed by a first order logic formula, then applying the rules for such formulas it is not difficult to prove the property. This translation does not introduce limitations on the formulas that can be specified nor on the induction mechanisms.

4.1 Improving determinism of *CRWL*

A common problem arising in the three approaches is the repetition of essentially the same proofs. The problem comes from the source logic CRWL. For a constructor term t, *CRWL* provides many different approximations $t \to t'$, for all $t' \sqsubseteq t$, that is, for all different t' obtained by replacing some subterms of t by $\bot$. This kind of non-determinism of $\to$ can be avoided, since for constructor terms t, only the maximal approximation $t \to t$ is really necessary. In this section we present a simplified *CRWL* calculus eliminating all those superfluous reductions associated to terms.

Definition 2 (CRWL'). The proof calculus *CRWL'* results of replacing the rule *(BT)* in *CRWL* (Fig. 1) by the new rule *(BT')*

$$\frac{}{e \to \bot} \quad \text{if } e = f(e_1, \ldots, e_n) \text{ or } e = \bot$$

The next result relates the provable statements of *CRWL* and *CRWL'*.

Proposition 5. *Let P be a CRWL-program. For any expression e and any term t:*

(i) $P \vdash_{CRWL} e \to t \Rightarrow P \vdash_{CRWL'} e \to t'$ *for some* $t' \sqsupseteq t$.
(ii) $P \vdash_{CRWL'} e \to t \Rightarrow P \vdash_{CRWL} e \to t$
As a consequence, if t is a total term: $P \vdash_{CRWL} e \to t \Leftrightarrow P \vdash_{CRWL'} e \to t$

We have tested our sample properties with the refined calculus *CRWL'*, conveniently translated to the different systems, obtaining significant shortenings in the proofs. Furthermore, since reduction between c-terms is now deterministic, it is possible to use equational reasoning in those parts of the proofs involving this kind of reductions. This has been a further source of simplification of the proofs while using ITP.

5 Beyond the completion: axiomatizing derivability

As we discussed at the end of Sect. 3, no FOL axiomatization can be complete for the least model of a program. In the case of $CompInd(P_L)$, although it covers many interesting properties, it is nevertheless quite easy to find examples revealing its limitations. Consider for example the following simple program *Loop*:

```
loop → loop
```

It is not difficult to see that $loop \twoheadrightarrow 0$ is valid in M_{Loop_L}, but $CompInd(Loop_L) \not\models loop \twoheadrightarrow 0$. A less trivial example is given by the following program *Even*:

```
even(0) → true          an_even → 0
even(s(0)) → false      an_even → s(s(an_even))
even(s(s(X))) → even (X)
```

Notice that *an_even* admits an infinite number of reductions giving all the even natural numbers. The property $even(an\_even) \twoheadrightarrow false$ is valid in M_{Even_L} but, again, is not deducible from $CompInd(Even_L)$.

We remark that the two given examples express negative properties involving nontermination. It is not so strange that completion is not able to prove them, since it is known that completion is related to finite failure. But nontermination analysis by itself does not suffice to prove the properties. Notice also that, in both cases, the properties can be proved by inductive reasoning over the universe of *CRWL*-derivations. This suggests some meta-theory at the object level, by considering a variant of *CRWL* (to be precise, of the logic program mirroring *CRWL*) where the *CRWL*-derivation trees for statements $e \to t$ are made explicit.

We first introduce some constructor terms representing CRWL-derivations.

Definition 3 (Derivation terms). The set of *derivation constructors symbols CSDer* consists of the following symbols:

$bt \in CSDer^0$
$dc_c \in CSDer^k$ for every $c \in DC^k$
$fa_{f,R} \in CSDer^{k+1}$ for every $f \in FS^k$ and every R rule for f.

Constructor terms built up with derivation constructors are called *derivation terms*.

We will use $d, d', \ldots$ to denote derivation terms.

Now, given a *CRWL*-program P, we associate to it a logic program defining a ternary relation $d \vdash e \to t$ whose intended meaning is *'d represents a CRWL-derivation of $e \to t$'*.

Definition 4 (Axiomatization of derivability (logic program)). Given a *CRWL* program P the associated logic program making explicit the proofs, $Der(P)$, consists of the following clauses defining the ternary relation $\_ \vdash \_ \to \_$:

$bt \vdash \bot \to \bot$
For every $c \in DC$:
$bt \vdash c(E_1, \ldots, E_n) \to \bot$
$dc_c(D_1, \ldots, D_n) \vdash c(E_1, \ldots, E_n) \to c(T_1, \ldots, T_n)$
$\Leftarrow D_1 \vdash E_1 \to T_1 \wedge \ldots \wedge D_n \vdash E_n \to T_n$

For every $f \in FS$:

$bt \vdash f(E_1, \ldots, E_n) \rightarrow \bot$

For every rule $\mathcal{R}$ $f(t_1, \ldots, t_n) = e$ for this f:

$fa_{f,\mathcal{R}}(D_1, \ldots, D_n, D) \vdash f(E_1, \ldots, E_n) \rightarrow T$
$\Leftarrow D_1 \vdash E_1 \rightarrow t_1 \wedge \ldots \wedge D_n \vdash E_n \rightarrow t_n \wedge D \vdash e \rightarrow T$

As we did with P_L in Sect. 3, we can think on the least model $M_{Der(P)}$ of $Der(P)$, the completion $Comp(Der(P))$ and its inductive extension $CompInd(Der(P))$.

In the *Loop* and *Even* examples, we have $CompInd(Der(Loop)) \models loop \nrightarrow 0$ and $CompInd(Der(Even)) \models even(an\_even) \nrightarrow false$.

We explore now some logical relations between $Der(P)$ and the original program. Our first result relates the reduction statements derived using this approach and those of the original calculus.

Proposition 6. *For every P CRWL program, and for every e expression and t term:*

(i) $Der(P) \models \exists D.D \vdash e \rightarrow t \Leftrightarrow P_L \models e \rightarrow t \Leftrightarrow P \vdash_{\text{CRWL}} e \rightarrow t$
(ii) $Comp(Der(P)) \models \nexists D.D \vdash e \rightarrow t \Rightarrow P \nvdash_{\text{CRWL}} e \rightarrow t$

In order to compare the behavior of $Der(P)$ with respect to more general properties φ, we define a natural conversion of FOL formulas using the relation $\_ \rightarrow \_$ into formulas using $\_ \vdash \_ \rightarrow \_$, as well as a natural relation between models of P_L and of $Der(P)$.

Definition 5. *(i)* If φ is a FOL formula using the relation $\_ \rightarrow \_$, we write $\widehat{\varphi}$ for the result of replacing in φ each subformula $e \rightarrow t$ by $\exists D.D \vdash e \rightarrow t$ (with D not occurring in $e \rightarrow t$).
(ii) Let M be a model for P_L, we define the following set S_M of models of $Der(P)$:

$$S_M = \{M' \models Der(P) \mid \forall e, t(M \models e \rightarrow t \Leftrightarrow \textit{exists } d \textit{ such that } M' \models d \vdash e \rightarrow t\}$$

Proposition 7. (i) $M' \models Der(P)$ *iff there exists* $M \models P_L$ *such that* $M' \in S_M$
(ii) $M_{Der(P)} \in S_{M_{P_L}}$

The following result relates validity in a model of P_L with validity in the corresponding model of $Der(P)$

Proposition 8. *Let φ be a formula and M model of P_L and $M' \in S_M$ then:*

$$M \models \varphi \Leftrightarrow M' \models \widehat{\varphi}$$

In particular,

$$M_{P_L} \models \varphi \Leftrightarrow M_{Der(P)} \models \widehat{\varphi}$$

As a consequence of the previous results, we conclude also that the properties derived from P_L and from $Der(P)$ are the same (via $\widehat{\_}$), as stated by the following proposition:

Proposition 9. *For any φ, $P_L \models \varphi \Leftrightarrow Der(P) \models \widehat{\varphi}$.*

All these results show that nothing new can be obtained from $Der(P)$ and $M_{Der(P)}$ with respect to P_L and M_{P_L}. The *Loop* and *Even* examples show that the real gain comes from $CompInd(Der(P))$ with respect to $CompInd(P_L)$. Therefore, those properties not involving reasoning on the structure of the CRWL-derivation will be proved using the first approach, where the proofs are simpler. Only when reasoning on the structure of the derivation is needed the second approach will we used.

We have tested this new approach with ITP and Isabelle. As it was expected, with this approach we can prove properties reasoning by structural induction on the derivation terms. As an example, consider the program *Loop*. Its associated translation into Isabelle is shown in figure 4. It is not too difficult to prove $loop \twoheadrightarrow 0$ reasoning by induction on the derivations and discarding all those incorrect derivations. The proof is slightly complicated because of the introduction of such incorrect cases, but the steps are not difficult.

As it has been previously remarked, the resulting proofs with the new approach can be in general more complicated than the corresponding ones with the original approach, whenever the latter is applicable. But this is not always true. For instance, consider again the program *Coin* and the sample properties of section 4. The property $\mathbf{P}_1$, rephrased as $\exists D.D \vdash coin \rightarrow 0$, can be still proved automatically in Isabelle. The situation is different for negative properties like $\mathbf{P}_2$, that are expressed in the new approach as universal quantifications over derivations. Therefore, when trying to prove such negative properties we have to inspect all possible derivations. There are only a few of them possible for a given expression as can be deduced from the logic program Der_P, but all the possibilities have to be explored, hence complicating the proof.

```
theory Demos = Main:

datatype exp = bottom | zero | s exp | coin | sum exp exp | double exp | loop
datatype dem = bt | dczero | dcs dem | facoin1 dem | facoin2 dem | fasum1 dem dem
              |fasum2 dem dem dem | fadouble dem dem | faloop dem
consts demo :: "(dem * exp * exp) set"
inductive demo
intros
rbt [intro]: "(bt, x, bottom) : demo"
rdczero [intro]: "(dczero, zero, zero) : demo"
rcs [intro]: "(d, x, t):demo ==> (dcs d, s x, s t):demo"
rfcoin1 [intro]: "(d, zero, t):demo ==> (facoin1 d, coin, t):demo"
rfcoin2 [intro]: "(d, s(zero), t):demo ==> (facoin2 d, coin, t):demo"
rsum1 [intro]: "[|(d, x, zero):demo ; (d1, y, t):demo|]
                ==> (fasum1 d d1, sum x y, t):demo"
rsum2 [intro]: "[|(d, x, s(t1)):demo ; (d1, y,t2):demo ; (d2, s(sum t1 t2), t):demo|]
               ==> (fasum2 d d1 d2, sum x y , t):demo"
rdouble [intro]: "[|(d, x, t1):demo ; (d1, sum t1 t1,t):demo|]
               ==> (fadouble d d1, double(x), t):demo"
rloop [intro]: "(d, loop, t):demo ==> (faloop d, loop, t):demo"
```

Fig. 4. Isabelle specification of the least model of $Der(P)$

6 Conclusions

We have presented some logical conceptual tools for proving properties of first order functional logic programs. Programs consist of constructor based rewrite systems possibly non-terminating and non-confluent, defining thus non-strict non-deterministic functions, with call-time choice semantics. This corresponds to the first order core of existing modern FLP systems like Curry or Toy.

Our logical starting point has been *CRWL*, a well known semantic framework for FLP. *CRWL* includes a proof calculus giving logical semantics to programs, and a model theory

satisfying that every program has an initial model. The program properties of interest are those valid in that initial model, which are then typically inductive properties.

In order to prove such program properties, we have mapped *CRWL* into logic programming in the following sense: to each *CRWL*-program P we associate in a simple manner a logic program P_L such that the least model of P_L consists exactly of the reduction statements which are *CRWL*-provable from P. As a nice consequence, all the machinery (theoretical and practical) of logic programming is available to us. For instance, the completion of P_L can be used to deduce negative results, and with its inductive extension we can deduce inductive properties of the least model.

We have made experiments with this approach by encoding into several existing proof assistants the completion of simple programs (the inductive extension is implicit in all these systems). Namely, we have used: ITP [10], a tool based on rewriting logic [22] and designed for proving properties of equational specifications; LPTP [31], a tool designed specifically for logic programs; and Isabelle [23], a well known general purpose proof assistant. In all cases, to prove simple properties of CRWL-programs is not as easy as one would desire. We have detected two particular aspects having great impact in the simplicity of proofs. One is, of course, the concrete encoding: for instance, in the ITP case, an unsorted version was clearly worse than the sorted one (distinguishing terms and expressions). The other one is the formulation of the *CRWL* logic itself: we have proposed a refinement eliminating superfluous sources of non-determinism of the reduction relation $\rightarrow$, with which some proofs are remarkably simpler and shorter.

Of course, due to Gödel-like arguments, no deductive system can prove all properties of initial models. The limits of the completion+induction approach are easily reachable by considering properties which are valid due to non-termination. This is natural, since completion is closely related to finite failure.

To enlarge the class of provable properties we have then sophisticated the logic programming specification P_L of the semantics of a *CRWL*-program P, by making explicit the *CRWL*-proof tree corresponding to *CRWL*-provable reduction statements for P. The resulting logic program $Der(P)$ has its own completion $Comp(Der(P))$, inductive extension of the completion $CompInd(Der(P))$, and its least model $M_{Der(P))}$. An interesting point is that the logical consequences of $Der(P)$ and $Comp(Der(P))$ are essentially the same of P_L and $Comp(P_L)$, and the same happens with the valid properties in M_{P_L} and $M_{Der(P)}$. What produces new results is $CompInd(Der(P))$ with respect to $CompInd(P_L)$, as we have indeed shown in our implementations.

We have in mind many things to do as future work. In the practical side it is important to test the approach with interesting non trivial case studies and to use other existing theorem provers like SPASS [12] and SATURATE [32]. In the theoretical side we plan to improve the approach by making the mapping of logics more precise, refining the target logic by considering many sorted logic programs, and refining the source logic by considering extensions of *CRWL* with other features like HO [15,16] or failure [21].

References

1. M. Alpuente, F.J. Correa, M. Falaschi. *A Debbugging Scheme of Functional Logic Programs*, Proc. WFLP'01, Electronic Notes on Theoretical Computer Science, Vol 64, 2002.
2. M. Alpuente , D. Ballis , F.J. Correa , M. Falaschi *Automated Correction of Functional Logic Programs*, Proc. European Symp. on Programming (ESOP'03), Springer LNCS 2618, pp. 54-68, 2003.
3. S. Antoy. *Optimal Non-deterministic Functional Logic Computations*, Proc. ALP/HOA 1997, Springer LNCS 1298, pp. 16–30, 1997.

4. S. Antoy Constructor-based Conditional Narrowing. Proc. Principles and Practice of Declarative Programming (PPDP'01), 199–206, ACM Press, 2001.
5. K.R. Apt. *Logic Programming*. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B, Chapter 10, Elsevier and The MIT Press, pp. 493–574, 1990.
6. D. Bert, R. Echahed. *Abstraction of Conditional Term Rewriting Systems*. Proc ILPS 1995 , pp. 162–176, 1995.
7. R. Caballero, F.J. López-Fraguas, M. Rodríguez-Artalejo. *$\mathcal{DDT}$: Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs*. Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS'2001), Springer LNCS 2024, pp. 170–184, 2001.
8. R. Caballero, M. Rodríguez-Artalejo. *A Declarative Debugging System for Lazy Functional Logic Programs*. Electronic Notes in Theoretical Computer Science 64, 63 pages, 2002.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott. *Maude 2.0 Manual*. `http://maude.cs.uiuc.edu`, 2003.
10. M. Clavel. *The ITP tool*. In A. Nepomuceno, J. F. Quesada, and J. Salguero, editors, Logic, Language and Information. Proc. of the 1st Workshop on Logic and Language, Kronos, 55–62, 2001. System available at `http://www.ucm.es/info/dsip/clavel/itp`.
11. H.B. Enderton. *A Mathematical Introduction to Logic*, Academic Press, 2001.
12. H. Ganzinger, R. Nieuwenhuis, P. Nivela. *The Saturate System*. In http://www.mpi-sb.mpg.de/SATURATE, 1994.
13. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *A Rewriting Logic for Declarative Programming*. Proc. European Symp. on Programming (ESOP'96), Springer LNCS 1058, pp. 156–172, 1996.
14. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. Journal of Logic Programming 40(1), pp. 47–87, 1999.
15. J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo. *A Higher Order Rewriting Logic for Functional Logic Programming*. Proc. Int. Conf. on Logic Programming, The MIT Press, pp. 153–167, 1997.
16. J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo. *Polymorphic Types in Functional Logic Programming*. FLOPS'99 special issue of the Journal of Functional and Logic Programming, 2001. `http://danae.uni-muenster.de/lehre/kuchen/JFLP`.
17. M.J.C. Gordon and T.F. Melham. *Introduction to HOL*, Cambridge Univ. Press, 1993.
18. M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.8, April 15, 2003. `http://www-i2.informatik.uni-kiel.de/~curry/`.
19. H. Hussmann. *Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting*. Journal of Logic Programming 12, pp. 237–255, 1992.
20. F.J. López Fraguas, J. Sánchez Hernández. *$\mathcal{TOY}$: A Multiparadigm Declarative System*. Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999.
21. F.J. López-Fraguas, J. Sánchez-Hernández. *A Proof Theoretic Approach to Failure in Functional Logic Programming*. Theory and Practice of Logic Programming 4(1), pp. 41–74, 2004.
22. J. Meseguer. *Conditional Rewriting Logic as a Unified Model of Concurrency*. Theoretical Computer Science 96, pp. 73–155, 1992.
23. T. Nipkow, L.C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higer-Order Logic*. Springer LNCS 2283, 2002.
24. P. Padawitz. *Inductive Theorem Proving for Design Specifications*, J. Symbolic Computation 21, 41–99, 1996.
25. P. Padawitz. *Swinging Types = Functions + Relations + Transition Systems*, Theoretical Computer Science 243, 93–165, 2000.
26. M. Palomino Tarjuelo. *Comparing Meseguer's Rewriting Logic with the Logic CRWL*. Electronic Notes in Theoretical Computer Science 64, 22 pages, 2002.
27. L.C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Univ. Press, 1987.
28. D. Pedreschi, S. Ruggieri. *Verification of Logic Programs*. J. Log. Program. 39 (1-3), pp. 125-176, 1999.
29. M. Rodríguez-Artalejo. *Functional and Constraint Logic Programming*. in H. Comon, C. Marché and R. Treinen (eds.), *Constraints in Computational Logics, Theory and Applications*, Revised Lectures of the International Summer School CCL'99, Springer LNCS 2002, Chapter 5, pp. 202–270, 2001.
30. Jaime Sánchez-Hernández, Una Aproximación al fallo en programación declarativa multiparadigma. PhD Univ. Complutense Madrid, 2004 (in spanish).

31. R.F. Stäerk. *The theoretical foundations of LPTP (A logic program theorem prover).* Journal of Logic Programming 36, pp. 241–269, 1998.
32. C. Weidenbach, U. Brahn, T. Hillenbrand, E. Keen, C. Theobald and D. Topic. *SPASS version 2.0* Proc. of the 18th International Conference on Automated Deduction CADE'02, Springer LNCI 2392, pp. 275-279, 2002.

TypeTool: A Type Inference Visualization Tool

Hugo Simões and Mário Florido

University of Porto, DCC & LIACC
R. do Campo Alegre 823, 4150-180 Porto, Portugal
{hrsimoes,amf}@ncc.up.pt

Abstract TypeTool is a tool for visualizing the type inference process for functional programming languages. It is a Web-based tool, which is freely usable at the Web in a quite simple way. This tool is especially useful for students, because it helps to understand the type systems of the most common typed functional languages. The tool is also useful for programmers who want to quickly become aware of the functional paradigm. TypeTool shows in detail the constraints generated by the type inference algorithm for any given expression, their solutions and their application in the type inference process. It deals with the Simple Type System for the Lambda-calculus and with the polymorphic type system of pure ML.

1 Introduction

The important role of type systems in modern, higher-order programming languages such as Haskell and ML is now well established.

Type systems are powerful verification tools which guide the programmer by pointing out errors at compile-time. One big problem of these systems is the lack of clarity and conciseness of type errors reported by modern compilers. Thus it is not easy to see what modifications are needed to fix an ill-typed program. These problems are exacerbated when teaching functional programming using typed languages. Languages such as Haskell and ML are excellent choices for teaching introductory programming. One reason for this is exactly their type discipline which helps to enforce a design discipline for the beginning programmer. But unfortunately, the lack of clarity of type errors messages, sometimes discourages the students from programming in a functional language, and rejection of ill-typed programs may be seen as a nuisance instead of a blessing.

When the University of Porto implemented programming courses based on a typed functional programming language (in this particular case the chosen language is Haskell), we presented the basis of type inference technology early in the course program. This significantly improved the way students deal with type errors because they understand the type system. This understanding makes it more easy to fix ill-typed programs and clearly convinces the student that rejecting ill-typed programs at compile time is a benefit, not a nuisance.

The main difficulty to teach type systems to beginners programmers was their lack of the formal reasoning needed to fully understand these systems. To address these problems we built *TypeTool*, a type inference visualization tool initially targeted at students.

This paper presents TypeTool, a simple type inference visualization tool, presents several examples, describes its overall architecture and sketches solutions to some implementation issues.

Although TypeTool was first designed for students, the tool has grown past its original goals. It can be useful for any beginning programmer in a functional programming language. One can try TypeTool at its web site:

http://www.ncc.up.pt/typetool/

Due to space limitations, in this paper we will only show TypeTool applied to pure ML. But our system also works for the Simple Type System for the λ-calculus [4].

We assume that the reader is familiar with functional programming and type inference for functional programs. A good survey of the area may be found in [15]. We start in section 2 with the presentation of the related work. In section 3 we present the type system of pure ML. Then, some examples are given in section 4. In sections 5 and 6 we briefly describe TypeTool's architecture and present our technology and programming languages choices for implementing this application. And finally we conclude in section 7.

2 Related Work

There was some previous work into visualizing the type inference process for functional languages. New visual models for improving the understanding of type inference and type errors were presented by Jung and Michaelson in [13] and by Erwig in [7]. The goal of TypeTool is to have a system as simple as possible to help us on teaching the basics of type inference technology. We concentrate the visualization on three main processes of a type inference algorithm: type constraints generation, constraint solving and annotating the program with the inferred types. Thus our visualization model is far simpler than the previous ones being quite effective with respect to our initial goals. There were also simple visualization tools for new type systems (more complex than the ML type system) (see [18]).

Interactive tools that explain the type inference algorithm step by step were presented in [17,6]. Other tools enabled the programmer to browse through the program syntax to see the type of subexpressions (see [12,3]). This last approach had clear advantages because it avoided the huge output that can be produced by other systems (including TypeTool) when applied to big programs. For now this is not a problem in our system because TypeTool is being used to help to understand the most used type systems for a functional core language, where typical examples are small expressions. However, if we want to extend TypeTool to deal with real programs we will also have to be able to show partial type information. To deal with this problem we intend to show annotated subtrees previously selected by the user, but this is left for future work.

There was also a lot of work trying to improve the quality of error messages and to give better explanations of type errors ([11,20,19,14,10]). These systems had as initial goal to explain better type errors to programmers which do not understand the type system. Our goal is different: we want to help the programmer to understand better the details of the type system.

3 The Damas-Milner Type System

The Damas-Milner type system [5] is the base of type systems for languages that allow the use of parametric polymorphism, such as ML or Haskell.

Here we assume the reader to be familiar with standard notation for the λ-calculus [2].

3.1 The Term Language

Given an infinite set of variables $\mathcal{V}$, the term language is defined by the following grammar:

$$M ::= x \mid MM' \mid \lambda x.M \mid let\ x = M\ in\ M'$$

3.2 Types

We now briefly describe the syntax of types and the Damas-Milner type system.

Definition 31 *Assuming we are given a set of type variables α, the syntax of types is given by $\tau ::= \alpha \mid \tau' \rightarrow \tau''$.*

Definition 32 *We say that σ is a* type scheme *if σ is a type τ or σ is of the form $\forall\alpha_1, \ldots, \alpha_n.\tau$, where $\alpha_1, \ldots, \alpha_n$ are type variables called* generic *variables.*

Definition 33 *Let τ be a type and σ be a type scheme, τ is a* generic instance *of σ if and only if $\sigma = \tau$ or $\sigma = \forall\alpha_1, \ldots, \alpha_n.\tau'$ and $\exists\tau_1, \ldots, \tau_n$ such that $\tau = [\tau_i/\alpha_i]\tau'$ ($[\tau_i/\alpha_i]$ denotes the substitution of α_i by τ_i).*

Definition 34 *An* assumption *is a pair of the form $x : \sigma$ where x is a variable and σ is a type scheme. A set of assumptions is called an* environment.

3.3 The Type System

Let $x \in \mathcal{V}$, $\alpha \in \mathbb{V}$, M and M' be terms, τ and τ' types and σ and σ' type schemes. Let $\Gamma \vdash M : \sigma$ mean that term M has type σ given the environment Γ. The Damas-Milner type system is defined by the following rules:

(*Axiom*) $$\Gamma \vdash x : \sigma, \text{ if } (x : \sigma) \in \Gamma$$

(*Generalization*) $$\frac{\Gamma \vdash M : \sigma, \text{there is no free occurrence of } \alpha \text{ in } \Gamma}{\Gamma \vdash M : \forall\alpha.\sigma}$$

(*Instantiation*) $$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M : \sigma[\tau/\alpha]}$$

(*Application*) $$\frac{\Gamma \vdash M : (\tau' \rightarrow \tau), \Gamma \vdash M' : \tau'}{\Gamma \vdash (MM') : \tau}$$

(*Abstraction*) $$\frac{\Gamma \cup \{x : \tau'\} \vdash M : \tau}{\Gamma \vdash \lambda x.M : (\tau' \rightarrow \tau)}$$

(*Let*) $$\frac{\Gamma \vdash M : \sigma, \Gamma \cup \{x : \sigma\} \vdash M' : \tau}{\Gamma \vdash let\ x = M\ in\ M' : \tau}$$

Example 1. The type derivation for *let* $i = (\lambda x.x)$ *in* ii is given in figure 1.

3.4 Type Inference

The Damas-Milner type system is decidable. The type inference algorithm for this system, originally presented in [5], follows:

Definition 35 *Let V be a set of variables, Γ an environment and $\tau \in \mathbb{T}$. The closure of τ with respect to V, $\overline{V}(\tau)$, is the type scheme $\forall\alpha_1, \ldots, \alpha_n.\tau$, where $\alpha_1, \ldots, \alpha_n$ are variables that occur in τ and are not in V. $\overline{\Gamma}(\tau)$ is the closure of τ with respect to the set of variables that occur in Γ.*

Definition 36 *Let Γ be an environment, M a term, $\tau \in \mathbb{T}$ and S a substitution. Let $UNIFY$ be Robinson unification [16] such that $UNIFY(\tau_1, \tau_2) = S$, where S is the most general unifier of τ_1 and τ_2. Function $W(\Gamma, M) = (S, \tau)$ defines the type inference algorithm for the Damas-Milner type system:*

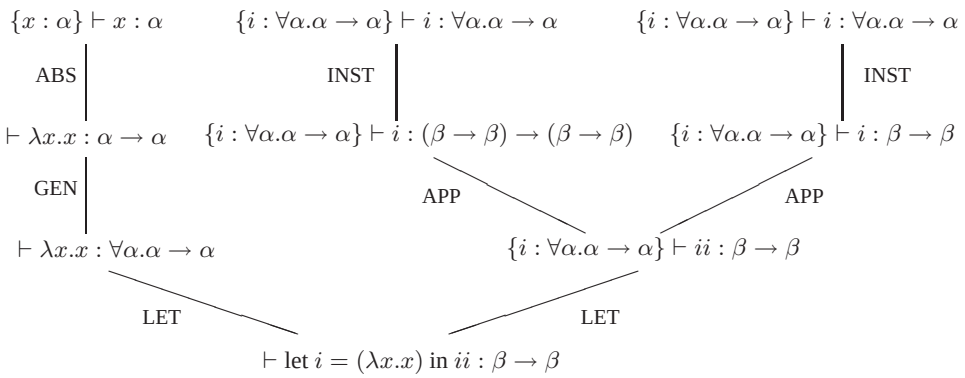


Figure 1. Type derivation for *let* $i = (\lambda x.x)$ *in* ii

1. *If M is a variable x and* $x : \forall\alpha_1, \ldots, \alpha_n.\tau' \in \Gamma$ *then S is the identity function and* $\tau = [\beta_i/\alpha_i]\tau'$, *where each* β_i *is a fresh variable* $(1 \leq i \leq n)$.
2. *If* $M \equiv M_1 M_2$, *let:*
   - $W(\Gamma, M_1) = (S_1, \tau_1)$;
   - $W(S_1\Gamma, M_2) = (S_2, \tau_2)$;
   - $U = UNIFY(S_2\tau_1, \tau_2 \to \beta)$, *where* β *is a fresh variable;*

   then $S = U \circ S_2 \circ S_1$ *and* $\tau = U\beta$;
3. *If* $M \equiv \lambda x.N$, *let* β *be a fresh variable and* $W(\Gamma_x \cup \{x : \beta\}, N) = (S_1, \tau_1)$, *then* $S = S_1$ *and* $\tau = S_1\beta \to \tau_1$;
4. *If* $M \equiv let\ x = M_1\ in\ M_2$, *let:*
   - $W(\Gamma, M_1) = (S_1, \tau_1)$ *and*
   - $W(S_1\Gamma_x \cup \{x : \overline{S_1\Gamma}(\tau_1)\}, M_2) = (S_2, \tau_2)$

   then $S = S_2 \circ S_1$ *and* $\tau = \tau_2$

In [5] it was proved the soundness and completeness of this algorithm.

The following example, where we sketch an application of W to an expression by specifying the arguments and results of W and $UNIFY$, illustrates the main features of the type inference algorithm.

Example 2. Consider the application of function W to *let* $i = (\lambda x.x)$ *in* ii.

$W(\{\}, \lambda x.x) = ([\,], \alpha \to \alpha)$

Using the closure of type $(\alpha \to \alpha)$, as the type of i in $W(\{i : \forall\alpha.\alpha \to \alpha\}, ii)$, we have:

$W(\{i : \forall\alpha.\alpha \to \alpha\}, i) = ([\,], \alpha_1 \to \alpha_1)$ and

$W(\{i : \forall\alpha.\alpha \to \alpha\}, i) = ([\,], \alpha_2 \to \alpha_2)$

Since $U = UNIFY(\alpha_1 \to \alpha_1, (\alpha_2 \to \alpha_2) \to \beta) = [(\alpha_2 \to \alpha_2)/\alpha_1, (\alpha_2 \to \alpha_2)/\beta]$, then:

$W(\{\}, \text{let } i = (\lambda x.x) \text{ in } ii) = (U, \alpha_2 \to \alpha_2)$

Note that we could infer a type for this term because i is used in (ii) with two different types.

4 Using TypeTool

Throughout this section we will see some examples of the graphical interface illustrating a type inference request.

To perform an inference request, the user can choose one of the following methods [see figure 2]:

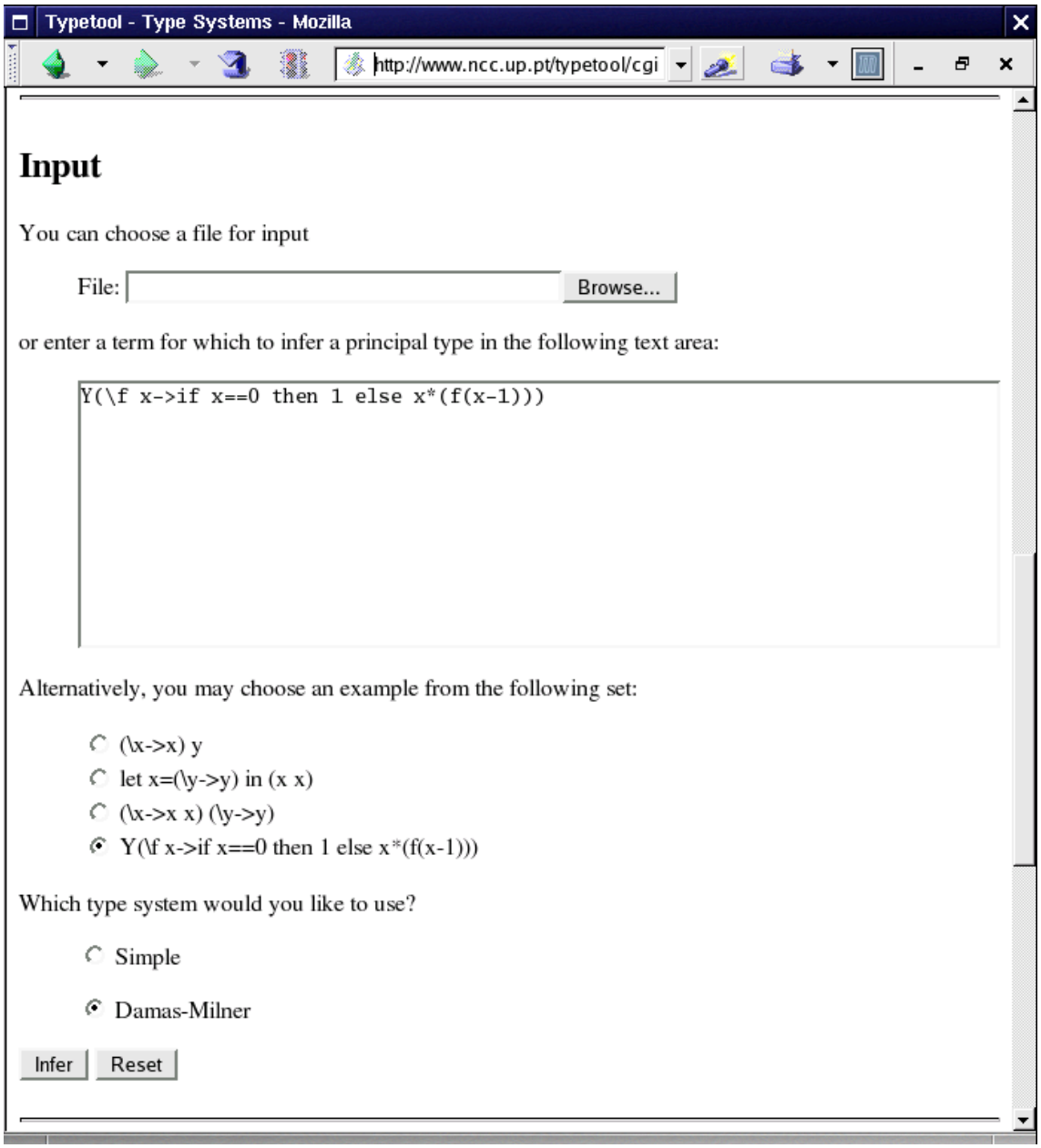


Figure 2. Choosing a term

- upload a file containing the term for which one wants to infer the type
- write the term directly on the text area
- select one of the predefined examples

This alternatives were described in decreasing priority. Thus, if we write a term and an example is selected, the application considers as input only the written term. File upload has priority over the two other alternatives.

The term in figure 2 is the factorial function (as usual Y stands for the fixed point operator), for which we will infer a type in the Damas-Milner system. Clicking on the “infer” button will take us to an HTML web page with the inference result [see figures 3, 4 and 5].

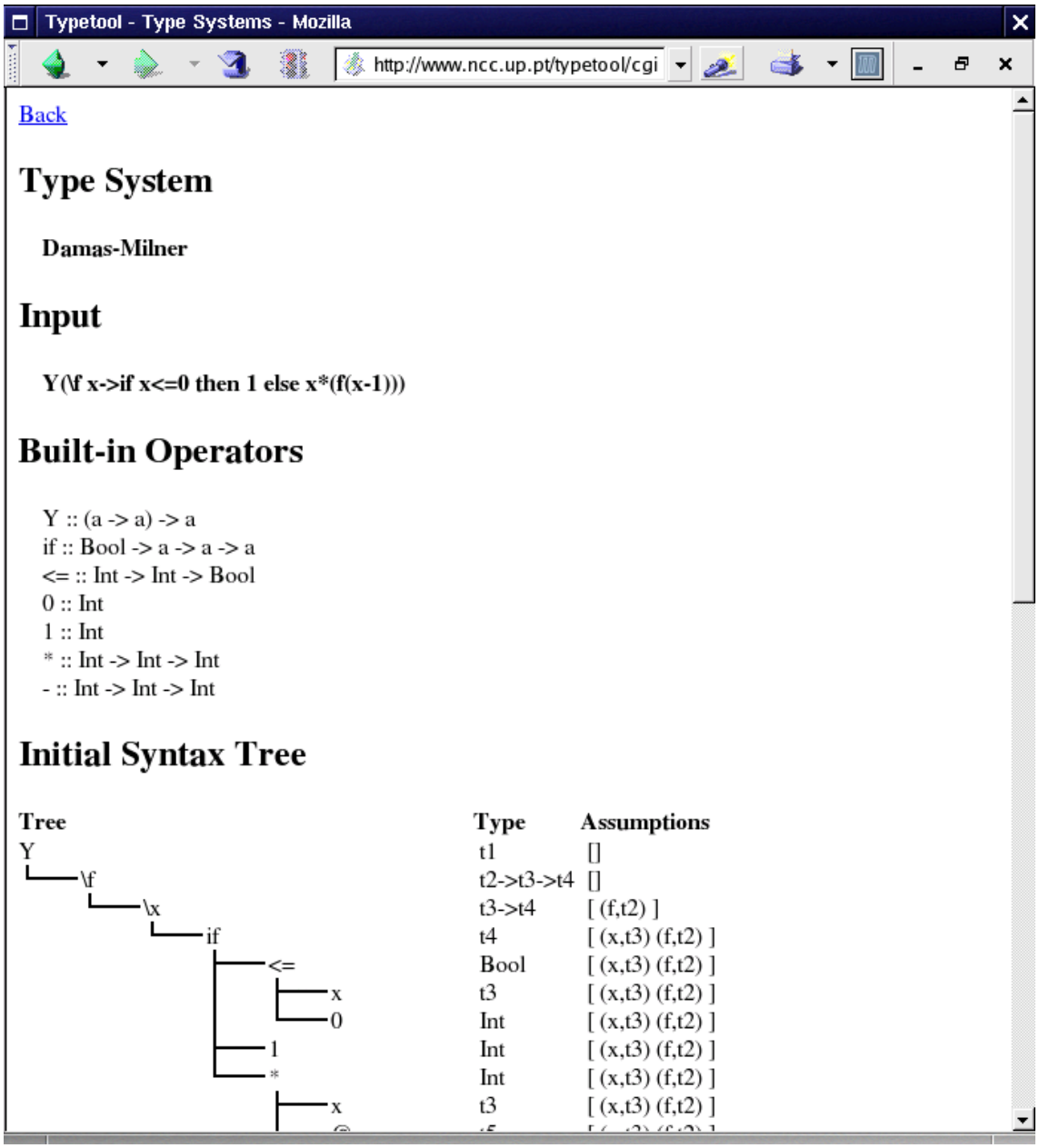


Figure 3. List of built-ins

In figure 3 we can see which type inference system was selected and which term was introduced. Next, the built-in operators[1] that were found in the term are presented. Note that built-ins are treated as special variables with their corresponding types declared in an initial set of assumptions.

In figure 4 we have the syntactic tree of the term, for which we can see the initially generated type and, right below, which constraints (equations) were produced during the inference process. Applying the unification algorithm to this constraint set, we can get the final substitution. This substitution is then applied to the initial syntax tree to obtain the final derivation tree [see figure5].

[1] built-in operators added to the term language of the Damas-Milner type system

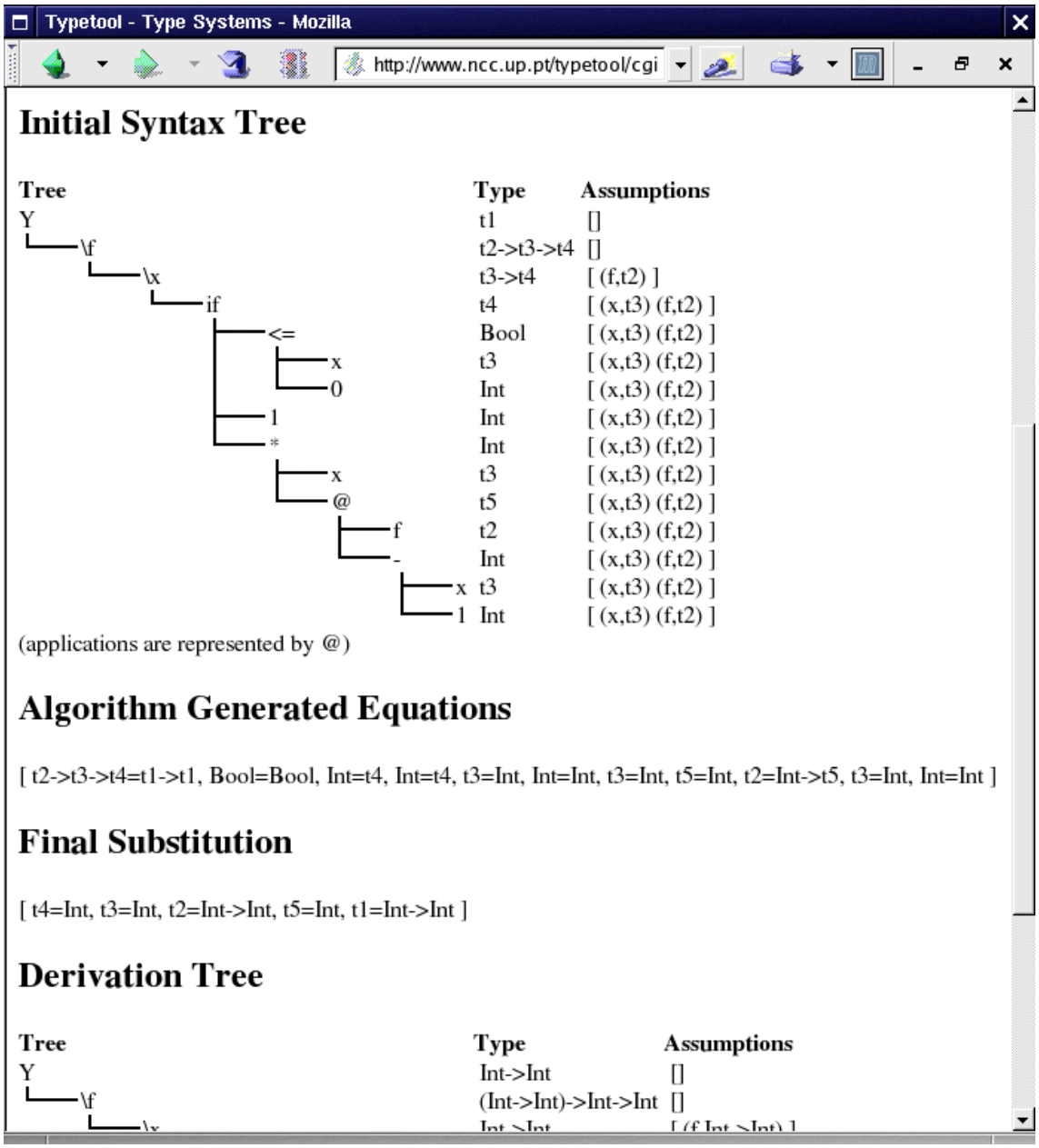


Figure 4. Generated equations and corresponding solutions

As another example consider example 1. In this case the result produced by TypeTool is presented in figure 6.

The example in figure 7 shows the case of a term which is not typable in the Damas-Milner type system. This happens because when typing *x x* the algorithm tries to unify the type of a function with its argument.

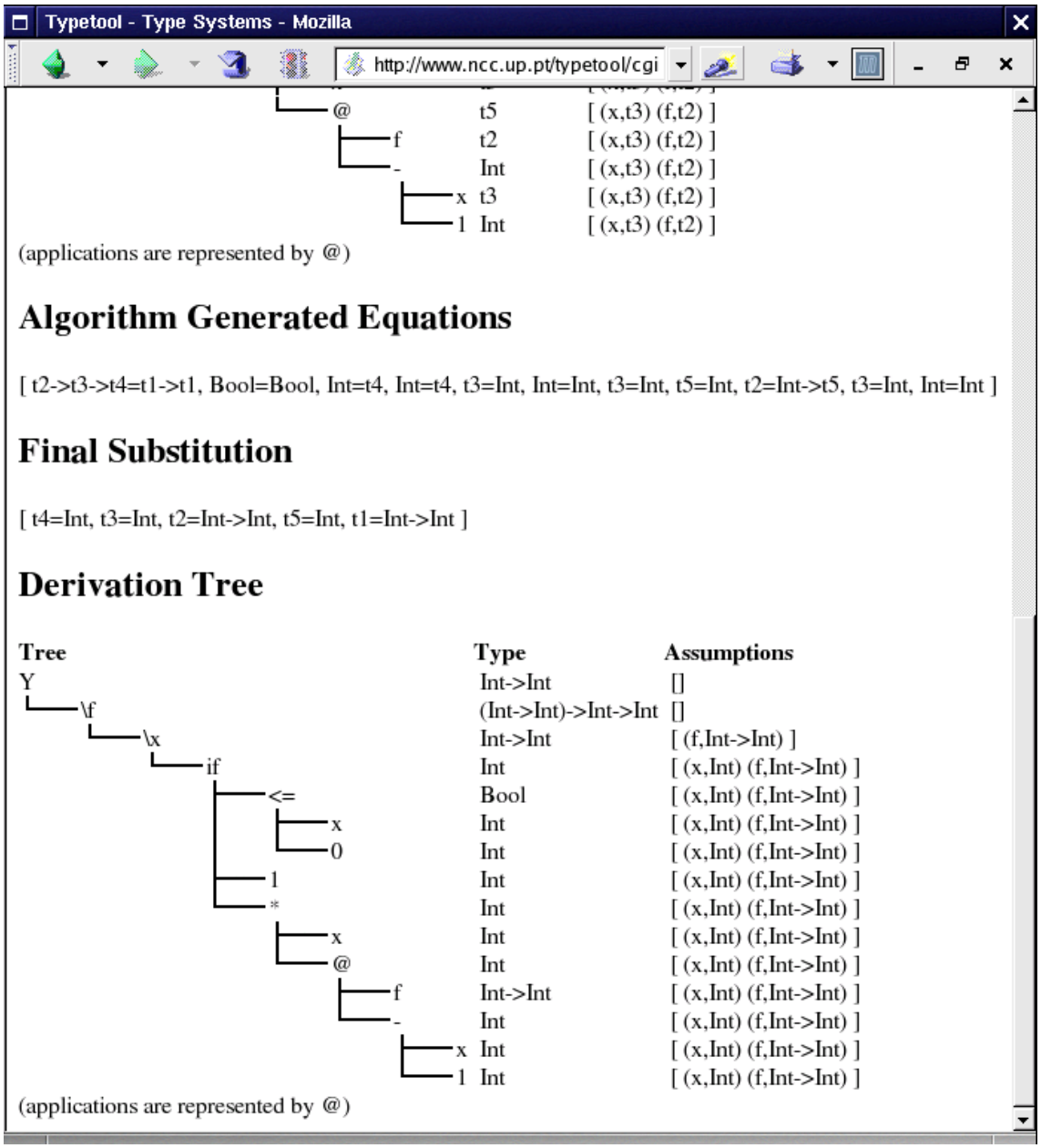


Figure 5. Type derivation

5 System's Architecture

TypeTool follows a typical CGI application architecture, in which a user sends a request (through a browser) to the Web server, which then executes a CGI script and redirects the script reply back to the user.

Before showing how the interaction between different requests/reply is performed, TypeTool application components are now briefly described.

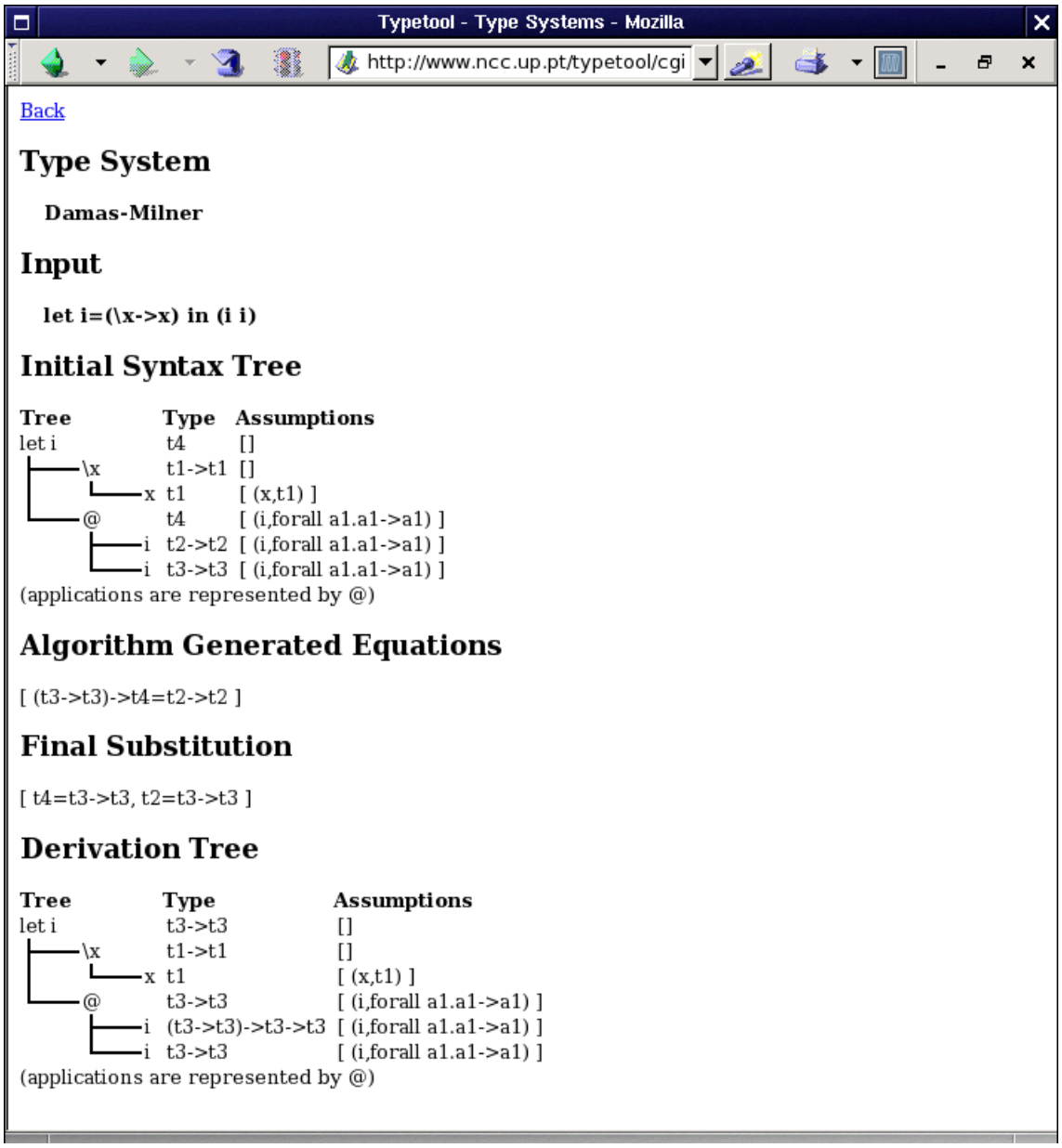


Figure 6. Example of type derivation for *let* $i = (\lambda x.x)$ *in* ii

5.1 TypeTool components

TypeTool is essentially composed by a central script and a set of modules, where each installed module is responsible for a type system. The central script, besides allowing the user to visualize the application state (installed modules and possible actions), is also an interface between the user and each module, making the interaction between this two parts. Each module replies to the central script inference requests.

5.2 Different kinds of user/TypeTool interaction

The user can perform the following requests to the TypeTool application: interface request and inference request.

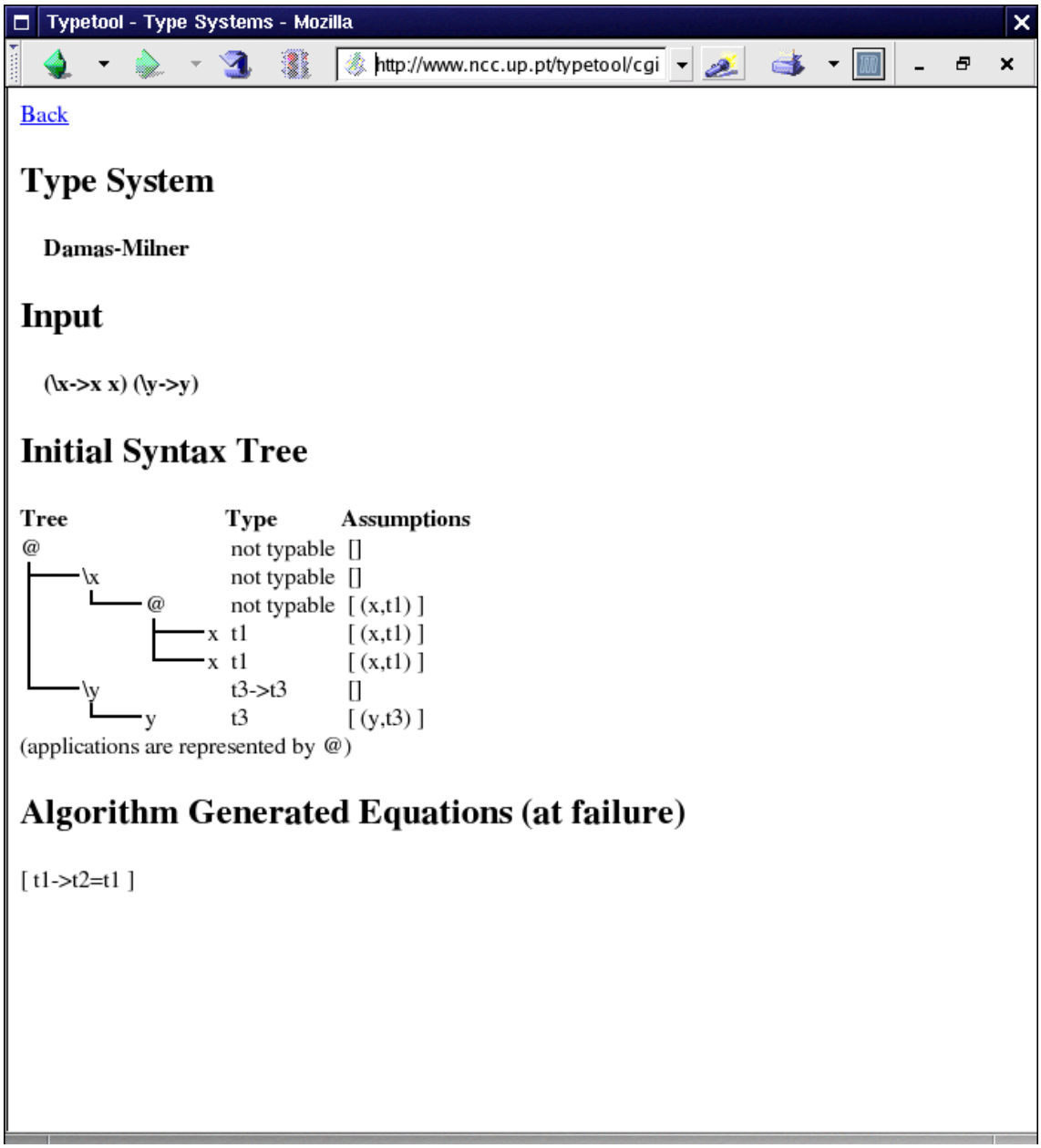


Figure 7. Example of a term for which type inference fails

Interface request This kind of request occurs when a user accesses the central script's URL, through a browser.

The central script is then in charge of gathering information about the installed type system modules. Based on this information, the script can now produce an HTML web page (interface) that allows the use of the application [see figure 8].

Inference request After selecting a type system and entering an expression (for which to infer a type), the user is then able to send the inference request to the server.

Then the central script receives the request and redirects the expression to the selected type system module [see figure 9].

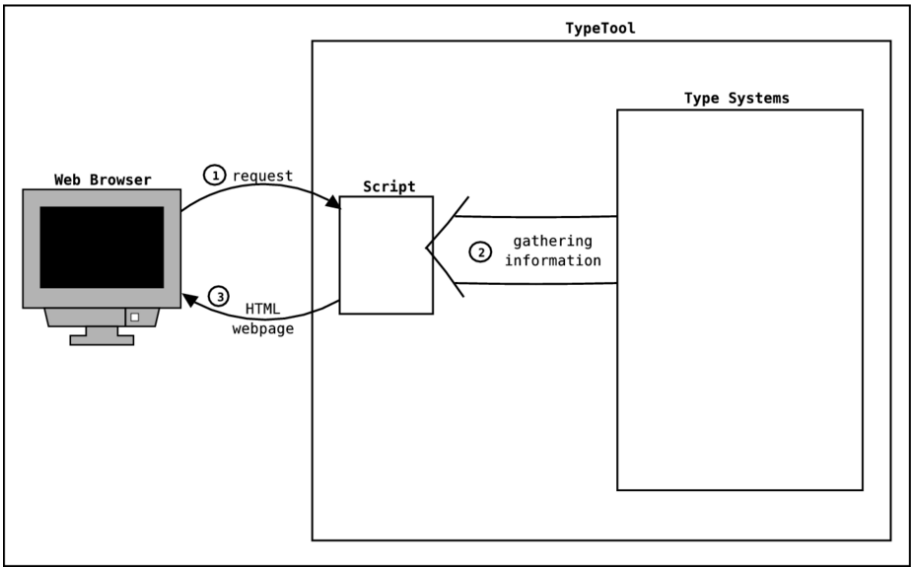


Figure 8. Creating TypeTool's graphical interface

The type system module starts by checking the expression syntax and semantics. When the inference process is concluded, the output is produced [see figure10].

6 Implementation

6.1 Choice of technologies and programming languages

CGI technology The choice of CGI technology for implementing TypeTool is two-fold. First, CGI is supported by every web server, without installing further software. Secondly, it is a simple technology offering vast possibilities, being perfectly suited to the development of the needed functionalities of a graphical interface accessible through a standard web browser.

Perl TypeTool's central script is written in Perl, taking advantage of this language. These advantages are especially relevant when Perl is used in the context of CGI applications (see [9] for more details).

Prolog and CHR For the inference processes of the installed type systems, the choice was Prolog with Constraint Handling Rules (CHR) [8], along the lines presented in [1].

CHR: CHR is a high-level language designed to write constraint solvers using rewriting. As a special purpose language, CHR "extends" a host language with (more) constraint solving capabilities.

A constraint is considered to be a distinguished, special first-order predicate (atomic formula) and there are two types of constraints. One for built-in (predefined) constraints and the other for CHR (user-defined) constraints. Built-in constraints are handled by a predefined constraint solver that already exists in the host language (note that we can consider host language statements as built-in constraints) and CHR constraints are those defined by a CHR program.

Definition 61 *A CHR program is a finite set of CHR rules. There are three kinds of CHR rules:*

- *A* simplification *is of the form* $H_1, ..., H_i$ <=> $G_1, ..., G_j \mid B_1, ..., B_k$

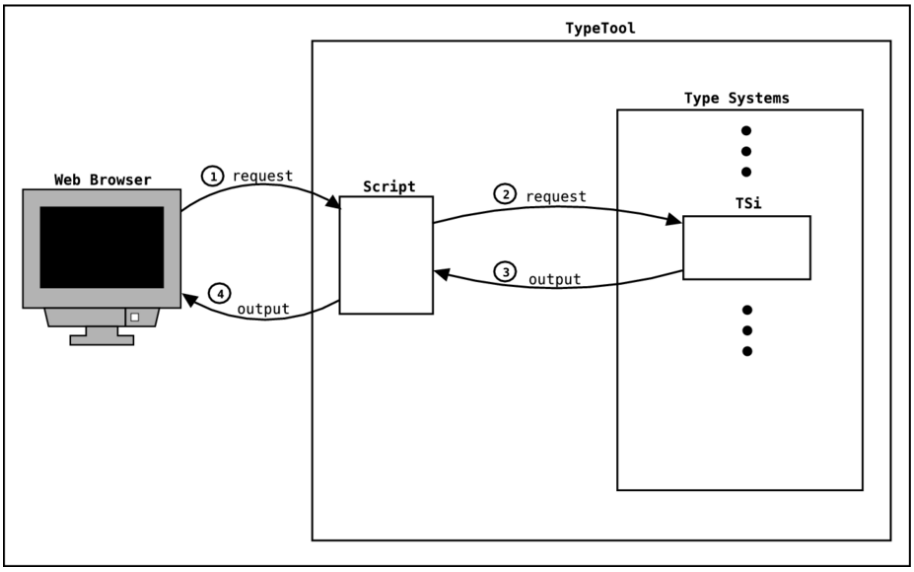


Figure 9. Inference request to a given type system

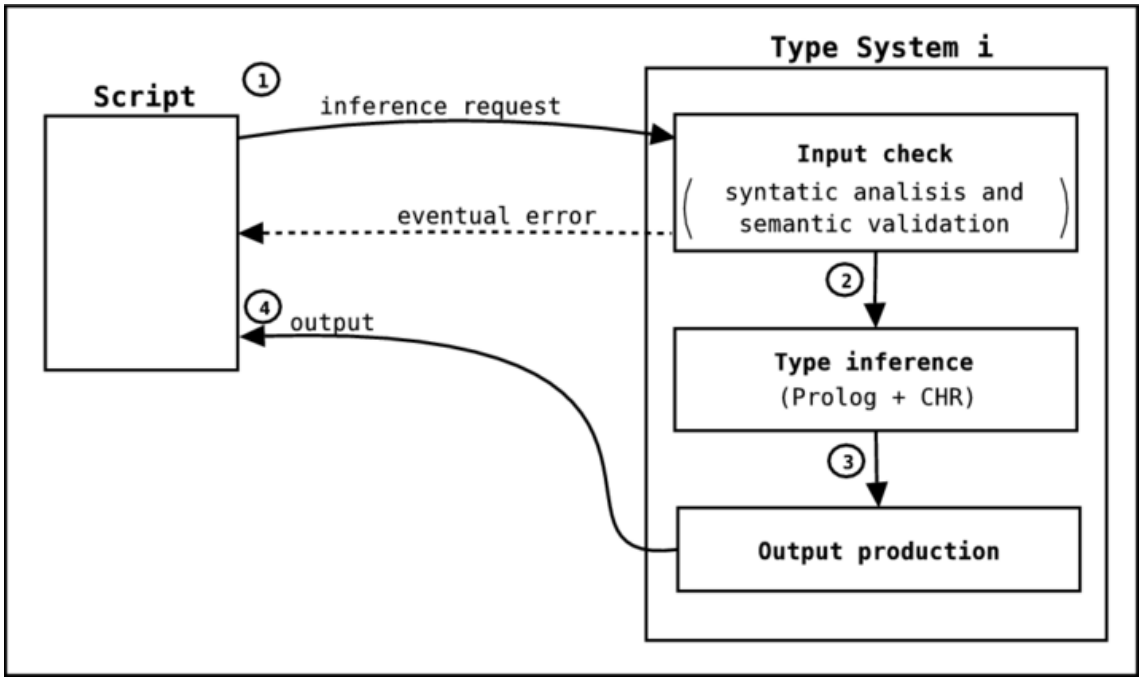


Figure 10. Execution flow of the inference process in detail

- *A* propagation *is of the form* $H_1, ..., H_i$ ==> $G_1, ..., G_j \mid B_1, ..., B_k$
- *A* simpagation *is of the form* $H_1, ..., H_l \setminus H_{l+1}, ..., H_i$ <=> $G_1, ..., G_j \mid B_1, ..., B_k$

with $i \geq 1$, $j \geq 0$, $k \geq 0$, $l \geq 1$ *and where the multi-head* $H_1, ..., H_i$ *is a nonempty sequence of CHR constraints, the guard* $G_1, ..., G_j$ *is a sequence of built-in constraints, and the body* $B_1, ..., B_k$ *is a sequence of built-in and CHR constraints.*

Empty sequences are represented by the built-in constraint true *and, for simplicity, the empty guard,* true*, can be removed from a rule together with the* '|' *operator.*

Declaratively, a rule relates heads and body provided the guard is true. A simplification rule means that the heads are true if and only if the body is true. A propagation rule means that the body is true if the heads are true. A simpagation rule combines a simplification and a propagation rule. The rule `Heads1 \ Heads2 <=> Body` is equivalent to the simplification rule
`Heads1, Heads2 <=> Body, Heads1`. More information can be found in [8].

Due to the declarative character of the algorithms used in the type systems inference process and their need for an explicit use of the unification algorithm, the choice of CHR occurred naturally, given the clear and elegant way of implementing the unification algorithm on this constraint solver. As an example, the CHR implementation of unification is presented in the following lines (`var(X)` denotes a type variable, `X -> Y` an "arrow" type and `T1 = T2` an equation to be solved by unification):

```
T1 -> T2 = T3 -> T4  <=> T1 = T3, T2 = T4.
var(X) = var(X)      <=> true.
T1 -> T2 = var(X)    <=> var(X) = T1 -> T2.
var(X) = T1 -> T2    <=> occurs(var(X), T1 -> T2) | fail.
var(X) = T \ T1 = T2 <=> occurs(var(X), T1 -> T2) |
                         replace(var(X), T, T1, NT1),
                         replace(var(X), T, T2, NT2),
                         NT1 = NT2.
```

In the CHR rules presented above the last rule applies the substitution *var(X) = T* to every constraint where *var(X)* occurs. Note that the 4th argument of the predicate "replace" is the result of replacing, in the 3rd argument, the 1st by the 2nd argument.

Prolog: The declarative character, mentioned above, supports the use of a declarative language for the implementation of the type systems inference algorithms. The choice of Prolog in particular (and not a functional language) was made basically because now CHR is more stable linked to Prolog.

7 Conclusion

We presented a lightweight web-based type inference visualization tool. The user can follow the type inference algorithm for any given expression. TypeTool was first designed for students, but it has grown past its original goals and it is being used by professional programmers which want to understand in more detail the type systems of the most common functional programming languages. Since it is freely available on the Web, TypeTool had about 1500 visits and we had a lot of feedback which enabled us to improve the tool.

This system works for the Simple Type System, the basis of every type system for functional languages, and the polymorphic type system of a core ML. In the future we intend to extend the system to deal with Haskell type classes and to the full syntax of Haskell and ML.

Acknowledgements The authors want to thank Sandra Alves and Luis Damas for several useful suggestions related with this work. The work presented in this paper has been partially supported by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*.

References

1. Sandra Alves and Mário Florido. Type inference using constraint handling rules. *Electronic Notes in Theoretical Computer Science*, 64, 2002.
2. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
3. Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 193–204, 2001.
4. H. B. Curry. Functionality in combinatory logic. In *National Academy of Sciences of the U.S.A.*, volume 20, pages 584–590, 1934.
5. L. Damas and R. Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
6. Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.
7. Martin Erwig. Visual type inference. Draft paper, 2003.
8. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
9. Scott Guelich, Shishir Gundavaram, and Gunther Birznieks. *CGI Programming with Perl*. O'Reilly, second edition, 2000.
10. Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Programming Languages and Systems, 12th European Symposium on Programming*, volume 2618 of *LNCS*, pages 284–301, April 2003.
11. Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3–13, Uppsala, Sweden, 2003. ACM Press.
12. F. Huch, O. Chitil, and A. Simon. Typeview: a tool for understanding type errors. In M. Mohnen and P. Koopman, editors, *12th International Workshop on Implementation of Functional Languages*, Aachner Informatik-Berichte, pages 63–69, 2000.
13. Yang Jung and Greg Michaelson. A visualisation of polymorphic type checking. *Journal of Functional Programming*, 10(1):57–75, January 2000.
14. Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.
15. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
16. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.
17. Helen Soosaipillai. An explanation based polymorphic type checker for standard ml. Master's thesis, Heriot-Watt University, Edinburgh, Scotland, 1990.
18. System I Experimentation Tool. http://types.bu.edu/modular/compositional/system-i/.
19. J. A. Walz and G. F. Johnson. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 44–57, January 1986.
20. Mitchell Wand. Finding the source of type errors. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 38–43, January 1986.

Dynamic Predicates in Functional Logic Programs*

Michael Hanus

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

Abstract In this paper we propose a new concept to deal with dynamic predicates in functional logic programs. The definition of a dynamic predicate can change over time, i.e., one can add or remove facts that define this predicate. Our approach is easy to use and has a clear semantics that does not depend on the particular (demand-driven) evaluation strategy of the underlying implementation. In particular, the concept is not based on (unsafe) side effects so that the order of evaluation does not influence the computed results—an essential requirement in non-strict languages.

Dynamic predicates can also be persistent so that their definitions are saved across invocations of programs. Thus, dynamic predicates are a lightweight alternative to the explicit use of external database systems that can be applied in many applications. Moreover, they extend one of the classical application areas of logic programming to functional logic programs. We present the concept, its use and an implementation in a Prolog-based compiler.

1 Introduction and Related Work

Functional logic languages aim to integrate the best features of functional and logic languages in order to provide a variety of programming concepts to the programmer. For instance, the concepts of demand-driven evaluation, higher-order functions, polymorphic typing from functional programming can be combined with logic programming features like computing with partial information (logical variables), constraint solving and non-deterministic search for solutions. This combination leads to optimal evaluation strategies [2] and new design patterns [4] that can be applied to provide better programming abstractions, e.g., for implementing graphical user interfaces [11] or programming dynamic web pages [12].

However, one of the traditional application areas of logic programming is not yet sufficiently covered in existing functional logic languages: the combination of declarative programs with persistent information, usually stored in relational databases, that can change over time. Logic programming provides a natural framework for this combination (e.g., see [7,9]) since externally stored relations can be considered as facts defining a predicate of a logic program. Thus, logic programming is an appropriate approach to deal with deductive databases or declarative knowledge management. In this paper, we propose a similar concept for functional logic languages. Nevertheless, this is not just an adaptation of existing concepts to functional logic programming. We will show that the addition of advanced functional programming concepts, like the clean separation of imperative and declarative computations by the use of monads [24], provides a better handling of the dynamic behavior of database predicates, i.e., when we change the definition of such predicates by adding or removing facts. To motivate our approach, we shortly discuss the problems caused by traditional logic programming approaches to dynamic predicates.

The logic programming language Prolog allows to change the definition of predicates[1] by adding or deleting clauses using predefined predicates like `asserta` (adding a new *first* clause), `assertz` (adding a new *last* clause), or `retract` (deleting a matching clause). Problems occur if the use of these predicates is mixed with their update. For instance, if a new clause is added during the evaluation of a literal, it is not directly clear whether this new clause should be visible during

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2.

[1] In many Prolog systems, such predicates must be declared as "dynamic" in order to change their definitions dynamically.

backtracking, i.e., a new proof attempt for the same literal. This has been discussed in [18] where a so-called "logical view" of database updates is proposed. In the logical view, only the clauses that exist at the first proof attempt to a literal are used. Although this solves the problems related to backtracking, advanced evaluation strategies cause new problems.

It is well known that advanced control rules, like coroutining, provide a better control behavior w.r.t. the termination and efficiency of logic programs [21]. Although the completeness of SLD resolution w.r.t. any selection rule seems to justify such advanced control rules, it is not the case w.r.t. dynamic predicates. For instance, consider the Prolog program

```
ap(X) :- assertz(p(X)).

q :- ap(X), p(Y), X=1.
```

If there are no clauses for the dynamic predicate `p`, the proof of the literal `q` succeeds due to the left-to-right evaluation of the body of the clause for `q`. However, if we add the block declaration (in Sicstus-Prolog) "`:- block ap(-).`" to specify that `ap` should be executed only if its argument is not a free variable, then the proof of the literal `q` fails, because the clause for `p` has not been asserted when `p(Y)` should be proved.

This example indicates that care is needed when combining dynamic predicates and advanced control strategies. This is even more important in functional logic languages that are usually based on demand-driven (and concurrent) evaluation strategies where the exact order of evaluation is difficult to determine in advance [2,10].

Unfortunately, existing approaches to deal with dynamic predicates do not help here. For instance, Prolog and its extensions to persistent predicates stored in databases, like the Berkeley DB of Sicstus-Prolog or the persistence module of Ciao Prolog [6], suffer from the same problems. In the other hand, functional language bindings to databases do not offer the constraint solving and search facilities of logic languages. For instance, HaSQL[2] supports a simple connection to relational databases via I/O actions but provides no abstraction for computing queries (the programmer has to write SQL queries in plain text). This is improved in Haskell/DB [17] which allows to express queries through the use of specific operators. More complex information must be deduced by defining appropriate functions.

Other approaches to integrate functional logic programs with databases concentrate only on the semantical model for query languages. For instance, [1] proposes an integration of functional logic programming and relational databases by an extended data model and relational calculus. However, the problem of database updates is not considered and an implementation is not provided. Echahed and Serwe [8] propose a general framework for functional logic programming with processes and updates on clauses. Since they allow updates on arbitrary program clauses (rather than facts), it is unclear how to achieve an efficient implementation of this general model. Moreover, persistence is not covered in their approach.

Since real applications require the access and manipulation of persistent data, we propose a new model to deal with dynamic predicates in functional logic programs where we choose the declarative multi-paradigm language Curry [16] for concrete examples.[3] Although the basic idea is motivated by existing approaches (a dynamic predicate is considered as defined by a set of basic facts that can be externally stored), we propose a clear distinction between the accesses and updates to a dynamic predicate. In order to abstract from the concrete (demand-driven) evaluation strategy, we propose the use of time stamps to mark the lifetime of individual facts.

[2] `http://members.tripod.com/~sproot/hasql.htm`

[3] Our proposal can be adapted to other modern functional logic languages that are based on the monadic I/O concept to integrate imperative and declarative computations in a clean manner, like Escher [19], Mercury [23], or Toy [20].

Dynamic predicates can also be persistent so that their definitions are saved across invocations of programs. Thus, our approach to dynamic predicates is a lightweight alternative to the use of external database systems that can be easily used in many applications. Nevertheless, one can also use an external database if the size of the dynamic predicate definitions becomes too large.

The next section provides a basic introduction into Curry. Section 3 contains a description of our proposal to integrate dynamic predicates into functional logic languages. Section 4 sketches a concrete implementation of this concept and Section 5 contains our conclusions.

2 Basic Elements of Curry

In this section we review those elements of Curry which are necessary to understand the contents of this paper. More details about Curry's computation model and a complete description of all language features can be found in [10,16].

Curry is a modern multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming and supporting programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [22], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

A Curry *program* consists of the definition of functions and data types on which the functions operate. Functions are evaluated lazily. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. The behavior of function calls with free variables depends on the evaluation mode of functions which can be either *flexible* or *rigid*. Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule ("*narrowing*"). Calls to rigid functions are suspended if a demanded argument is uninstantiated ("*residuation*").

Example 1. The following Curry program defines the data types of Boolean values, "possible" (maybe) values, and polymorphic lists (first three lines) and functions for computing the concatenation of lists and the last element of a list:

```
data Bool    = True    | False
data Maybe a = Nothing | Just a
data List a  = []      | a : List a

conc :: [a] -> [a] -> [a]
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last :: [a] -> a
last xs | conc ys [x] =:= xs   = x   where x,ys free
```

The data type declarations define `True` and `False` as the Boolean constants, `Nothing` and `Just` as the constructors for possible values (where `Nothing` is considered as no value), and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell).

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[4] Since `conc` is flexible,[5] the equation "`conc ys [x] =:= xs`" is solved by instantiating the first argument `ys` to the list `xs` without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

In general, functions are defined by (*conditional*) *rules* of the form

$$f\, t_1 \ldots t_n \;\texttt{|}\; c \;\texttt{=}\; e \quad \texttt{where}\; vs \;\texttt{free}$$

with f being a function, $t_1, \ldots, t_n$ *patterns* (i.e., expressions without defined functions) without multiple occurrences of a variable, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, lambda abstractions etc, and vs is the list of *free variables* that occur in c and e but not in $t_1, \ldots, t_n$. The condition and the `where` parts can be omitted if c and vs are empty, respectively. The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable.

A *constraint* is any expression of the built-in type `Success`. For instance, the trivial constraint `success` is an expression of type `Success` that denotes the always satisfiable constraint. "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2, i.e., this expression is evaluated by proving both argument constraints concurrently. Each Curry system provides at least *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable patterns. However, specific Curry systems can also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [13].

Predicates in the sense of logic programming can be considered as functions with result type `Success`. For instance, a predicate `isPrime` that is satisfied if the argument (an integer number) is a prime can be modeled as a function with type

```
isPrime :: Int -> Success
```

The following rules define a few facts for this predicate:

```
isPrime 2 = success
isPrime 3 = success
isPrime 5 = success
isPrime 7 = success
```

Apart from syntactic differences (that support, in contrast to pure logic programming, the use of predicates and partial applications of predicates as first-class citizens in higher-order functions), any pure logic program have a direct correspondence to a Curry program. For instance, a predicate `isPrimePair` that is satisfied if the arguments are primes that differ by 2 can be defined as follows:

```
isPrimePair :: Int -> Int -> Success
isPrimePair x y = isPrime x & isPrime y & x+2 =:= y
```

The operational semantics of Curry, precisely described in [10,16], is based on an optimal evaluation strategy [2] which is a conservative extension of lazy functional programming and (concurrent)

[4] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[5] As a default, all functions except for I/O actions and external functions are flexible.

logic programming. Due to its demand-driven behavior, it provides optimal evaluation (e.g., shortest derivation sequences, minimal solution sets) on well-defined classes of programs (see [2] for details). Curry also offers the standard features of functional languages, like higher-order functions (e.g., "\x -> e" denotes an anonymous function that assigns to each x the value of e) or monadic I/O. Since the latter is important for the ideas in this paper, we sketch the I/O concept of Curry which is almost identical to the monadic I/O of Haskell [24].

In the monadic approach to I/O, an interactive program is considered as a function computing a sequence of actions which are applied to the outside world. An *action* changes the state of the world and possibly returns a result (e.g., a character read from the terminal). Thus, actions are functions of type

$$World \texttt{ -> (}\alpha\texttt{,}World\texttt{)}$$

(where $World$ denotes the type of all states of the outside world). This function type is also abbreviated by `IO` α. If an action of type `IO` α is applied to a particular world, it yields a value of type α and a new (changed) world. For instance, `getChar` of type `IO Char` is an action which reads a character from the standard input whenever it is executed, i.e., applied to a world. The important point is that values of type $World$ are not accessible to the programmer—she/he can only create and compose actions on the world. For instance, the action `getChar` can be composed with the action `putChar` (which has type `Char -> IO ()` and writes a character to the terminal) by the sequential composition operator `>>=` (which has type `IO` α `->` (`(`α `->` `IO` β`)` `->` `IO` β), i.e., "`getChar >>= putChar`" is a composed action which prints the next character of the input stream on the screen. The second composition operator `>>` is like `>>=` but ignores the result of the first action. Furthermore, `done` is the "empty" action which does nothing (see [24] for more details). For instance, a function which takes a string (list of characters) and produces an action that prints it to the terminal followed by a line feed can be defined as follows:

```
putStrLn []     = putChar '\n'
putStrLn (c:cs) = putChar c >> putStrLn cs
```

It should be noted that an action is executed when the program (applied to the world) is executed. Since the world cannot be copied, non-deterministic actions as a result of a program are not allowed. Therefore, all possible search must be encapsulated between I/O operations using the features for encapsulating search [5,15].

3 Dynamic Predicates

In this section we describe our proposal to dynamic predicates in functional logic programs and show its use by several examples.

3.1 General Concept

Since the definition of dynamic predicates is also intended to be stored persistently in files, we assume that dynamic predicates are defined by ground (i.e., variable-free) facts. However, in contrast to predicates that are explicitly defined in a program (e.g., `isPrime` in Section 2), the definition of *dynamic* predicates is not provided in the program code but will be dynamically computed. Thus, dynamic predicates are similar to "external" functions whose code is not contained in the program but defined elsewhere. Therefore, the programmer has to specify in a program only the (monomorphic) type signature of a dynamic predicate (remember that Curry is strongly typed) and mark its name as "`dynamic`".

As a simple example, we want to define a dynamic predicate `prime` to store prime numbers whenever we compute them. Thus, we provide the following definition in our program:

```
prime :: Int -> Dynamic
prime dynamic
```

Similarly to `Success`, the predefined type "`Dynamic`" is abstract, i.e., there are no accessible data constructors of this type but a few predefined operations that act on objects of this type (see below). From a declarative point of view, `Dynamic` is similar to `Success`, i.e., `prime` can be considered as a predicate. However, since the definition of dynamic predicates may change over time, the access to dynamic predicates is restricted in order to avoid the problems mentioned in Section 1. Thus, the use of the type `Dynamic` ensures that the specific access and update operations (see below) can be applied only to dynamic predicates. Furthermore, the keyword "`dynamic`" informs the compiler that the code for `prime` is not in the program but externally stored (similarly to the definition of external functions).

In order to avoid the problems related to mixing update and access to dynamic predicates, we put the corresponding operations into the I/O monad since this ensures a sequential evaluation order. Thus, we provide the following predefined operations:

```
assert :: Dynamic -> IO ()

retract :: Dynamic -> IO Bool

getKnowledge :: IO (Dynamic -> Success)
```

`assert` adds a new fact about a dynamic predicate to the database where the *database* is considered as the set of all known facts for dynamic predicates. Actually, the database can also contain multiple entries (if the same fact is repeatedly asserted) so that the database is a multi-set of facts. For the sake of simplicity, we ignore this detail and talk about sets in the following.

Since the facts defining dynamic predicates should not contain unbound variables,[6] `assert` is a rigid function, i.e., it suspends when the arguments (after evaluation to normal form) contain unbound variables. Similarly, `retract` is also rigid[7] and removes a matching fact, if possible (this is indicated by the Boolean result value). For instance, the sequence of actions

```
assert (prime 1) >> assert (prime 2) >> retract (prime 1)
```

asserts the new fact `(prime 2)` to the database.

The action `getKnowledge` is intended to get the set of facts stored in the database at the time when this action is executed. In order to provide access to the set of facts, `getKnowledge` returns a function of type "`Dynamic -> Success`" which can be applied to expressions of type "`Dynamic`", i.e., calls to dynamic predicates. For instance, the following sequence of actions asserts a new fact `(prime 2)` and retrieves its contents by unifying the logical variable `x` with the value 2:[8]

```
assert (prime 2) >> getKnowledge >>= \known ->
                    doSolve (known (prime x))
```

[6] Since the definition of a dynamic predicate could also be stored in files or external databases (see Section 3.2), this restriction is reasonable.

[7] One could argue that `retract` could be also called with logical variables which should be bound to the values of the retracted facts; however, this might cause non-deterministic actions (if more than one fact matches) which leads to run-time errors.

[8] The action `doSolve` is defined as "`doSolve c | c = done`" and can be used to embed constraint solving into the I/O monad.

Since writing monadic sequences of I/O actions is not well readable, we will use Haskell's "do" notation [22] in the following. Thus, we write the previous action sequence in the following form:

```
do assert (prime 2)
   known <- getKnowledge
   doSolve (known (prime x))
```

Since there might be several facts that match a call to a dynamic predicate, we have to encapsulate the possible non-determinism occurring in a logic computation. This can be done in Curry by the primitive action to encapsulate the search for all solutions to a goal:

```
getAllSolutions :: (a -> Success) -> IO [a]
```

`getAllSolutions` takes a constraint abstraction and returns the list of all solutions, i.e., all values for the argument of the abstraction such that the constraint is satisfiable.[9] For instance, the evaluation of

```
getAllSolutions (\x -> known (prime x))
```

returns the list of all values for `x` such that `known (prime x)` is satisfied. Thus, we can define a function `printKnownPrimes` that prints the list of all known prime numbers as follows:

```
printKnownPrimes = do
   known <- getKnowledge
   sols <- getAllSolutions (\x -> known (prime x))
   print sols
```

If we just want to check whether a particular fact of a dynamic predicate is known, we can define the following general function:

```
isKnown :: Dynamic -> IO Bool
isKnown p = do
   known <- getKnowledge
   sols <- getAllSolutions (\_ -> known p)
   return (sols /= [])
```

Here we are not interested in individual solutions. Thus, we write the anonymous variable "`_`" as the argument to the search goal and finally check whether some solution has been computed.

Note that we can use all logic programming techniques also for dynamic predicates: we just have to pass the result of `getKnowledge` (i.e., the variable `known` above) into the clauses defining the deductive part of the database program and wrap all calls to a dynamic predicate with this result variable. For instance, we can print all prime pairs by the following definitions:

```
primePair known (x,y) =
         known (prime x) & known (prime y) & x+2 =:= y

printPrimePairs = do
   known <- getKnowledge
   sols <- getAllSolutions (\p -> primePair known p)
   print sols
```

[9] `getAllSolutions` is an I/O action since the order of the result list might vary from time to time due to the order of non-deterministic evaluations.

The constraint `primePair` specifies the property of being a prime pair w.r.t. the knowledge `known`, and the action `printPrimePairs` prints all currently known prime pairs.

If one wants to avoid passing the variable `known` through all predicates that do inferences on the current knowledge, one can also define these predicates locally so that `known` becomes automatically visible to all predicates. In order to write the program code even more in the logic programming style, we define the composition of `known` and `prime` as a single name. The following code, which defines a constraint for sequences of primes, shows an example for this "LP" style (" . " denotes function composition):

```
primeSequence known l = primes l

 where

  isPrime = known . prime

  primes [p] = isPrime p
  primes (p1:p2:ps) = isPrime p1 &
                      isPrime p2 &
                      (p1<p2) =:= True &
                      primes (p2:ps)
```

Our concept provides a clean separation between database updates and accesses. Since we get the knowledge at a particular point of time, we can access all facts independent on the order of evaluation. Actually, the order is difficult to determine due to the demand-driven evaluation strategy. For instance, consider the following sequence of actions:

```
do assert (prime 2)
   known1 <- getKnowledge
   assert (prime 3)
   assert (prime 5)
   known2 <- getKnowledge
   sols1 <- getAllSolutions (\x -> known1 (prime x))
   sols2 <- getAllSolutions (\x -> known2 (prime x))
   return (sols1,sols2)
```

Executing this with the empty database, the pair of lists `([2],[2,3,5])` is returned. Although the concrete computation of all solutions is performed later than they are conceptually accessed (by `getKnowledge`) in the program text, we get the right facts (in contrast to Prolog with coroutining, see Section 1). Therefore, `getKnowledge` conceptually copies the current database for later access. However, since an actual copy of the database can be quite large, this is implemented by the use of time stamps (see Section 4).

3.2 Persistent Dynamic Predicates

One of the key features of our proposal is the easy handling of persistent data. The facts about dynamic predicates are usually stored in main memory which supports fast access. However, in most applications it is necessary to store the data also persistently so that the actual definitions of dynamic predicates survive different executions (or crashes) of the program. One approach is to store the facts in relational databases (which is non-trivial since we allow arbitrary term structures as arguments). Another alternative is to store them in files (e.g., in XML format). In both cases the programmer has to consider the right format and access routines for each application. Our approach

is much simpler (and often also more efficient if the size of the dynamic data is not extremely large): it is only necessary to declare the predicate as "persistent". For instance, if we want to store our knowledge about primes persistently, we define the predicate `prime` as follows:

```
prime :: Int -> Dynamic
prime persistent "file:prime_infos"
```

Here, `prime_infos` is the name of a directory where the run-time system automatically puts all files containing information about the dynamic predicate `prime`.[10] Apart from changing the `dynamic` declaration into a `persistent` declaration, nothing else needs to be changed in our program, i.e., the same actions like `assert`, `retract`, or `getKnowledge` can be used to change or access the persistent facts of `prime`. Nevertheless, the persistent declaration has important consequences:

- All facts and their changes are persistently stored, i.e., after a termination (or crash) and restart of the program, all facts are recovered.
- Changes to dynamic predicates are immediately written into a log file so that they can be recovered.
- `getKnowledge` gets always the current knowledge persistently stored, i.e., if other processes also change the facts of the same predicate, it becomes immediately visible with the next call to `getKnowledge`.
- In order to avoid conflicts between parallel processes working on the same dynamic predicates, there is also a transaction concept (see Section 3.3).

Note that the easy and clean addition of persistency was made possible due to our concept to separate the update and access to dynamic predicates. Since updates are put into the I/O monad, there are obvious points where changes must be logged. On the other hand, the `getKnowledge` action needs only a (usually short) synchronization with the external data and then the knowledge can be used with the efficiency of the internal program execution.

3.3 Transactions

The persistent storage of dynamic predicates causes another problem: if several processes running in parallel updates the same data, some synchronization is necessary. Since we intend to use our proposal also for web applications [12], there is a clear need to solve the synchronization problem since in such applications one does not know when the individual programs reacting to client's requests are executed. Fortunately, the database community has solved this problem via transaction models so that we only have to adapt them into our framework of functional logic programming.

We consider a *transaction* as a sequence of changes to (possibly several different) dynamic predicates that should only be performed together or completely ignored. Moreover, the changes of a transaction become visible to other parallel processes only if the complete transaction has been successfully executed. This model can be easily supported by providing two I/O actions:

```
transaction :: IO a -> IO (Maybe a)

abortTransaction :: IO a
```

`transaction` takes an I/O action (usually, a sequence of updates to dynamic predicates) as argument and tries to execute it. If this was successfully done, the result r of the argument action is

[10] The prefix "`file:`" instructs the compiler to use a file-based implementation of persistent predicates. For future work, it is planned also to use relational databases to store persistent facts so that this prefix is used to distinguish the different access methods.

returned as `(Just` r`)` and all changes to the dynamic predicates become visible to other processes. Otherwise, i.e., in case of a failure, run-time error, or if the action `abortTransaction` has been executed, all changes to the dynamic predicates performed during this transaction are undone and `Nothing` is returned to indicate the failure of the transaction. For instance, consider the following transaction:

```
try42 = do assert (prime 42)
           abortTransaction
           assert (prime 43)
```

If we execute "`transaction try42`", then no change to the definition of the persistent dynamic predicate `prime` becomes visible.

4 Implementation

In order to test our concept and to provide a reasonable implementation, we have implemented it in the PAKCS implementation of Curry [13]. This implementation is fairly efficient and has been used for many non-trivial applications, e.g., a web-based system for e-learning [14]. The system compiles Curry programs into Prolog by transforming pattern matching into predicates and exploiting coroutining for the implementation of the concurrency features of Curry [3]. Due to the use of Prolog as the back-end language, the implementation of our concept is not very difficult. Therefore, we highlight only a few aspects of this implementation.

First of all, the compiler of PAKCS has to be adapted since the code for dynamic predicates must be different from other functions. Thus, the compiler translates a declaration of a dynamic predicate into specific code so that the run-time evaluation of a call to a dynamic predicate yields a data structure containing information about the actual arguments and the name of the external database (in case of persistent predicates). In this implementation, we have not used a relational database for storing the facts since this is not necessary for the size of the dynamic data (in our applications only a few megabytes). Instead, all facts are stored in main memory and in files in case of persistent predicates. First, we describe the implementation of non-persistent predicates.

Each `assert` and `retract` action is implemented via Prolog's `assert` and `retract`. However, as additional arguments we use time stamps to store the lifetime (birth and death) of all facts in order to implement the visibility of facts for the `getKnowledge` action (similarly to [18]). Thus, there is a global clock ("update counter") in the program that is incremented for each `assert` and `retract`. If a fact is asserted, it gets the actual time as birth time and ∞ as the death time. If a fact is retracted, it is not retracted in memory but only the death time is set to the actual time since there might be some unevaluated expression for which this fact is still visible. `getKnowledge` is implemented by returning a predefined function that keeps the current time as an argument. If this function is applied to some dynamic predicate, it unifies the predicate with all facts and, in case of a successful unification, it checks whether the time of the `getKnowledge` call is in the birth/death interval of this fact.

Persistent predicates are similarly implemented, i.e., all known facts are always kept in main memory. However, each update to a persistent predicate is written into a log file. Furthermore, all facts of this predicate are stored in a file in Prolog format. This file is only read and updated during the initialization time of the program, where the following operations are performed:

1. The previous database file with all Prolog facts is read.
2. All changes from the log file are replayed, i.e., executed.
3. A new version of the database file is written.

4. The log file is cleared.

In order to avoid problems in case of program crashes during this critical period, the initialization phase is made exclusive to one process via operating system locks and backup files are written.

To reduce the time to load the database, we store it also in an intermediate format (Prolog object file format of Sicstus-Prolog). With this binary format, the database for most applications can be loaded very efficiently. For instance, it needs 120 milliseconds to load a database of 12.5 MB Prolog source code on a 2.0 GHz Linux-PC (AMD Athlon XP 2600 with 256 KB cache).

In order to implement transactions and the concurrent access to persistent data, operating system locks are used. Moreover, version numbers of the database are stored in order to inform the running program about changes to the database by other processes. These changes are taken into account when `getKnowledge` is executed. Transactions are implemented by writing marks into the log files and considering only complete transactions when recovering the database in the initialization phase described above.

5 Conclusions

We have proposed a new approach to deal with dynamic predicates in functional logic programs. It is based on the idea to separate the update and access to dynamic predicates. Updates can only be performed on the top-level in the I/O monad in order to ensure a well-defined sequence of updates. The access to dynamic predicates is initiated also in the I/O monad in order to get a well-defined set of visible facts for dynamic predicates. However, the actual access can be done at any execution time since the visibility of facts is controlled by time stamps. This is important in the presence of an advanced operational semantics (demand-driven evaluation) where the actual sequence of evaluation steps is difficult to determine in advance.

Furthermore, dynamic predicates can be also persistent so that their definitions are externally stored and recovered when programs are restarted. This persistence model is also supported by a transaction concept in order to provide the parallel execution of processes working on the same data. We have sketched an implementation of this concept in a Prolog-based compiler.

Although the use of our concept is quite simple (one has to learn only three basic I/O actions), it is quite powerful at the same time since the applications of logic programming to declarative knowledge management can be directly implemented with this concept. We have used this concept in practice to implement a bibliographic database system and obtained quite satisfying results. The loading of the database containing almost 10,000 bibliographic entries needs only a few milliseconds, and querying all facts is also performed in milliseconds due to the fact that they are stored in main memory.

For future work, we want to test this concept in larger applications. Furthermore, we intend to implement this concept by the use of a relational database instead of the current file-based implementation. In this case, it might be interesting to extend the set of access primitives in order to combine them into larger queries that can be directly solved by the database system.

References

1. J.M. Almendros-Jiménez and A. Becerra-Terón. A Safe Relational Calculus for Functional Logic Deductive Databases. *Electronic Notes in Theoretical Computer Science*, Vol. 86, No. 3, 2003.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
3. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pp. 171–185. Springer LNCS 1794, 2000.

4. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.
5. B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. In *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)*, 2004.
6. J. Correas, J.M. Gómez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Model for Persistence in CLP Systems. Technical Report CLIP3/2003.0, Technical University of Madrid, 2003.
7. S.K. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.
8. R. Echahed and W. Serwe. Combining Mobile Processes and Declarative Programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pp. 300–314. Springer LNAI 1861, 2000.
9. H. Gallaire and J. Minker, editors. *Logic and Databases*, New York, 1978. Plenum Press.
10. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
11. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
12. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
13. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2003.
14. M. Hanus and F. Huch. An Open System to Support Web-based Learning. In *Proc. 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pp. 269–282. Technical Report DSIC-II/13/03, Universidad Politécnica de Valencia, 2003.
15. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
16. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
17. D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)*, pp. 109–122. ACM SIGPLAN Notices 35(1), 1999.
18. T.G. Lindholm and R.A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 21–39. MIT Press, 1987.
19. J. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, No. 3, pp. 1–49, 1999.
20. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
21. L. Naish. Automating control for logic programs. *Journal of Logic Programming (3)*, pp. 167–183, 1985.
22. S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. `http://www.haskell.org`, 1999.
23. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, Vol. 29, No. 1-3, pp. 17–64, 1996.
24. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

Encapsulating Non-Determinism in Functional Logic Computations*

Bernd Braßel Michael Hanus Frank Huch

Institute of Computer Science, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany
{bbr,mh,fhu}@informatik.uni-kiel.de

Abstract. One of the key features of the integration of functional and logic languages is the access to non-deterministic computations from the functional part of the program. In order to ensure the determinism of top-level computations in a functional logic program, which is usually a monadic sequence of I/O operations, one has to encapsulate the non-determinism (i.e., search for solutions) of logic computations. However, an appropriate approach to encapsulation can be quite subtle if subexpressions are shared, as in lazy evaluation strategies. In this paper we propose a new approach to encapsulate non-deterministic computations for the declarative multi-paradigm language Curry. It is based on providing a primitive I/O action for encapsulation from which various specialized search operators can be derived. In order to provide a formal foundation for this new approach to encapsulation, we define the operational semantics of the new primitive.

1 Why Encapsulate and How (Not) To

Functional logic languages are intended to integrate the best features provided in functional and logic languages. They are also a basis to improve the evaluation strategies of existing languages since optimal evaluation strategies exist for functional logic languages [4]. However, there is one subtle problem when combining the worlds of functional and logic programming. Usually, the top-level of a realistic functional (logic) programs is a monadic sequence of I/O operations that should be applied to the outside world (e.g., see [17]). Since the outside world cannot be copied, all non-determinism in logic computations must be encapsulated, as proposed in [10] for the declarative multi-paradigm language Curry. Modern functional logic languages are based on demand-driven evaluation strategies [4,7] which require the sharing of common subexpressions. This can cause strange behavior if some of these shared subexpressions occur within encapsulation operators. This problem will be discussed in the following.

1.1 Problems of Combining Sharing and Encapsulation

As the connection between sharing and encapsulation is central to this article, we provide a small series of examples with increasing complexity. The function "`coin`" will play the role of the archtype of all non-determinism. It is defined as

```
coin = 0
coin = 1
```

To give a first impression of the complications of non-determinism when sharing is added, regard the following two functions:

Example 1 (Different Values in the Presence of Sharing).

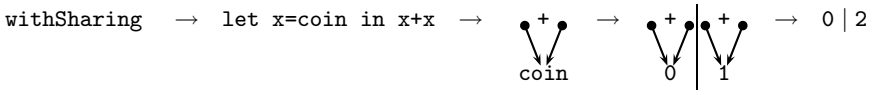
```
withoutSharing = coin + coin
withSharing    = let x = coin in x+x
```

* This work has been partially supported by the DFG under grant Ha 2457/1-2.

According to the meaning of non-deterministic functions [5] or the operational semantics of Curry [2], the two functions should show different behavior. Evaluating "`withoutSharing`" will compute all four possible combinations of `0` and `1` and add each of them, yielding `0`, `1`, `1`, or `2`. In contrast, a call to "`withSharing`" will only reduce to one of the two solutions `0` or `2`. The reason for this can be seen when looking at the reduction for both function calls, where non-deterministic choices will be denoted by putting | between them:

```
withoutSharing  →  coin + coin  →  0 + coin | 1 + coin
                →  0 + 0 | 0 + 1 | 1 + 0 | 1 + 1  →  0 | 1 | 1 | 2
```

The example without sharing can be seen as a reduction on terms. In contrast to this, a function employing sharing is a reduction on directed graphs:



One of the key properties of integrating functional and logic languages is to provide access to non-deterministic search computations from the purely functional part of the program. To do this, a particular primitive is needed, which takes an arbitrary expression and yields all possible values of this expression in a single data structure, e.g., a list. We will call such a function `getAllValues` and, corresponding to Example 1, it should show the following behavior: `getAllValues withoutSharing` should evaluate to `[0,1,1,2]` and `getAllValues withSharing` should lead to `[0,2]`.

Such a search primitive has been proposed in [10] and is contained in the definition of Curry [12]. However, the formal definition of this search primitive in [10,12] is based on a term rewriting semantics and does not cover the behavior when some subexpressions are shared. As we will see, there are different possibilities how to deal with sharing in the context of encapsulated search, and the purpose of this paper is to clarify these differences and propose a new and practically useful alternative.

We could already see one problem with a search primitive. If a function like `getAllValues` really evaluates to a *list* of possible results, the actual sequence of this list depends on the search strategy. From a declarative point of view, both sequences `[0,1]` and `[1,0]` are legitimate results of `getAllValues coin`. From this perspective it seems mandatory that `getAllValues` should return a set rather than a list of results. Later on we will argue, however, why in our approach `getAllValues` does indeed return an ordered structure. We will continue to assume a list as the result of `getAllValues` until coming back to this point later on.

Based on the previous examples, we can now consider applications of `getAllValues` for which the expected result is less clear:

Example 2 (Difference Between Strong and Weak Encapsulation).

```
coinList = getAllValues coin ++ getAllValues coin
```

Judging from the discussion up to now, it is reasonable that the result of a call to `coinList` should yield the solution `[0,1,0,1]`. However, what happens if we identify both calls to `coin` via sharing?

```
coinListWithSharing = let x=coin in getAllValues x ++ getAllValues x
```

Since the behavior w.r.t. sharing was left open in [10,12], there are at least two possibilities, which can be found in different implementations of Curry. The first possibility will be called *strong encapsulation* in the following. The strong encapsulation view is driven by the idea

that the search operator should definitely encapsulate all non-determinism. Consequently, if the expression that should be encapsulated contains a subexpression that is connected (via sharing) to some expression outside of the encapsulation, this connection is cut off. Conceptually, this means that encapsulated search creates a *a copy* of the expression before starting its evaluation. In this case, `coinListWithSharing` evaluates to `[0,1,0,1]`. One can argue that this is reasonable due to the correspondence to `coinList` and the property that sharing should have no effect on the computed results ("referential transparency"). Actually, the PAKCS implementation of Curry [9] does indeed feature this behavior.[1]

However, there is another view which we call *weak encapsulation*. This view is based on the idea that sharing should be respected even inside encapsulated search. When processing an expression shared with the outside of the encapsulation, the evaluation of this expression will *not* be encapsulated. If this expression generates non-determinism, this non-determinism also effects the computation outside which shares the expression. Thus, for `coinListWithSharing` we obtain two non-deterministic solutions `[0,0]` and `[1,1]`. Actually, the encapsulated search implemented in the Münster Curry Compiler (MCC [13]) features weak encapsulation.[2]

In order to explain the complications with the combination of sharing and encapsulated search in more detail, we consider a slight modification of Example 2:

Example 3 (Problems of Strong Encapsulation).

```
coinListWithSharing2 = let x=coin in getAllValues x ++ [x] ++ getAllValues x
```

Comparing to Example 2, one might expect that the results of a call to `coinListWithSharing2` should yield `[0,1]++[0]++[0,1]` | `[0,1]++[1]++[0,1]` for strong encapsulation. However, this is not the case, as we will see by examining the reduction on the corresponding directed graphs step by step:

`coinListWithSharing2` → `let x=coin in getAllValues x ++ [x] ++ getAllValues x`

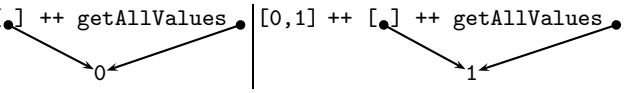
→ `getAllValues` `++ [•] ++` `getAllValues`

Since strong encapsulation duplicates subexpressions which are shared with the outside before evaluating them, the connection of `coin` inside the first `getAllValues` is cut from the other occurrences (the argument of the second `getAllValues` is still shared since its evaluation has not been initiated):

→ `getAllValues` `++ [•] ++ getAllValues` → `[0,1] ++ [•] ++ getAllValues`

Due to the standard definition of `++` (which demands the evaluation of the left argument first), the list in the middle is evaluated in the next step. As this list contains (a reference to) a call to `coin`, this step yields a non-deterministic branching:

→ `[0,1] ++ [•] ++ getAllValues` | `[0,1] ++ [•] ++ getAllValues`



[1] In PAKCS `getAllValues` can be defined as `getAllValues x = findall (=:= x)`.

[2] For reasons, which will be clarified in the following, `getAllValues` can not be defined in MCC. Instead of `getAllValues x`, we have to use the expression `findall (\y -> y =:= x)`. The analogon to `coinListWithSharing` is then:
`coinListWithSharing = let x = coin in findall (\y -> y =:= x) ++ findall (\y -> y =:= x)`
which yields the solutions `[0,0]` and `[1,1]`.

The second call to `getAllValues` is evaluated last, again cutting off the sharing connection.

```
→   [0,1] ++ [0] ++ getAllValues 0 | [0,1] ++ [1] ++ getAllValues 1
→           [0,1]++[0]++[0]       |         [0,1]++[1]++[1]
```

It is obvious that this example is problematic from a declarative point of view. Thinking in equations, both calls to `getAllValues x` in the example should evaluate to the same result. Even worse, the result of `coinListWithSharing2` would yet be different if the expression was evaluated right-to-left rather than left-to-right. All of this shows that the strong encapsulation view does not provide a declarative functional access to non-determinism.

After studying Example 3, it becomes clearer that there are good reasons to avoid the encapsulation of non-determinism when sharing is involved, like in the weak encapsulation view. Unfortunately, this approach is just as unsatisfying as strong encapsulation from a declarative point of view.

Example 4 (Problems of Weak Encapsulation).
In the weak encapsulation view, the expressions

```
findall (\y -> y =:= coin)
```
and
```
let x = coin in findall (\y -> y =:= x)
```

are not equivalent. The first yields `[0,1]`, whereas the second evaluates to `[0] | [1]`. Moreover, `findall (\y -> y =:= coin)` is different from `findall (=:= coin)`: again the first equals `[0,1]`, the second results in `[0] | [1]`.

Because of all this, one cannot define `getAllValues` for MCC at all. Any definition like

```
getAllValues x = findall (\y -> y =:= x)
```

results in the non-determinism being not encapsulated.

1.2 Problems of Encapsulation of Logical Variables

Encapsulation in the presence of logical variables features some parallels to the sharing problem. Again, we can distinguish a strong encapsulation view from a weak one.

Example 5 (Encapsulation of Logical Variables).

```
f 1 = 1                f 2 = 2               g 0 = 0
main = getAllValues (f x) ++ [g x] ++ getAllValues (f x) where x free
```

Functional logic languages allows the evaluation of function calls with logical variables as arguments. Thus, a call to `f` with a logical variable `x` results in the two non-deterministic alternatives `1` and `2` with `x` bound to `1` and `2`, respectively. For the call `g x`, `x` is bound to `0`. If we take the view of strong encapsulation, Example 5 should evaluate (with a left-to-right strategy) to `[1,2]++[0]++[]`. This is because the sharing connection to the outside of `getAllValues` is cut off and the bindings to `x` are not visible on the top level, i.e., they are encapsulated. Strong encapsulation of logical variables is performed by Prolog (see [14] for a detailed discussion of Prolog's `findall`). It is obvious that this behavior is just as problematic as the one discussed in Example 3.

Example 6 (Problems of Strongly Encapsulating Logical Variables).
The behavior of the following Prolog Program has similarities with Example 3.

```
coin(0).              coin(1).
test1(L) :- findall(X,coin(X),L),findall(X,coin(X),L).
test2(L) :- X=0,findall(X,coin(X),L),findall(X,coin(X),L).
test3(L) :- findall(X,coin(X),L),X=0,findall(X,coin(X),L).
```

The proof of the literal `test1(L)` succeeds with `L=[0,1]` and `test2(L)` succeeds with `L=[0]` whereas the proof of `test3(L)` fails because of the left-to-right semantics of Prolog.

Indeed the problems of strongly encapsulating logical variables become even worse if we think in terms of lazy evaluation, which we will do in detail in the next subsection. If `getAllValues` could be evaluated lazily, then it is possible to influence the result of a call to `getAllValues` by the following computation even if `getAllValues` was explicitly sequentialized. Such an explicit sequence can be expressed by monads which is especially important for defining sequences of I/O actions [17]. Later on (Section 1.4) we will argue why it is sensible to provide the access to encapsulated search only from the I/O monad. For now, we assume that there is a function `getAllValuesIO` of type `IO [a]` which returns a list of all values. If `getAllValuesIO` features strong encapsulation and is evaluated lazily, the following program is problematic:

Example 7 (Strong Encapsulation and Influencing the Past).

```
f 1 = 1
main = do values <- getAllValuesIO (f x)
          print values
          doSolve (x=:=1)
       where x free
```

```
f 2 = 2
main' = do values <- getAllValuesIO (f x)
           doSolve (x=:=1)
           print values
        where x free
```

If `getAllValuesIO` is strongly encapsulating, the statement `print values` in `main` generates `[1,2]` whereas the one in `main'` prints `[1]`. The reason is that `print` evaluates its argument to normal form. Thus, the execution of `print` is the time when the decision (whether `x` should be copied or not) is made. This shows that it is possible to influence the result of the encapsulated search by actions that are explicitly declared to take place later in time.

Because of these increased problems, no current implementation of Curry takes the view of strong encapsulation where logical variables are concerned, whereas the strict world of Prolog is (a bit) better suited for strong encapsulation.

What is the result of Example 5 in the view of weak encapsulation? In analogy to Section 1.1, we could state that `getAllValues (f x)` can not encapsulate the non-determinism occurring in `(f x)` but the computation splits into two non-deterministic branches. In both branches `x` is bound to a value (`1` or `2`) and, consequently, the call `(g x)` fails due to the unification of the instantiated variable `x` and `0`.

However, none of the implementations of Curry takes this view of weak encapsulation since they use a third possibility between strong and weak encapsulation for logical variables (which is also used in Oz [16]): *rigid* encapsulation.

In the view of rigid encapsulation, if a logical variable declared outside the encapsulation is processed, i.e., bound or returned as result, then the whole computation suspends. It can later be resumed if the variable gets bound by a concurrent computation. In this view, the evaluation of Example 5 would suspend. There are programs, however, for which rigid encapsulation encapsulates the non-determinism where weak encapsulation can not.

Example 8 (Rigid Encapsulation vs. Weak).

```
f 1 = 1            f 2 = 2
main | getAllValues (f x) =:= y & x=:=0 = y where x,y free
```

In rigid encapsulation, a call to `main` results in `[]` regardless of the order in which the two concurrent constraints are evaluated. In contrast, weak encapsulation produces a branching with two failing computations for left-to-right evaluation and `[]` for right-to-left.

All current implementations of Curry which feature encapsulated search take a variation of the rigid encapsulation view. It seems to be the method most conform to the "least surprise" principle when thinking in terms of declarative programming. However, rigid encapsulation has also its problems:

Example 9 (Problems of Rigid Encapsulation).

```
g 0 = 0
main1 = getAllValues (g x) ++ [g x] where x free
main2 = [g x] ++ getAllValues (g x) where x free
```

With the standard definition of `++`, which requires only the evaluation of its left argument, `main1` suspends (its result is undefined) whereas `main2` can be reduced to `[0,0]`.

When looking at current implementations featuring rigid encapsulation, there are notable differences. In PAKCS, the computation is suspended directly when applying the search operator to an expression containing a logical variable declared outside. In MCC, the computation is only suspended when such a variable is to be bound.

Example 10.

```
f x y = x
main = (findall (\y -> y =:= f 1 x)) where x free
```

The call to `main` suspends using PAKCS, whereas it reduces to `[1]` using MCC.

As a less academic example one can think of a web service. In Curry such a service communicates to the outer world via ports, i.e., synchronizing by logical variables [8]. In this case, the difference between rigid encapsulation in MCC and PAKCS is that in MCC the web service may perform all sorts of start up routines before suspending on the port, whereas in PAKCS this initialization only takes place when a message comes in.

Although MCC is preferable to PAKCS in this respect, there is another problematic difference to what we called rigid encapsulation. If the result of an encapsulated search is a logical variable declared outside the encapsulation, MCC returns this result. Unfortunately, this feature opens the door to "influencing the past" analogously to Example 7.

Example 11 (Influencing the Past in MCC).

```
f 1 x = x          f 2 2 = 2         f 3 3 = 3         g = let x free in f x
main = do values <- getAllValuesIO (g y)       main' = do values <- getAllValuesIO (g y)
          doSolve (y=:=2)                                 doSolve (y=:=3)
          print values                                    print values
       where y free                                    where y free
```

When calling `main`, `print values` generates `[2,2]` whereas for `main'` the result `[3,3]` is printed. In PAKCS, as in our operational semantics, both calls are suspended.

1.3 Lazy Evaluation and Search Strategies

One last feature of current implementations of encapsulated search should be noted: the possibility to evaluate a call to `getAllValues` lazily. The problem is well known and, thus, we give only a small example.

Example 12 (Eager vs. Lazy Encapsulated Search).

```
coin = 0
coin = coin
main = head (findall (\y -> y=:=coin))
```

Evaluating the search space eagerly, like in PAKCS, the call to `main` does not terminate. Using MCC, which features lazy search, `main` reduces to `0`.

Of course, termination is not only influenced by lazy or eager evaluation but also by the search strategy employed. Current implementations of Curry employ depth-first search. In consequence, a slight change to Example 12 leads to a different run-time behavior:

Example 13 (Depth-First Search).

```
coin = coin
coin = 0
main = head (findall (\y -> y=:=coin))
```

This program does not terminate both using PAKCS and MCC. However, the order of choices, which is relevant for depth-first search, can also be influenced by pattern matching, as the following example shows:

```
coin x = coin x
coin 0 = 0
main   = head (findall (\y -> y=:=coin 0))
```

Due to the additional pattern matching in `coin`, a call to `main` terminates using MCC.

1.4 Wish List for Future Implementations of Encapsulated Search

From the discussion above, we can now derive the desirable features of encapsulated search.

Strong Encapsulation of Sharing: One of the main reasons to provide encapsulated search is to make sure that certain parts of the program evaluation are definitely deterministic. This is especially important for I/O actions. Since weak encapsulation does not ensure encapsulation of non-determinism, cf. Examples 2 and 4, some variant of strong encapsulation has to be used to achieve deterministic I/O.

As we have seen in Example 3, strong encapsulation leads to unexpected results. In order to obtain a declarative access to search, we have to omit nested encapsulations of search.

Restriction of Encapsulated Search to Top Level: In the problematic examples for strong encapsulation, Examples 3 and also 5, `getAllValues` is called within arguments of other functions. If we restrict the use of `getAllValues` to the deterministic top level of the computation, the irritating examples are excluded. In the operational semantics, we propose in the next section, Example 3 leads to a run-time error. The intended usage within Curry is *the restriction of encapsulated search to the I/O monad.* Within the monad, different calls to a search operator like `getAllValues` have to be explicitly sequentialized. Using search as an I/O action also models its behavior more correctly, since we have seen in some examples that identical calls to `getAllValues` can have different results depending on progress and order of evaluation. In such a model, it is also less problematic to use ordered structures to represent the results of encapsulated search.

When restricting search to the I/O monad, almost all of the examples above are incorrectly stated and have to be explicitly sequentialized. Some of them will then lead to run-time error because non-determinism is not allowed in the I/O monad. Note that many

of such errors can be detected at compile time by means of a program analysis for non-determinism, like the one developed in [11].

Rigid Encapsulation for Logical Variables: As discussed in Section 1.2, rigid encapsulation is the best known choice for the treatment of logical variables. In conjunction with the top-level restriction, it might seem that strong encapsulation like in Prolog might be a good choice, too. However, it was also stated that strong encapsulation is increasingly problematic in the context of a lazy search primitive which is next on the wish list. In contrast, the only kinds of problematic examples for rigid encapsulation, cf. Example 9, are excluded by the top-level restriction discussed above.

Lazy Evaluation: For a seamless combination of functional and logic programming, a lazy search primitive is desirable. Otherwise, there is no complete correspondence between a search on the top level of the interactive environment and a search within the program. Moreover, the programmer has to think in terms of terminating reduction for the searches he wishes to encapsulate.[3]

Influencing the Search Strategy: We will show in the next section that by a lazy search primitive another desirable effect can be accomplished. The actual search strategy can be defined by manipulating the result of the search primitive. We will show how to implement depth-first and breadth-first search using this approach.

In the next section we propose a formal definition of a new approach to encapsulate search in (non-strict) functional logic languages. This approach features all the properties discussed in this wish list.

2 A New Approach to Encapsulating Search in Functional Logic Programs

In this section we first sketch our basic design of the functional access to non-deterministic search (Section 2.1). Section 2.2 contains the formal definition of this design.

2.1 A Data Structure for Representing Search

The basic idea of our approach is that a non-deterministic computation yields a data structure representing the actual search space. The definition of this representation should be independent of the search strategy employed. The basic structure of the search space can be captured by the following algebraic data type:

```
data SearchTree a = Val a | Fail | Or [SearchTree a]
```

Thus, a non-deterministic computation yields either the successful computation of a value v (constructor-rooted term or logical variable) represented by `Val` v, an unsuccessful computation (`Fail`), or a branching to several subcomputations represented by `Or` $[t_1, \ldots, t_n]$ where $t_1, \ldots, t_n$ are search trees representing the subcomputations.

Analogously to `findall` in MCC, this structure should be provided lazily, i.e., search trees are only evaluated to head normal form. By means of pattern matching on the search tree, a programmer can explore the structure and demand the evaluation of subtrees. Hence,

[3] Note that the encapsulated search primitives of Oz [16] and Prolog's `findall` [14] are related to strict languages where sharing only occurs via logical variables.

it is possible to define arbitrary search strategies on the structure of the search tree. For instance, variations of `getAllValues` for depth-first search and breadth-first search can be defined as follows:

```
getAllValuesD :: SearchTree a -> [a]                  -- depth-first search
getAllValuesD (Val v) = [v]
getAllValuesD Fail    = []
getAllValuesD (Or ts) = concatMap getAllValuesD ts

getAllValuesB :: SearchTree a -> [SearchTree a]       -- breadth-first search
getAllValuesB t = getAllValuesB' [t]

getAllValuesB' :: [SearchTree a] -> [SearchTree a]
getAllValuesB' [] = []
getAllValuesB' (t:ts) = filter isValue (t:ts)++
                        getAllValuesB' (concatMap (\(Or ts) -> ts) (filter isOr (t:ts)))
```

where `isValue` and `isOr` are test predicates for the constructors `Val` and `Or`, respectively. Evaluating the search tree lazily, these functions evaluate the list of all values in a lazy manner. For instance, a first solution can be computed using the function `head`.

2.2 An Operational Semantics for Encapsulated Search

In this section, we present an operational semantics as a proposal for a standard way to deal with the problems discussed above. This semantics is based on the operational semantics for functional logic languages presented in [2]. In the following, we explain how to extend this operational semantics in order to provide access to search data. We will explain our considerations rule by rule together with a discussion of the differences from the original rules in [2].

We define the semantics not directly for Curry. Instead, we consider a core sublanguage called Flat Curry into which Curry programs can be translated. Local function definitions are eliminated by lambda lifting. Higher-order constructs are translated to primitive functions `partcall` and `apply`. Needed narrowing and residuation are implemented as case expressions, which correspond to definitional trees [3], `fcase` for flexible functions, `case` for rigid functions, and `or` for non-deterministic branching. Furthermore, we consider normalized Flat Curry programs in which functions are only applied to variables (bound to expressions in `let`) representing references to shared expressions. For more details, see [1,2].

The basic components of the original semantics are (a) a heap which is a mapping from variables to expressions and which is necessary to model sharing, (b) a control which always holds the expression currently processed, and (c) a stack which holds two types of information: case expressions for pattern matching and variables which will be updated as soon as their corresponding expressions have been evaluated to head normal form.

For our approach we need the possibility to *encapsulate* computations. This is done by considering *a sequence of heaps* rather than a single heap and a *sequence of stacks* rather than a single one. These sequences of heaps and stacks are essentially push-down structures, the topmost is always the one currently processed. A rule which deals with the topmost heap Γ and the topmost element x of the topmost stack S has the following form:

Rulename	*Heaps*	*Control*	*Stacks*
example	$\gamma \cdot \Gamma$	e	$x \cdot S \cdot (S')$
$\rightarrowtail\!\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto e]$	e	$S \cdot (S')$

Note that, for the sake of readability, the sequence of heaps grows towards the right and the stacks towards the left. $\cdot$ denotes the concatenation on sequences. As each stack in the sequence of stacks is itself modeled by a sequence, we use brackets () to separate the different stacks, whereas no such separation is necessary for the heaps. Finally, the notation $\Gamma[x \mapsto e]$ denotes a heap in which the variable x maps to the expression e and other mappings of x in Γ are ignored (i.e., it represents a destructive heap update). Logical variables are represented by self-references ($[x \mapsto x]$).

The heap/stack sequences grow whenever a new layer of encapsulation is needed. Thus, there may be as many layers as there are calls to `getAllValues` in the program plus one. This additional layer encapsulates top-level non-determinism. This is important because the top level is usually deterministic, featuring I/O actions to present the computed values. As discussed in Section 1, this means that any non-determinism has to be encapsulated.

As in the original approach, we assume that the evaluation starts with the designated function `main`. Thus, with the additional layer, the initial state of the operational semantics is

Heaps	*Control*	*Stacks*
$[] \cdot []$	`main`	$\varepsilon \cdot (\varepsilon)$

Naturally, $\varepsilon \cdot (\varepsilon)$ will be written as (ε).

Now we are ready to discuss the different rules applicable on the operation states. The first two rules deal with retrieving information from the heaps:

Rule	*Heaps*	*Control*	*Stacks*
varcons	$\gamma \cdot \Gamma[x \mapsto c(\overline{x_n})] \cdot \gamma'$	x	S
$\rightarrowtail\!\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto c(\overline{x_n})] \cdot \gamma'$	$c(\overline{x_n})$	S
varexp	$\gamma \cdot \Gamma[x \mapsto e] \cdot \gamma'$	x	S
$\rightarrowtail\!\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto e] \cdot \gamma'$	e	$x \cdot S$

where e is not constructor-rooted, $e \neq x$, and there is no mapping from x in any of the heaps in γ'

In our notation, c denotes a constructor symbol, e represents arbitrary expressions, and overlined terms like $\overline{x_n}$ represent the sequence of terms $x_1, \ldots, x_n$. The only difference to the original rules of [2] is the condition on the heaps in γ'. As a small example, the configuration $([] \cdot [x \mapsto x] \cdot [x \mapsto 0], x, ((\varepsilon)))$ can only be succeeded by $([] \cdot [x \mapsto x] \cdot [x \mapsto 0], 0, ((\varepsilon)))$.

The next three rules, val,fun and let, are very similar to the original rules. The only point worth mentioning is that heap manipulations in these rules can only affect the topmost heap:

Rule	*Heaps*	*Control*	*Stacks*
val	$\gamma \cdot \Gamma$	v	$x \cdot S$
$\rightarrowtail\!\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto v]$	v	S
fun	γ	$f(\overline{x_n})$	S
$\rightarrowtail\!\!\!\!\rightarrow$	γ	$\rho(e)$	S
let	$\gamma \cdot \Gamma$	$let\ \{\overline{x_k = e_k}\}\ in\ e$	S
$\rightarrowtail\!\!\!\!\rightarrow$	$\gamma \cdot \Gamma[\overline{y_k \mapsto \rho(e_k)}]$	$\rho(e)$	S

where in val v is constructor-rooted or a variable with $\Gamma[v] = v$,
in fun $f(\overline{y_n}) = e$ is a program rule and $\rho = \{\overline{y_n \mapsto x_n}\}$,
and in let $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are fresh w.r.t. the heaps

The next rule, or, is more interesting as it strongly differs from the original:

Rule	*Heaps*	*Control*	*Stacks*
or	$\Gamma \cdot \gamma \cdot \Gamma' \cdot \Gamma''$	e_1 *or* e_2	$S \cdot (S')$
$\rightarrowtail\!\!\!\!\!\rightarrow$	$\Gamma[vs \mapsto [z_1, z_2],$	$Or(vs)$	S'
	$z_{1/2} \mapsto search(\Gamma'', e_{1/2}, S)] \cdot \gamma \cdot \Gamma'$		

where z, z_1 and z_2 are fresh

Here and in the following $x \mapsto [z_1, \ldots, z_n]$ is a shortcut for

$$x \mapsto l_0, \overline{l_{n-1} \mapsto z_n : l_n}, l_n \mapsto [] \text{ where } l_0, \ldots, l_n \text{ are fresh variables.}$$

In this rule, we lift the idea of lazy evaluation to the meta-level of search. This results in a lazy construction of the `SearchTree`, demanded by the top-level evaluation or a pattern matching on the `SearchTree` by means of `(f)case`, like for any other head normal form. Whenever a non-deterministic branching is executed, we know that the resulting constructor of the search tree at this point is `Or`. Thus, the current layer of encapsulation has been evaluated to head normal form. The or rule stores this result by updating the base heap (which corresponds to the top level of evaluation) and putting $Or(vs)$ on the control. As the evaluation is finished (for now), the current layer of encapsulation can be removed while its context, i.e., its topmost heap, the expression on the control, and the topmost stack, is carefully stored for future reference. We omit non-determinism at the top level of the computation, by requiring at least one heap between the top level and the current heap. Hence, non-determinism must be encapsulated in every program. Note that the other heaps in $\gamma \cdot \Gamma'$ are unaffected by the rule: the different layers of encapsulation are independent of each other.

The next rule, search, is responsible for restoring this context whenever one of the arguments of `Or` is demanded:

Rule	*Heaps*	*Control*	*Stacks*
search	γ	$search(\Gamma, e, S)$	S'
$\rightarrowtail\!\!\!\!\!\rightarrow$	$\gamma \cdot \Gamma$	e	$S \cdot (S')$

It can be easily checked that restoring the evaluation context with search is indeed dual to storing it with or.

The next two rules are concerned with the implementation of pattern matching. They are almost identical to the original ones presented in [2].

Rule	*Heaps*	*Control*	*Stacks*
case	γ	$(f)case\ e\ of\ \{\overline{p_k \rightarrow e_k}\}$	S
$\rightarrowtail\!\!\!\!\!\rightarrow$	γ	e	$(f)\{\overline{p_k \rightarrow e_k}\} \cdot S$
select	γ	$c(\overline{y_n})$	$(f)\{\overline{p_k \rightarrow e_k}\} \cdot S$
$\rightarrowtail\!\!\!\!\!\rightarrow$	γ	$\rho(e_i)$	S

where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$

The two variations `case` and `fcase` (short for "flexible case") correspond to the evaluation modes "rigid" and "flexible" as described in [12]. The difference will become important when discussing rule guess below.

Before discussing the remaining rules, we give an example of a simple program applying only those rules discussed so far.

Example 14 (Operational Semantics in Action).
Consider the following simple program:

```
main = let c = coin in                          coin = 0 or 1
         case c of { 0 -> c;
                     1 -> c}
```

Using our operational semantics, we obtain:

$$
\begin{array}{lllr}
 & [\,]\cdot[\,] & \texttt{main} & (\varepsilon) \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{fun}} & [\,]\cdot[\,] & \texttt{let c=coin in case \{0->c;1->c\}} & (\varepsilon) \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{let}} & [\,]\cdot[\texttt{c}\mapsto\texttt{coin}] & \texttt{case c of \{0->c;1->c\}} & (\varepsilon) \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{case}} & [\,]\cdot[\texttt{c}\mapsto\texttt{coin}] & \texttt{c} & \texttt{\{0->c;1->c\}}(\varepsilon) \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{varexp}} & [\,]\cdot[\texttt{c}\mapsto\texttt{coin}] & \texttt{coin} & \texttt{c\{0->c;1->c\}}(\varepsilon) \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{fun}} & [\,]\cdot[\texttt{c}\mapsto\texttt{coin}] & \texttt{0 or 1} & \texttt{c\{0->c;1->c\}}(\varepsilon) \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{or}} & \Gamma_1 & \texttt{Or vs} & \varepsilon
\end{array}
$$

where $\Gamma_1 = [\texttt{vs}\mapsto\texttt{z1:vs1}, \texttt{vs1}\mapsto\texttt{z2:vs2}, \texttt{vs2}\mapsto\texttt{[]},$
$\texttt{z1}\mapsto\texttt{search([c}\mapsto\texttt{coin],0,c\{0->c;1->c\})},$
$\texttt{z2}\mapsto\texttt{search([c}\mapsto\texttt{coin],1,c\{0->c;1->c\})}]$

If we later resume the evaluation of the second argument of the search tree, we get:

$$
\begin{array}{lllr}
 & \Gamma_1 & \texttt{z2} & S \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{varexp}} & \Gamma_1 & \texttt{search([c}\mapsto\texttt{coin],1,c\{0->c;1->c\})} & \texttt{z2}\,S \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{search}} & \Gamma_1\cdot[\texttt{c}\mapsto\texttt{coin}] & \texttt{1} & \texttt{c\{0->c;1->c\}(z2)}S \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{val}} & \Gamma_1\cdot[\texttt{c}\mapsto\texttt{1}] & \texttt{1} & \texttt{\{0->c;1->c\}(z2)}S \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{select}} & \Gamma_1\cdot[\texttt{c}\mapsto\texttt{1}] & \texttt{c} & \texttt{(z2)}S \\
\rightarrowtail\!\!\!\!\!\rightarrow_{\mathsf{varcons}} & \Gamma_1\cdot[\texttt{c}\mapsto\texttt{1}] & \texttt{1} & \texttt{(z2)}S
\end{array}
$$

The next rule, guess, is responsible for starting those non-deterministic searches which are induced by guessing bindings for logical variables (narrowing). Consider a flexible function f. When f is called with a logical variable x as an argument, pattern matching on this variable results in a non-deterministic branching. In each branch, x is bound to a different constructor corresponding to the patterns of the `fcase`. This behavior is modeled by the rule guess which combines the rules select and or discussed above.

Rule	*Heaps*	*Control*	*Stacks*
guess	$\Gamma\cdot\gamma\cdot\Gamma'\cdot\Gamma''[x\mapsto x]$	x	$f\left\{\overline{c_k(\overline{x_{n_k}})\rightarrow e_k}\right\}\cdot S\cdot(S')$
$\rightarrowtail\!\!\!\!\!\rightarrow$	$\Gamma[vs\mapsto[z_1,\ldots,z_k],\overline{z_k\mapsto s_k}]\cdot\gamma\cdot\Gamma'$	$Or(vs)$	S'

where $k>1$, and for all $i\in\{1,\ldots,k\}$ $\rho_i=\{\overline{x_{n_i}\mapsto y_{n_i}}\}$, $\overline{y_{n_i}}$ are fresh
and $s_i = search(\Gamma''[x\mapsto c_i(\overline{y_{n_i}}),\overline{y_{n_i}\mapsto y_{n_i}}],\rho_i(e_i),S)$

Since this rule induces non-determinism, we again restrict its application to encapsulated search computations (recognizable by the existence of at least three heaps) to omit non-deterministic top-level computations.

Note that, in order to apply rule guess, the logical variable x must be declared within the topmost heap. No guessing of bindings will take place when x is declared outside the encapsulation. This corresponds to the view of *rigid encapsulation* discussed in Section 1.2. The computation will suspend, i.e., no rule is applicable until the variable is bound by another (concurrent) constraint. However, treating concurrency is beyond the scope of this paper but discussed in [1].

Furthermore, in order to apply rule guess, there has to be more than one pattern in the case expression. If there is only a single pattern, there is no need to perform a non-deterministic branching. Therefore, the corresponding rule guess-1 is quite simple and more similar to select than to guess:

Rule	*Heaps*	*Control*	*Stacks*
guess-1	$\gamma \cdot \Gamma[x \mapsto x]$	x	$f\{c(\overline{x_n}) \to e\} \cdot S$
$\rightarrowtail\!\!\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto c(\overline{y_n}), \overline{y_n \mapsto y_n}]$	$\rho(e)$	S

where $\overline{y_n}$ are fresh and $\rho = \{\overline{x_n \mapsto y_n}\}$

The next three rules deal with computations that have been finished, both at top level or for an encapsulated search. If the computation was successful (i.e. the remaining stack is empty), the result has to be wrapped with the corresponding constructor *Val*. Analogously to or, the context of the computation has to be stored for future reference.

Rule	*Heaps*	*Control*	*Stacks*
rescons	$\Gamma \cdot \gamma \cdot \Gamma'$	$c(\overline{x_n})$	(S)
$\rightarrowtail\!\!\!\!\!\rightarrow$	$\Gamma\left[z \mapsto c(\overline{z_n}), \overline{z_n \mapsto \mathit{search}(\Gamma', x_n, \varepsilon)}\right] \cdot \gamma$	$\mathit{Val}(z)$	S
resvar	$\Gamma \cdot \gamma \cdot \Gamma'[x \mapsto x]$	x	(S)
$\rightarrowtail\!\!\!\!\!\rightarrow$	$\Gamma[z \mapsto z] \cdot \gamma$	$\mathit{Val}(z)$	S

where $z, \overline{z_n}$ are fresh

Failing computations are encoded by the constructor *Fail*. Obviously, the evaluation context of such a computation can be thrown away.

Rule	*Heaps*	*Control*	*Stacks*
fail	$\gamma \cdot \Gamma$	$c(\overline{x_n})$	$(f)\{\overline{p_k \to e_k}\} \cdot S \cdot (S')$
$\rightarrowtail\!\!\!\!\!\rightarrow$	γ	*Fail*	S'

where $p_i \neq c(\cdots)$ for $i = 1, \ldots, k$

The purpose of the last rule of the operational semantics is simply to ensure that an encapsulated search can only be started at top level (a configuration with exactly two heaps).

Rule	*Heaps*	*Control*	*Stacks*
getsearchtree	$\Gamma \cdot \Gamma'$	$\mathit{getSearchTree}(e)$	S
$\rightarrowtail\!\!\!\!\!\rightarrow$	$\Gamma \cdot \Gamma'$	$\mathit{search}([\,], e, \varepsilon)$	S

3 A Complex Example

The following program demonstrates the effect of using the function *getSearchTree* twice which is in some sense nested because of lazy evaluation. First, the program computes the search tree resulting from `coin`. This tree is then used in the function `getAny` to select all values in this tree non-deterministically. This non-deterministic computation is again encapsulated by *getSearchTree* and both encapsulated results are compared.

```
main = let c = coin,                        coin = 0 or 1
           cST = getSearchTree c,
           any = getAny cST,                x == y = let a = prim_Eq(x,y),
```

```
          cST' = getSearchTree any
        in cST == cST'

getAny t = case t of {
             Or ts -> getAnyL ts;
             Val v -> v}
```

```
          b = hnf(y,a)
        in hnf(x,b)

getAnyL ts = case ts of {
               (x:xs) ->
                 (getAny x) or (getAnyL xs)}
```

To compare the two results, we use the (predefined) function "==". It evaluates its arguments to head normal form (`hnf`) and compares them by `prim_Eq` afterwards. The function `prim_Eq` is a primitive function which, for identical top-level constructors, compares their arguments recursively by means of `(==)` and sequential conjunction (`&&`). See [1] for a more detailed description of the integration of primitive functions into the semantics. Appendix A presents the semantics for this example.

The computation yields the value `Val False`. This (perhaps) surprising result is produced due to different searches performed for the computations of `cST` and `cST'`. A full evaluation of the search tree would yield `cST = Or [Val 0, Val 1]`. Applying the function `any` to this value and encapsulating it again by `getSearchTree`, we obtain `cST' = Or [Val 0,Or [Val 1,Fail]]`. The presented program compares the structure of the search tree exactly. However, both compute the same values and a comparison of `getAllValuesD` or `getAllValuesB` applied to `cST` and `cST'` would succeed.

4 Conclusion

In this paper we presented a new approach to the encapsulation of non-determinism in lazy functional logic languages. The initial discussion showed that existing approaches are inadequate. A good approach should strongly encapsulate sharing, omit nested usage of encapsulation, rigidly encapsulate logical variables and generate the encapsulation of the search lazily. Our solution is the lazy creation of a search tree, representing the search space of the computation. By means of this lazy data structure, it is possible to define different search strategies on the level of the functional logic language. The basic idea in the construction of the search tree is adapted from laziness in a standard operational semantics for functional logic languages with sharing and attracts to be a consequent extension of this semantics to the level of meta-programming.

The search tree presented in this paper covers only the kind of fairness one can accomplish by the operator `try` [10]. However, this is not a principal limitation of the approach. Extending search trees by

```
data SearchTree a = Val a | Fail | Or [SearchTree a] | Eval (SearchTree a)
```

and yielding a value of type `Eval` in every `fun` rule, we would cover fair search in the stronger sense. However, the practicability of this extension can only be estimated by an implementation of the approach.

For future work, we want to prove that the presented semantics and the semantics without encapsulation [2] compute comparable results for non-deterministic computations. Furthermore, we have to show that our approach implements strong and rigid encapsulation for shared expressions and logical variables, respectively, and that the proposed properties of these encapsulation strategies hold. To estimate the practical usability of our approach, we plan to integrate this semantics into our Flat Curry interpreter based on [1,2]. In a second step, we want to use the presented ideas for a new implementation of Curry. In contrast to PAKCS, which translates to Prolog, we want to use Haskell [15] as target language. This implementation should then cover all discussed aspects.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 70–6. Elsevier Science Publishers, 2002.
2. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for functional logic languages. *Electronic Notes in Theoretical Computer Science*, 76, 2002.
3. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
6. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
7. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
8. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.
9. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Pakcs: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2003.
10. M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.
11. M. Hanus and F. Steiner. Type-based nondeterminism checking in functional logic programs. In *Proc. of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 202–213. ACM Press, 2000.
12. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
13. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
14. L. Naish. All solutions predicates in Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 73–77, Boston, 1985.
15. S.L. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, 1999.
16. C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.
17. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

A A Complex Example

The example program of Section 3 has the following operational semantics:

$$
\begin{array}{lll}
[\,]\cdot[\,] & \texttt{main} & (\varepsilon)\\
\rightarrowtail\!\!\!\!\rightarrow [\,]\cdot[\,] & \texttt{let c=coin,...in cST==cST'} & (\varepsilon)\\
\rightarrowtail\!\!\!\!\rightarrow [\,]\cdot \Gamma_1 = [\texttt{c} \mapsto \texttt{coin}, & \texttt{cST == cST'} & (\varepsilon)\\
\quad \texttt{cST} \mapsto \texttt{getSearchTree c}, & & \\
\quad \texttt{any} \mapsto \texttt{getAny cST}, & & \\
\quad \texttt{cST'} \mapsto \texttt{getSearchTree any}] & & \\
\rightarrowtail\!\!\!\!\rightarrow [\,]\cdot \Gamma_1 & \texttt{let a=... in hnf(cST,b)} & (\varepsilon)\\
\rightarrowtail\!\!\!\!\rightarrow [\,]\cdot \Gamma_2 = \Gamma_1[\texttt{a} \mapsto \texttt{prim\_Eq(cST,cST')}, & \texttt{hnf(cST,b)} & (\varepsilon)\\
\quad \texttt{b} \mapsto \texttt{hnf(cST',a)}] & &
\end{array}
$$

$\rightarrowtail\!\!\!\!\rightarrow [\,] \cdot \Gamma_2$	`cST`	`hnf(b)`(ε)
$\rightarrowtail\!\!\!\!\rightarrow [\,] \cdot \Gamma_2$	`getSearchTree(c)`	`cST hnf(b)`(ε)
$\rightarrowtail\!\!\!\!\rightarrow [\,] \cdot \Gamma_2$	`search([],c,`ε`)`	`cST hnf(b)`(ε)
$\rightarrowtail\!\!\!\!\rightarrow [\,] \cdot \Gamma_2 \cdot [\,]$	`c`	$($`cST hnf(b)`$(\varepsilon))$
$\rightarrowtail\!\!\!\!\rightarrow [\,] \cdot \Gamma_2 \cdot [\,]$	`coin`	`c`$($`cST hnf(b)`$(\varepsilon))$
$\rightarrowtail\!\!\!\!\rightarrow [\,] \cdot \Gamma_2 \cdot [\,]$	`0 or 1`	`c`$($`cST hnf(b)`$(\varepsilon))$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 = [$`vs` $\mapsto$ `z1:vs'`, `vs'` $\mapsto$ `z2:vs''`, `vs''` $\mapsto$ `[]`, `z1` $\mapsto$ `search([],0,c)`, `z2` $\mapsto$ `search([],1,c)`$] \cdot \Gamma_2$	`Or vs`	`cST hnf(b)`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 = \Gamma_2[$`cST` $\mapsto$ `Or vs`$]$	`Or vs`	`hnf(b)`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4$	`b`	(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4$	`hnf(cST',a)`	(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4$	`cST'`	`hnf(a)`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4$	`getSearchTree any`	$S_1 =$ `cST' hnf(a)`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4$	`search([],any,`ε`)`	S_1
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`any`	(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`getAny cST`	`any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`case cST of...`	`any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`cST`	`{Or ts->...}any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`Or vs`	`{Or ts->...}any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`getAnyL vs`	`any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`case vs of ...`	`any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`vs`	`{(x:xs)->...}any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`z1:vs'`	`{(x:xs)->...}any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_3 \cdot \Gamma_4 \cdot [\,]$	`(getAny z1) or (getAnyL vs')`	`any`(S_1)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 = \Gamma_3[$`vs1` $\mapsto$ `[z3,z4]` `z3` $\mapsto$ `search([],getAny z1,any)` `z4` $\mapsto$ `search([],getAnyL vs',any)`$] \cdot \Gamma_4$	`Or vs1`	S_1
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_6 = \Gamma_4[$`cST'` $\mapsto$ `Or vs1`$]$	`Or vs1`	`hnf(a)`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_6$	`a`	(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_6$	`prim_Eq(cST,cST')`	`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_6$	`vs == vs1`	`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_7 = \Gamma_6[$`b1` $\mapsto$ `z1==z3`, `b2` $\mapsto$ `z2==z4`$]$	`b1 && b2`	`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_7$	`case b1 of...`	`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_7$	`b1 {True -> b2; False -> Fale}`	`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_7$	`z1==z3`	$\{\ldots\}$`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_7$	`let e1=prim_Eq(z1,z3),e2=hnf(z3,e1) in hnf(z1,e2)`	$\{\ldots\}$`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_8 = \Gamma_7[$`e1` $\mapsto$ `prim_Eq(z1,z3)`, `e2` $\mapsto$ `hnf(z3,e1)`$]$	`hnf(z1,e2)`	$\{\ldots\}$`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_8$	`z1`	`hnf(e2)`$\{\ldots\}$`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_8$	`search([],0,c)`	`z1 hnf(e2)`$\{\ldots\}$`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_8 \cdot [\,]$	`0`	`c`$($`z1 hnf(e2)`$\{\ldots\}$`a`$(\varepsilon))$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_5 \cdot \Gamma_8 \cdot [$`c` $\mapsto$ `0`$]$	`0`	$($`z1 hnf(e2)`$\{\ldots\}$`a`$(\varepsilon))$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 = \Gamma_5[$`v1` $\mapsto$ `0`$] \cdot \Gamma_8$	`Val v1`	`z1 hnf(e2)`$\{\ldots\}$`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} = \Gamma_8[$`z1` $\mapsto$ `Val v1`$]$	`Val v1`	`hnf(e2)`$\{\ldots\}$`a`(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10}$	`e2`	$\{\ldots\}$`a`(ε)

$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10}$	`hnf(z3,e1)`	$\{\ldots\}\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10}$	`z3`	$\mathtt{hnf(e1)}\{\ldots\}\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10}$	`search([],getAny z1,any)`	$S_2 = \mathtt{z3}\,\mathtt{hnf(e1)}\{\ldots\}\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} \cdot [\,]$	`getAny z1`	$\mathtt{any}(S_2)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} \cdot [\,]$	`case z1 of...`	$\mathtt{any}(S_2)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} \cdot [\,]$	`z1`	$\{\mathtt{Or\ ts\ ->\ ...}\}\mathtt{any}(S_2)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} \cdot [\,]$	`Val v1`	$\{\mathtt{Or\ ts\ ->\ ...}\}\mathtt{any}(S_2)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} \cdot [\,]$	`v1`	$\mathtt{any}(S_2)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} \cdot [\,]$	`0`	$\mathtt{any}(S_2)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} \cdot [\mathtt{any} \mapsto \mathtt{0}]$	`0`	$\mathtt{z1}(S_2)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_9 \cdot \Gamma_{10} \cdot [\mathtt{any} \mapsto \mathtt{0}, \mathtt{z1} \mapsto \mathtt{0}]$	`0`	(S_2)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} = \Gamma_9[\mathtt{v2} \mapsto \mathtt{0}] \cdot \Gamma_{10}$	`Val v2`	S_2
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{12} = \Gamma_{10}[\mathtt{z3} \mapsto \mathtt{Val\ v2}]$	`Val v2`	$\mathtt{hnf(e1)}\{\ldots\}\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{12}$	`e1`	$\mathtt{hnf(e1)}\{\ldots\}\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{12}$	`prim_Eq(z1,z3)`	$\{\ldots\}\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{12}$	`prim_Eq(v1,v2)`	$\{\ldots\}\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{12}$	`True`	$\{\ldots\}\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{12}$	`b2`	$\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{12}$	`z2==z4`	$\mathtt{b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{12}$	`let e3=prim_Eq(z2,z4),e4=hnf(z4,e3) in hnf(z2,e4)`	$\mathtt{b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{13} = \Gamma_{12}[\mathtt{e3} \mapsto \mathtt{prim\_Eq(z2,z4)}, \mathtt{e4} \mapsto \mathtt{hnf(z4,e3)}]$	`hnf(z2,e4)`	$\mathtt{b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{13}$	`z2`	$\mathtt{hnf(e4)b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{13}$	`search([],1,c)`	$\mathtt{z2\,hnf(e4)b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{11} \cdot \Gamma_{13} \cdot [\,]$	`1`	$\mathtt{c}(\mathtt{z2\,hnf(e4)b2}\,\mathtt{a}(\varepsilon))$
$\rightarrowtail\!\!\!\!\rightarrow^* \Gamma_{11} \cdot \Gamma_{13} \cdot [\mathtt{c} \mapsto \mathtt{1}]$	`1`	$(\mathtt{z2\,hnf(e4)b2}\,\mathtt{a}(\varepsilon))$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} = \Gamma_{11}[\mathtt{v3} \mapsto \mathtt{1}] \cdot \Gamma_{13}$	`Val v3`	$\mathtt{z2\,hnf(e4)b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15} = \Gamma_{13}[\mathtt{z2} \mapsto \mathtt{Val\ v3}]$	`Val v3`	$\mathtt{hnf(e4)b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15}$	`e4`	$\mathtt{b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15}$	`hnf(z4,e3)`	$\mathtt{e4\,b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15}$	`z4`	$\mathtt{hnf(e3)e4\,b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15}$	`search([],getAnyL vs',any)`	$S_3 = \mathtt{z4\,hnf(e3)e4\,b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15} \cdot [\,]$	`getAnyL vs'`	$\mathtt{any}(S_3)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15} \cdot [\,]$	`case vs' of...`	$\mathtt{any}(S_3)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15} \cdot [\,]$	`vs'`	$\{\mathtt{(x:xs)->...}\}\mathtt{any}(S_3)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15} \cdot [\,]$	`z2:vs''`	$\{\mathtt{(x:xs)->...}\}\mathtt{any}(S_3)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{14} \cdot \Gamma_{15} \cdot [\,]$	`(getAny z2) or (getAnyL vs'')`	$\mathtt{any}(S_3)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{16} \cdot \Gamma_{15}$	`Or vs2`	S_3
where $\Gamma_{16} = \Gamma_{14}[\mathtt{vs2} \mapsto \mathtt{[z5,z6]}, \mathtt{z5} \mapsto \mathtt{search([],getAny\ z2,any)}$ $\mathtt{z5} \mapsto \mathtt{search([],getAnyL\ vs'',any)}]$		
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{16} \cdot \Gamma_{17} = \Gamma_{15}[\mathtt{z4} \mapsto \mathtt{Or\ vs2}]$	`Or vs2`	$\mathtt{hnf(e3)e4\,b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{16} \cdot \Gamma_{17}$	`e3`	$\mathtt{e4\,b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{16} \cdot \Gamma_{17}$	`prim_Eq(z2,z4)`	$\mathtt{e4\,b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{16} \cdot \Gamma_{17}$	`False`	$\mathtt{e4\,b2}\,\mathtt{a}(\varepsilon)$
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{16} \cdot \Gamma_{17}[\mathtt{e4,b2,a} \mapsto \mathtt{False}]$	`False`	(ε)
$\rightarrowtail\!\!\!\!\rightarrow \Gamma_{16}[\mathtt{v4} \mapsto \mathtt{False}]$	`Val v4`	ε

Comparing Copying and Trailing Implementations for Encapsulated Search

Wolfgang Lux

Universität Münster
wlux@uni-muenster.de

Abstract The multi-paradigm language Curry seamlessly integrates concepts from functional programming, logic programming, and concurrent constraint solving. One of its distinctive features is the encapsulated search that allows the programmer to control non-deterministic computation steps in a program. This makes it possible to use other strategies for computing the solutions of a goal than the global backtracking search which is usually employed by implementations of logic programming languages.

The encapsulated search requires an implementation to maintain multiple bindings for a single variable at the same time. Different solutions to this problem have been proposed in the context of parallel logic programming, including copying, binding arrays, and extended trailing schemes. In this paper we compare two implementations of the Münster Curry compiler using copying and trailing, respectively. Preliminary experimental data for both implementations suggest that trailing is preferable to copying.

1 Introduction

The multi-paradigm language Curry [Han03] integrates features from functional languages, logic languages, and concurrent constraint programming. Curry supports the two most important operational principles developed for the evaluation of functional logic languages, namely narrowing and residuation. A distinctive feature of Curry is the encapsulated search [HS98] that serves two different purposes. First, it allows the programmer to constrain non-deterministic search in programs that interact with the external world without breaking Curry's declarative approach to I/O, and, second, it makes it possible to implement and use search strategies other than the usual depth-first search for computing the solutions of a goal without having to change the goal itself.

With search strategies implemented using the encapsulated search, it becomes possible to explore different parts of a goal's search tree concurrently. This is useful, for instance, in order to avoid non-termination if the search tree contains infinite paths. In addition, search strategies can be used for pruning the search tree, e.g., with previously computed solutions. For an implementation this means that different instantiations of the variables in the goal can exist at the same time. This problem has been studied, among others, in the context of or-parallel implementations of logic languages. Such implementations explore different parts of the search tree concurrently in order to speed up the search for solutions when don't know non-determinism is involved.

Essentially, three different solutions have been developed for maintaining multiple bindings: Extended trailing schemes, binding arrays, and copying. The encapsulated search in Curry is more general than an or-parallel evaluation. Nevertheless, these techniques are applicable to it as well.

In this paper we compare copying and trailing implementations of the encapsulated search. The novel aspect of our work is that the comparison is based on two implementations of the same abstract machine using the same compiler front-end. In addition, most other comparisons are based only on an or-parallel evaluation, with the exception of [Sch99]. However, that paper compares different implementations and tries only to show the general feasibility of a copying approach. Thus, we think to be the first to perform a fair comparison between both approaches. We will also consider implementation details, in particular related to memory management, that are entailed by the choice of the algorithm. Measurements comparing both implementations suggest that the trailing

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
last xs | append ys [y] =:= xs = y where y,ys free
```

Figure 1. A sample Curry program

approach is in general the better choice for an implementation, even though copying leads to a more streamlined implementation and runs faster on purely functional code.

The rest of the paper is organized as follows. In the next section we will briefly introduce the language Curry and the encapsulated search. In section 3, we will discuss the different algorithms for managing multiple bindings in general. The next sections introduce our abstract machine and outline the differences between the trailing and copying implementations of the machine. Sections 6 and 7 present experimental data for both implementations and related work, and the final section concludes.

2 The Encapsulated Search in Curry

Curry combines lazy functional programming with the capabilities of logic programming languages and concurrent constraint solving. Curry's operational model is based on an optimal reduction strategy [Ant97] that integrates narrowing and residuation, the two most important operational principles developed for the implementation of functional logic languages. Narrowing [Red85] combines unification and reduction, allowing the non-deterministic instantiation of logic variables in expressions, whereas the residuation strategy [ALN87] delays the evaluation of functions until their arguments have been sufficiently instantiated. We will not consider concurrent evaluation and residuation in the rest of this paper.

Curry uses the same syntax and supports most features of Haskell [Pey03]. In addition, programs in Curry can use logic variables, which allow representing data with partial information and implicit search for solutions. The kernel of Curry includes only equality constraints, but implementations can provide constraint solving capabilities for other domains as well. Constraint expressions in Curry have type `Success` and can be used in guards of function definitions where they are checked for satisfiability. The primitive constraint functions are the trivially satisfied constraint `success`, the (concurrent) conjunction of two constraints `(&)`, and the equality constraint `(=:=)`. An equality constraint e_1 `=:=` e_2 between two expressions e_1 and e_2 is satisfied if both expressions can be reduced to the same finite data term. During the evaluation of both arguments, unbound variables are instantiated as necessary.

Predicates in the sense of logic programming are functions with result type `Success`. As there is no syntactical distinction between predicates and other functions, higher order functions like `map` can be applied to them as well.

Fig. 1 shows a simple Curry program defining two functions `append` and `last` that implement list concatenation and return the last element of a list, respectively. The definition of `last` uses an implicit search in its guard expression `append ys [y] =:= xs`. Logic variables have to be introduced explicitly with a declaration of the form $x_1, \ldots, x_n$ `free`. This is necessary in order to control the scope of variables in the context of an encapsulated search, but also helps to avoid some trivial programming errors.

A distinctive feature of Curry is the encapsulated search [HS98] that allows the programmer to provide optimal search strategies for the goals in the program without having to change the goals themselves. In addition, it makes it possible to use non-deterministic search in the context of Curry's

monadic I/O system, which enforces a single threaded interaction with the external world and thus provides a declarative approach to input and output.

The basic primitive of the encapsulated search is the function `try :: (a → Success) → [a → Success]`. The argument of `try` is the predicate whose solution is searched. For clarity, we will call this function a *search goal* henceforth. When an application `try` g is evaluated, the search goal g is applied to a fresh variable x and this application is reduced until one of the following three conditions is met: The reduction of the expression $g\ x$ fails, the reduction succeeds, or the reduction can be continued only by performing a non-deterministic computation step. In the first case `try` returns an empty list, and in the second case it returns a list `[`g'`]` where g' is the solved form of the search goal g. The solved form of g is a function that is equivalent to the λ-abstraction λ`x` $\rightarrow$ `x=:=`t where t is the solution of g. If a non-deterministic computation step must be performed, the reduction of the search goal is suspended and `try` returns a list with one function for each possible continuation of the computation after the non-deterministic step. For instance, the evaluation of `try (`λ`y → let ys free in append ys [y] =:= [1,2])` yields a list equivalent to

```
[(λy → let ys free in
       ys =:= [] & append [] [y] =:= [1,2]),
 (λy → let ys,y',ys' free in
       ys =:= (y':ys') & append (y':ys') [y] =:= [1,2])]
```

On top of the primitive search operator, search strategies for a goal can be implemented. For instance, a breadth-first search strategy can be implemented as follows.

```
bfs g = search [g]
  where search []     = []
        search (g:gs) = collect (try g) gs
        collect []         gs' = search gs'
        collect [g]        gs' = g : search gs'
        collect (g1:g2:gs) gs' = search (gs'++g1:g2:gs)
```

The local function `search` processes a list of goals that are not yet solved. As long as this list is not empty, `search` takes the first goal from the list and applies `try` to it. The result returned by `try` is then used by the `collect` function in order to decide how the search is continued. Each of the three equations defining `collect` corresponds to one of the conditions when the encapsulated search returns control to the caller. If the goal fails (first equation), `search` continues with the next goal. If the goal succeeds (second equation), its solved form is returned and `search` is applied to the remaining goals. If a non-deterministic computation step must be performed (third equation), all possible continuations are appended to the list of unsolved goals, and `search` continues with the first goal from the combined list. If the continuations were added to the front of the list of unsolved goals, one would obtain an implementation of depth-first search. Conforming to the overall lazy evaluation mechanism employed by Curry, search goals are evaluated only as far as needed by the program. Thus, by composing `bfs` with the function `head` that returns the head of a list one can define a search strategy that computes only one solution with a breadth-first search:

```
one g = head (bfs g)
```

Due to lazy evaluation, `one` can be applied to search goals with an infinite search tree. For instance, it could be used for finding a path between two nodes in a cyclic graph without having to maintain a set of nodes that have been traversed already. In addition, using `one` for this kind of problem has the appealing property that it returns the shortest path between the two nodes.

Search strategies are not limited to simple traversals of the search tree. The functions returned from the encapsulated search can be used like any other function defined in the program. For example it is possible to set up additional constraints that cut off parts of the search tree using previously computed solutions. It is even possible to continue the evaluation of the goals returned by `try` on the top-level.

3 Implementation Techniques for Multiple Bindings

Fig. 2 displays the search tree of the search goal λ`y` $\rightarrow$ `let ys free in append ys [y] =:= [1,2]`. The marked nodes correspond to the computation steps where the encapsulated search returns. The edges are labeled with the variable instantiations performed by the computation. In an implementation that traverses this search tree sequentially in depth-first order, only one

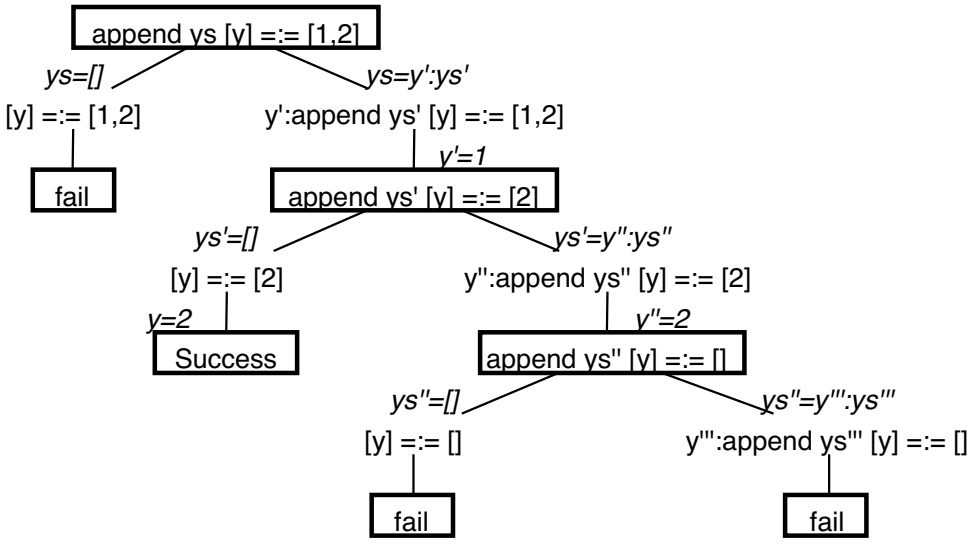


Figure 2. Search tree for λ`y` $\rightarrow$ `let ys free in append ys [y] =:= [1,2]`

set of variable bindings is active at each time. This is no longer the case for other strategies that can be implemented with the encapsulated search. In this case, search goals corresponding to different nodes in the search tree can exist at the same time in the program and therefore different sets of bindings for the search goal may exist simultaneously. In order to distinguish the different sets, we assign a unique identifier to each of them, which we will call a *search space identifier* in the following. The identified set of bindings will be called a *search space*.

The problem of maintaining multiple bindings for a variable simultaneously has been studied in the context of (or-)parallel implementations of logic languages, but also for concurrent constraint solving and for implementing first-class stores [JD98]. Implementation techniques that have been developed include copying [GHPS94], extended trailing schemes[MSS95], binding arrays [GSP95], and recomputation. All of these techniques can be used for the implementation of the encapsulated search as well.

3.1 Copying

In an implementation that uses copying, the representation of the search goal in memory is duplicated at a non-deterministic computation step. Thus, independent copies of the search goal exist for each search space that is explored by the program. Therefore, bindings performed in one part of the search tree do not interfere with those performed by other solutions. In the context of Curry this means that each of the search goals that are returned by `try` can be executed as in a purely functional program. In particular, it is trivial to switch between different nodes in the search tree. On the other hand, the obvious disadvantage of copying is that it has to create a new copy of the whole search goal when a non-deterministic computation step is performed. This not only does increase the memory usage of the program, but also requires computation time which is proportional to the size of the search goal.

The amount of copying can be reduced by various techniques. First of all, the Münster Curry compiler creates copies of a search goal lazily, i.e., only when one of the search continuations is actually evaluated. Furthermore, constant data can be shared among the different solutions. For instance, in the sample search goal from Fig. 2, the list `[1,2]` can be shared between all solutions and need not be copied. While this can be implemented trivially for constants which are manifest in the source code, it is more difficult to share ground data terms which are results of lazy applications.

3.2 Trailing

The trail stack used by backtracking implementations of logic and functional logic languages can be regarded as an efficient means to store only the difference between two neighboring nodes in the search tree. Obviously, for a depth-first traversal of the search tree it is sufficient to save only the information that is necessary in order to recover the bindings that were in effect at a particular node in the search tree after its subtree has been traversed. In addition, this information can be discarded immediately after restoring the bindings. For other search strategies this information can no longer be discarded and therefore a simple stack is no longer sufficient. Furthermore, the trail has to be complemented by an inverse trail[1] holding the bindings that must be restored when the node is visited again from its parent. For instance, at the node `append ys' [y] =:= [2]` in the center of Fig. 2, the trail contains the information that the variables `y'` and `ys` have to be unbound when returning to the root of the tree, whereas the inverse trail contains the information that `ys` has to be bound to the term `y':ys'`, and `y'` to the number `1` when entering the node again.

When switching between two nodes in the search tree, the program has to undo the bindings performed on the path from the active node in the tree up to the closest common ancestor with the target node, and then restore the bindings recorded on the inverse trails on the path from the ancestor to the target node. Obviously, the amount of work to be performed for this switch is proportional to the distance of the nodes in the search tree and thus limited only by the tree's size. In addition, the trailing implementation adds a small but constant overhead to the binding of a variable for recording the update on the trail. It is possible to avoid trailing of updates when the old state of the computation cannot be observed by the program. For instance, the unbound state of the variable `y'` in the example cannot be observed in any of the search goals that are returned from a traversal of the search tree with the encapsulated search and therefore its binding need not be saved. If memory is allocated in ascending order, it is easy to check whether trailing is necessary with at most two address comparisons.

[1] The inverse trail is called a *script* in AMOZ[MSS95], the abstract machine for Oz.

Another problem of trailing is that the trails accumulate bindings for nodes which are no longer actively used by the computation. For instance, for the goal from Fig.2, the trails will contain bindings for the auxiliary variables `ys` and `ys'`. Thus, their bindings are changed even when entering or leaving search spaces in which these variables are not used, e.g., the search space returned for the node `append ys'' [y] =:= []`. Techniques for removing such redundant trail entries are known for a depth-first traversal using a single trail stack, but these are not directly applicable to the distributed trail segments used by an implementation of the encapsulated search.

3.3 Binding Arrays

Binding arrays were introduced for the or-parallel execution of logic programs on shared memory multi-processor architectures. Instead of saving the binding of a variable directly in the corresponding node, a variable is now associated with a unique key. Each processor maintains a separate memory region, the binding array, where variable bindings are stored and the variables' keys are used as indices into the binding arrays. This technique can be used for implementing the encapsulated search as well. As for a copying implementation, switching between two nodes of the tree can be implemented as a constant time operation; only the pointer to the current binding array has to be changed. When a computation is split, the program must only copy the current binding array. While the size of the binding array is potentially unbounded, it will be smaller than the whole graph and therefore copying it will require less time than in a copying the graph. On the downside, every reference to a variable's binding is now penalized with a small but constant overhead due to the indirect access.

Binding arrays can be regarded as a two dimensional matrix that is indexed by variable keys and processor numbers, using the processor number as major index. Organizing this matrix in another way, namely by using the variable keys as major index, leads to yet another implementation technique for multiple bindings. In this scheme, which we call the *binding tables* approach, every variable is associated with an array of values that is indexed by search space identifiers. When a computation is split, each binding table has to be extended by one slot for the new search space. In order to avoid recording all variables globally, the new element is added to a variable's binding table only when the program actually looks up the binding of the variable. For such an implementation creating new search spaces and switching between two search spaces are now constant time operations. However, determining the value of a variable is no longer a constant time operation. When a variable is used for the first time in a search space, the program must propagate the binding from the parent of the current space into the new slot. As the variable might not have been used in the parent space, this propagation may continue up to root of the search space tree.

3.4 Summary

None of the solutions presented so far is optimal in the sense that all of three basic operations search space creation, switching between two spaces, and determining the binding of a variable are performed in constant time. In fact, [GJ93] have proved that no implementation mechanism for the or-parallel execution of logic programs exists with the property that all of the operations creating a new node in the tree, switching between two nodes in the tree, and accessing or updating a variable can be executed in constant time. A fortiori, this result also applies to the implementation of the encapsulated search. Therefore, one has to identify the "usual" needs of the implementation and choose the technique which is best suited to the average tasks being executed. In performing this decision one should also take into account the effects that are entailed on the implementation by each of the approaches, including code generation and memory management.

4 The Curry Abstract Machine

The Münster Curry compiler[2] is an implementation of Curry that generates efficient native code for the target machine using a translation into C code. It is based on an abstract machine that is similar to the JUMP machine [CL97], an abstract machine developed for the execution of functional, logic, and functional logic programs.

Fig. 3 shows the syntax of abstract machine code programs. Due to lack of space we cannot present the operational semantics of the abstract machine in this paper. Every abstract machine code

p	::= d_1 `;` … `;` d_n	programs
d	::= `data` $t\ x_1 \ldots x_k$ = *cstrs*	declarations
	$\mid f\ x_1 \ldots x_k$ = *st*	
cstrs	::= $cd_1 \mid \ldots \mid cd_n$	
cd	::= $c\ ty_1 \ldots ty_k$	constructor declarations
ty	::= x	types
	$\mid t\ ty_1 \ldots ty_k$	
	$\mid ty_1 \rightarrow ty_2$	
st	::= `return` x	statements
	$\mid$ `enter` x	
	$\mid f\ x_1 \ldots x_k$	
	$\mid x \leftarrow st_1$ `;` st_2	
	$\mid$ `let` { x_1 = e_1 `;` … `;` x_n = e_n } `in` *st*	
	$\mid$ `switch` (`rigid` $\mid$ `flex`) x `of` { $a_1 \mid \ldots \mid a_n$ }	
	$\mid$ `choices` { $st_1 \mid \ldots \mid st_n$ }	
e	::= x	expressions
	$\mid$ `data` $c\ x_1 \ldots x_k$	
	$\mid$ `partial` $f\ x_1 \ldots x_n$	
	$\mid$ `lazy` $f\ x_1 \ldots x_k$	
	$\mid$ `lvar`	
a	::= `data` $c\ x_1 \ldots x_k \rightarrow st$	case alternatives
$x \in$ *Var*, $c \in$ *Con*, $f \in$ *Fun*, $t \in$ *Type*		

Figure 3. Syntax of Abstract Machine Code Programs

function returns (a pointer to) a node that is in head normal form. New nodes are introduced by `let`-statements. A `data` expression creates a new node representing a data constructor application. The number of arguments must match the arity of the data constructor. Similarly, a `lazy` expression creates a node representing an unevaluated function application. Again, the number of arguments must match the arity of the function. Nodes representing partial applications are created with a `partial` expression, and fresh logic variables are created with `lvar` expressions.

Abstract machine code functions always return nodes which are in head normal form. The `return` statement is therefore used only for returning a node that is known to be in head normal form, i.e., either a data constructor application, a partial application, or a logic variable. The `enter` statement performs a jump to the code that evaluates the node bound to variable x to head normal form. Statement sequencing is implemented with the statement $x \leftarrow st_1$ `;` st_2. This statement first executes the statement st_1 and binds the variable x to the node that is returned. The statement st_2 is then executed in the extended environment.

The `switch` statement is used for implementing pattern matching; it scrutinizes the node bound to x – which must be in head normal form – and selects the matching alternative. If x is bound to an

[2] Available at `http://danae.uni-muenster.de/~lux/curry`

uninstantiated variable, evaluation of the current thread is either suspended or the variable is non-deterministically instantiated, depending on whether the `switch` is `rigid` or `flex`ible. Finally, the `choices` statement is used for the implementation of non-deterministic alternatives.

As an example, Fig. 4 shows the abstract machine code for the function `append` from Fig. 1.

```
append a1 ys =
   a2 ← enter a1;
   switch flex a2 of {
     [] → enter ys
   | (:) x xs →
       let { a3 = lazy append xs ys; a4 = data (:) x a3 } in
       return a4
   }
```

Figure 4. Abstract Machine Code for `append`

5 Implementation Issues

At present, we have implemented two versions of the abstract machine which use trailing and copying, respectively, in order to implement the encapsulated search. In both implementations, nodes are represented by arrays in the heap, where the first element of the array is a pointer to a record that contains descriptive information about the node. This information includes the size of the node, a tag number which is used for distinguishing the data constructors in a data type, and the entry-point of the code that evaluates the node to head normal form. This entry-point is used by the `enter` statement of the abstract machine code. Note that this statement is used only when the compiler cannot prove that the node is in head normal form, i.e., usually only for arguments passed to a function or nodes that are returned from a function call. Fig. 5a shows the representation of the expression `append [1] [2]` in the heap where we represent the tag pointer by the name of the function or data constructor, respectively.

When a lazy application has been evaluated, the node is overwritten with an indirection to the result. This is shown in Fig. 5b for the evaluation of the `append` application. The program could overwrite the application node directly when the result is not a variable node and fits into the application node. This optimization has not been implemented at present. However, the garbage collector can remove such indirection pointers as will be explained below.

In the trailing implementation, the update of the application node may have to be recorded on the trail in order to be able to restore the unevaluated state of the application if it can be observed by the program. This makes the flat representation of applications from Fig. 5a an unfortunate choice, because we either have to overwrite all arguments of the node – and therefore save them to the trail – or risk a space leak in the program. In order to circumvent this problem, the trailing implementation uses a different representation of lazy applications which is shown in Fig. 5c. For every lazy application, two nodes are allocated in the heap. The first is a fixed size node that comprises only the descriptor, and a pointer to an argument vector. The argument vector has its own tag descriptor in order to facilitate garbage collection. Incidentally, the argument vector could be used for representing the application directly when the compiler can prove that the result of the application is not shared (cf. [LGH+93] for a corresponding analysis). With this representation of lazy applications, only the argument pointer has to be saved on the trail. However, it requires an additional cell to

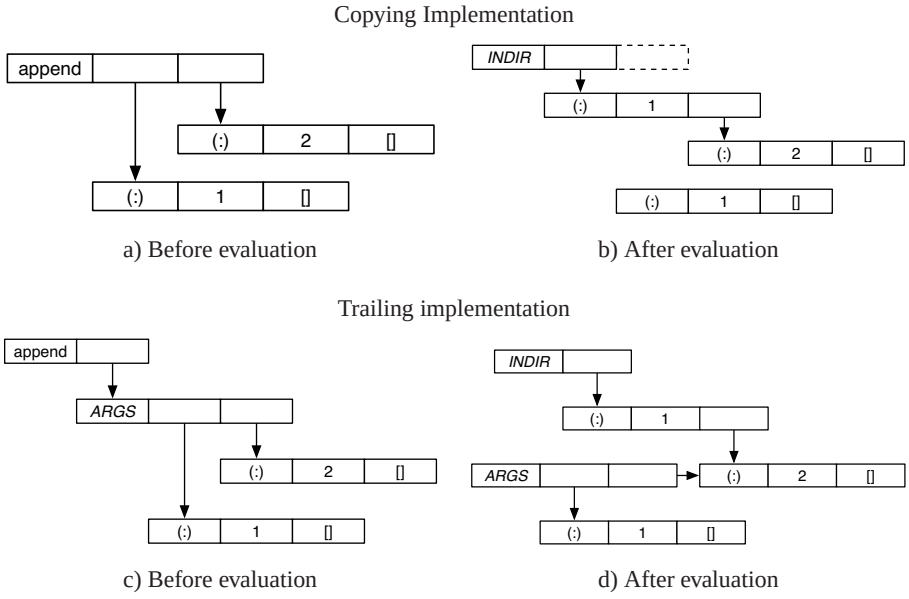


Figure 5. Representation of `append [1] [2]`

be allocated for each lazy application. Note that in a lazy language, without further analysis every application occurring in an argument position is a lazy application that can potentially be shared.

In order to avoid dereferencing indirection pointers that are introduced when a lazy application node is overwritten and also when a variable is bound, implementations of functional and logic languages replace pointers to indirection nodes by pointers to the referenced nodes during garbage collection.[3] There is no problem with implementing this optimization in the copying based implementation. However, when trailing is used, this optimization cannot be used. Consider that a garbage collection occurs in the program fragment

```
coin = 0
coin = 1
f x | x =:= y = ... (x + y) ... where y free
main = f coin
```

after the guard of `f` has been evaluated, but before the value of the addition `x + y` is computed. If the garbage collector replaces the references to indirection nodes resulting from the evaluation of `coin` and binding the variable `y` with the particular result of `coin`, the arguments of the addition become fixed and are not changed when the program starts exploring the search tree for the other solution of `coin`. Therefore, the trailing implementation does not short-circuit indirect references during garbage collection.

6 Experimental Results

The discussion in the previous section shows that the copying technique allows for a more streamlined implementation of the abstract machine. On the other hand, copying necessarily adds an overhead to the implementation of search strategies due to the copying of the graph corresponding to the

[3] This technique is known as variable shunting [SC91] in Prolog implementations.

search goal. This leaves the question whether the optimizations outweigh this overhead for practical problems. In order to give an answer to this question, we have run some benchmarks for both implementations of the abstract machine.

The results for the benchmarks are shown in Fig. 6. Execution and garbage collection times are measured in seconds, allocation in megabytes. The execution times are averaged over five runs of each benchmark problem. The last two columns show the ratios of execution time and memory allocation for the copying implementation with respect to the trailing implementation. The statistics were collected with the built-in facilities of the Münster Curry compiler. All benchmarks were executed on an Apple PowerBook equipped with a 550MHz PowerPC processor and 256 MBytes of main memory under Mac OS X 10.2.8.

	Copying			Trailing			Copy/Trail	
	user	gc	alloc	user	gc	alloc	user	alloc
best1st F	0.35	0.01	6	0.41	0.01	8	0.86	0.71
best1st D	0.58	0.01	10	0.60	0.01	12	0.97	0.84
boyer F	1.10	0.02	22	1.15	0.04	27	0.95	0.81
exp3_8 F	5.24	0.07	185	5.79	0.09	246	0.91	0.75
exp3_8 L	6.62	0.13	185	6.92	0.16	185	0.96	1.00
flparse 1	1.85	0.15	72	0.27	0.03	24	6.89	2.99
money F	0.82	0.00	22	0.80	0.00	32	1.02	0.71
money D	3.75	0.00	113	0.85	0.00	27	4.42	4.14
fib F	5.59	0.00	72	4.89	0.00	113	1.14	0.64
fib L	12.12	0.00	277	10.19	0.00	349	1.19	0.79
queens F	1.91	0.00	50	2.18	0.00	71	0.88	0.70
queens D	1.60	0.00	41	1.07	0.00	36	1.49	1.13
queens B	0.95	0.03	36	0.98	0.03	44	0.97	0.81
rev F	2.00	0.02	84	2.20	0.01	108	0.91	0.78
rev L	5.47	0.00	133	5.65	0.01	133	0.97	1.00
sieve F	0.81	0.00	12	0.87	0.00	17	0.92	0.73

F: functional
L: logical without search
D: depth-first search for all solutions
B: breadth-first search for all solutions
1: depth-first search for only one solution

Figure 6. Benchmark results

The best1st benchmark computes a solution for the 8-puzzle with 15 steps, boyer is the well-known Boyer benchmark, exp3_8 computes the eighth power of 3 on natural numbers. The flparse benchmark uses functional logic parsing combinators [CL99] in order to implement a simple calculator. This calculator is then applied to the string `1*...*1` containing 100 `1`s. Money computes the number of solutions of the SEND+MORE=MONEY puzzle, fib computes the 30th Fibonacci number using the naive algorithm, queens computes the number of solutions for the 8-queens problem (actually, the breadth-first solution uses only 7 queens), rev uses the naive reverse algorithm to reverse a list of 250 integers 100 times, and sieve computes the first 750 prime numbers using the classical Sieve of Erathosthenes. For some of the benchmarks we have made use of the possibility to implement a problem in different paradigms. This is indicated by the letter to the right of each benchmark.

In the set of benchmarks presented, the copying implementation is up to 10% faster in the case of purely functional code and a little bit faster for code written in a logic style, but without search. In the case of the n-queens benchmark implemented with a breadth-first search, the runtime does not differ

very much. However, for the benchmarks using depth-first search the copying implementation is significantly slower than the trailing implementation (up to a factor of nearly 7). The only exception is the best1st benchmark, where performance of both versions is nearly identical. This is due to the fact that in this benchmark the encapsulated search is used only for computing a list of possible moves at each position and therefore no deep search is involved in the problem.

As it could be expected, the copying implementation performs quite poorly as soon as a deep search is involved because of the large amount of copying that has to be performed. On the other hand, code without search performs slightly better for the copying implementation, which can be attributed to the more compact representation of lazy application nodes. While it is too early to draw final conclusions from this small set of benchmarks, we take the results as an indication that a trailing implementation is preferable to a copying one, at least as far as the implementation of copying in the Münster Curry compiler is concerned.

7 Related Work

Implementation techniques for managing multiple bindings have been studied for a while, among others, in the context of (or-)parallel implementations of logic languages. It has been shown that no technique can be optimal in the sense that all of the operations task creation, task switching, and variable access are implemented by constant time operations [GJ93,RPG99]. A fortiori, this result applies to the implementation of the encapsulated search in Curry that is more general than a parallel traversal of a goal's search tree.

With the exception of binding arrays, all of the presented implementation techniques have been used by different Curry implementations. An earlier implementation of the Münster Curry compiler, that was based on a less efficient stack-based abstract machine [LK99], was using only an extended trailing scheme [Lux99]. Unfortunately, it turns out that a pure trailing approach cannot implement the full encapsulated search. In order to be able to restore search continuations in arbitrary search spaces it is necessary to support copying as well. However, copying is rarely used in practice. It can occur only when a controlling search strategy continues the evaluation of a search goal without encapsulating the search, or when two different searches are combined. In addition, copying is applied to the final solution of a search goal.

The Curry2Java implementation [HS99] was the first to implement non-deterministic choice by independent threads in Java with the goal of providing fair search strategies. It was using binding tables in order to implement multiple bindings. The implementation was quite slow compared to a native code compiler like the Münster Curry compiler and development of the implementation has to stalled. NarrowMinder [AHMS01] has been designed to support fair evaluation of non-deterministic choice and uses copying.

The encapsulated search in Curry is a generalization of a similar feature introduced into the Oz language in order to control search [SSW94,Sch97]. [Sch99] also performs a comparison between copying and trailing implementations. However, the goal of that paper was only to show the feasibility of copying as an implementation mechanism. In contrast to our work, different implementations of constraint logic languages were compared in the paper.

8 Conclusion and Future Work

In this paper we have compared two different implementation techniques for encapsulated search in Curry, a feature that gives control to the programmer on how the search space of a goal is explored

and allows using non-deterministic search in the context of a declarative I/O mechanism. The experimental results for the Münster Curry compiler show that neither of the implementation techniques is strictly better than the other – a result that is not very surprising given the analysis from [GJ93] and [RPG99]. The copying implementation yields faster executables for purely functional programs due to some optimization that cannot be used with trailing. However, as soon as search is involved, the picture changes. For depth-first search algorithms the copying implementation was significantly slower than the trailing implementation.

The data presented in this paper can be regarded as an indication for trailing being more feasible for an implementation of the encapsulated search. However, this has to be backed by more experiments. In addition, the copying approach currently used is quite naive and should be improved.

Another topic of future research are implementations of the Münster Curry compiler using binding arrays and binding tables, respectively, in order to provide a more comprehensive comparison of implementation strategies.

References

AHMS01. Sergio Antoy, Michael Hanus, Bart Massey, and Frank Steiner. An implementation of narrowing strategies. In *Proc. PPDP'01*, pages 207–217. ACM, 2001.

ALN87. Hassan Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th ILPS*, pages 17–23, 1987.

Ant97. Sergio Antoy. Optimal non-deterministic functional logic computations. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Proc. ALP'97*, LNCS 1298, pages 16–30. Springer, 1997.

CL97. Manuel M.T. Chakravarty and Hendrik C.R. Lock. Towards the uniform implementation of declarative languages. *Computer Languages*, 23(2–4):121–160, 1997.

CL99. Rafael Caballero and Francisco Javier López-Fraguas. A functional-logic perspective on parsing. In Aart Middeldorp and Taisuke Sato, editors, *Functional and Logic Programming, 4th Fuji International Symposium, FLOPS'99*, LNCS 1772, pages 85–99. Springer, 1999.

GHPS94. Gopal Gupta, Manuel V. Hermenegildo, Enrico Pontelli, and Vítor Santos Costa. ACE: And/or-parallel copying-based execution of logic programs. In Pascal Van Hentenryck, editor, *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.

GJ93. Gopal Gupta and Bharat Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.

GSP95. Gopal Gupta, Vítor Santos Costa, and Enrico Pontelli. Shared paged binding array: A universal data-structure for parallel logic programming. In T. Chikayama and E. Tick, editors, *Proc. NSF/ICOT workshop on Parallel Logic Programming*, CIS-TR-94-04. University of Oregon, 1995.

Han03. Michael Hanus (ed.). Curry: An integrated functional logic language. Version 0.8. `http://www.informatik.uni-kiel.de/~mh/curry/report.html`, 2003.

HS98. Michael Hanus and Frank Steiner. Controlling search in declarative programs. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Proc. PLILP'98/ALP'98*, LNCS 1490, pages 374–390. Springer, 1998.

HS99. Michael Hanus and Ramin Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, Special Issue 1, 1999.

JD98. Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proc. POPL'98*, pages 158–168, 1998.

LGH+93. John Launchbury, Andy Gill, John Hughes, Simon Marlow, and Simon L. Peyton Jones. Avoiding unnecessary updates. In *Proc. Glasgow Workshop on Functional Programming*. Springer, 1993.

LK99. Wolfgang Lux and Herbert Kuchen. An efficient abstract machine for Curry. In Kurt Beiersdörfer, Gregor Engels, and Wilhelm Schäfer, editors, *Informatik '99 – GI Jahrestagung*, Informatik Aktuell, pages 390–399. Springer, 1999.

Lux99. Wolfgang Lux. Implementing encapsulated search for a lazy functional logic language. In Aart Middeldorp and Taisuke Sato, editors, *Functional and Logic Programming, 4th Fuji International Symposium, FLOPS'99*, LNCS 1772, pages 100–113. Springer, 1999.

MSS95. Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An abstract machine for Oz. In Manuel V. Hermenegildo and S. Doaitse Swierstra, editors, *Proc. PLILP'95*, volume 982 of *LNCS*, pages 151–168. Springer, 1995.

Pey03. Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press, 2003.

Red85. U. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. ILPS'85*, pages 138–151, 1985.

RPG99. Desh Ranjan, Enrico Pontelli, and Gopal Gupta. On the complexity of or-parallelism. *New Generation Computing*, 17(3), 1999.

SC91. Dan Sahlin and Mats Carlsson. Variable shunting for the WAM. Technical Report SICS RR R91.07, Swedish Institute of Computer Science, 1991.

Sch97. Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proc. CP'97*, LNCS 1330, pages 519–533. Springer, 1997.

Sch99. Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289. MIT Press, 1999.

SSW94. Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In Alan Borning, editor, *Proc. PPCP'94*, LNCS 874, pages 134–150. Springer, 1994.

Symbolic Representation of tccp Programs*

M. Alpuente[1], M. Falaschi[2], and A. Villanueva[1]

[1] DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain.
{alpuente,villanue}@dsic.upv.es.
[2] Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy. falaschi@dimi.uniud.it.

Abstract In this paper, we develop a symbolic representation for *timed concurrent constraint* (tccp) programs, which can be used for defining a new model–checking algorithm for reactive systems. Our approach is based on using streams to extend *Difference Decision Diagrams* (DDDs) which generalize the classical *Binary Decision Diagrams* (BDDs) with constraints. We use streams to model the values of system variables along the time, as occurs in many other (declarative) languages. Then, we define a symbolic (finite states) model checking algorithm for tccp which mitigates the state explosion problem that is common to more conventional model checking approaches. In particular, we show how the symbolic approach to model checking for tccp improves previous approaches based on the classical Linear Time Logic (LTL) model checking algorithm. **Keywords:** Timed Concurrent Constraint Programming, Model Checking, DDDs

1 Introduction

In the last decades, formal verification of industrial applications has become a hot topic of research. As the complexity of software systems increases, automatic verification tools which are able to guarantee the correct behavior of such systems are dramatically lacking. *Model checking* is a fully automatic formal verification technique which is able to demonstrate certain properties formalized as logical formulas which are automatically checked on a model of the system; otherwise, it provides a counterexample which helps the programmer to debug the wrong code.

The *concurrent constraint* paradigm (cc) was first introduced in [16] to model concurrent systems. A global store consisting of a set of constraints contains the information gathered during the computation. Constraints are dynamically added to the store which can also be consulted. The programming model was extended in [2] over a discrete notion of time in order to deal with reactive systems, that is, systems which continuously interact with their environment without producing a final result and execute infinitely along the time. The use of constraints and the notion of time which lay in tccp permit to program reactive systems in a very natural way. Reactive systems are usually modeled as concurrent systems which are more difficult to be manually debugged, simulated or verified than sequential systems. In previous works ([9,10,19]) we have defined an explicit model checking algorithm for tccp programs. Such method automatically constructs a model of the system which is similar to a Kripke Structure. Unfortunately, we are able to verify only small programs due to the explicit exploration of the graph.

Recent advances in model checking deal with huge state-spaces by using symbolic manipulation algorithms inside model checkers [5,7,13]. Other techniques such as abstract interpretation, partial evaluation, and on-the-fly methods have also been proposed in the literature as a mean to (partially) solve the state-space explosion problem [6].

The main purpose of this work is to improve the exhaustive model checking algorithm defined in the last years to verify tccp programs. Starting from the graph representation of [10], in this paper we formalize a symbolic representation of reactive systems specified in tccp. Such representation allows us to formulate a symbolic model checking algorithm which allows us to verify more complex reactive systems in tccp. In order to ensure the termination of our approach we refer to finite state systems in this work. It would be possible to remove this assumption and consider infinite

* This research is partially supported by the MCyT under grants TIC2001-2705-C03-01 and HU 2003-0003.

state systems by requiring the user to indicate a finite time interval for limiting the duration of tccp computations, as we did in [9,19]. This idea could be extended also to our new framework. To the best of our knowledge, we define the first symbolic model checking algorithm for tccp.

The paper is organized as follows. In Section 2 we introduce the tccp programming language and the tccp Structure constructed from the program specification and which is the reference point of this work. We introduce also an example which is used in the remaining sections to illustrate formal definitions. In Section 3 we introduce the verification method that we propose and in Section 4 we define the technical mechanisms that we need to apply the verification method. In particular we introduce the symbolic structure used to represent tccp programs. In Section 5 we show the algorithms that allow us to automatize the construction process and finally, in Section 6 we develop an example of property verification. Section 7 is devoted to conclusions and future work.

2 The tccp Framework

The cc paradigm has some nice features which can be exploited to improve the difficult process of verifying software: the declarative nature of the language ease the programming task of the user, and the use of constraints naturally reduces the state space of the specified system.

2.1 The tccp language

The *Timed Concurrent Constraint Language* (tccp) was developed in [2] by F. de Boer *et al.* as a framework for modeling reactive and real-time systems. It was defined by extending the concurrent computational model of the cc paradigm [16,18] with a notion of discrete time.

Basically, a cc program describes a system of agents that can add (*tell*) information into a store as well as check (*ask*) whether a constraint is entailed by such global store. The basic agents defined in tccp are those inherited from cc plus a new conditional agent described below. Moreover, a *discrete global clock* is provided. Computation evolves in steps of one time unit by adding or asking (entailment test) some information to the store. It is assumed that *ask* and *tell* actions take one time unit, and the parallel operator is interpreted in terms of maximal parallelism. Moreover, it is assumed that constraint entailment tests take a constant time independently of the size of the store[1].

Let us first recall the notion of cylindric constraint system as it is used in the cc paradigm. A simple constraint system can be defined as a set of tokens (or primitive constraints) together with an entailment relation. Examples of such constraint systems are the Herbrand constraint system, the FD constraint system [12] and the `Gentzen` constraint system [17].

Definition 1 (Simple constraint system [18]). Let D be a non-empty set of *tokens* (primitive constraints). A *simple constraint system* is a structure $\langle D, \vdash \rangle$ where $\vdash \subseteq 2^D \times D$ is an *entailment relation* satisfying:

C1 $u \vdash P$ whenever $P \in u$,
C2 $u \vdash Q$ whenever $u \vdash P$ for all $P \in v$ and $v \vdash Q$.

A *cylindric* constraint system consists of a simple constraint system plus an existential quantification operator which is monotonic, conservative and supports renaming. The existential quantification allows one to model local variables in a given agent. The formal definition of the notion of *cylindric constraint system* can be found in [2,11].

[1] In practice, some syntactic restrictions are imposed in order to ensure that these hypotheses are reasonable (see [2] for details).

In this work, we consider a specific constraint system which allows us to verify a class of software systems. In particular, we consider the traditional arithmetic for real numbers including addition, equality and order comparison. This part of the constraint system handles the information related to the constrained nature of the system. On the other hand, we are also interested to handle streams which in tccp are modeled as lists of terms. Each stream represents the value of a given system variable along the time. Intuitively, in the current time instant, the head of the list represents the value of a variable and the tail of the list models the future. The entailment relation for lists is specified by Clark's Equality Theory. For example, $[X|Z] = [a|Y]$ [2] entails $X = a$ and $Z = Y$.

We use $\mathcal{V}$ to denote the set of variables ranging over $\mathbb{R}$ (or $\mathbb{Z}$), and $\mathcal{LV}$ is the set of lists of such variables. From now, we will use $\mathbb{D} \in \{\mathbb{R}, \mathbb{Z}\}$ to denote arbitrarily one of the two domains. Roughly speaking, we define the set of tokens of our constraint system as the set of *difference constraints* of the form $X - Y \leq c$ and $X - Y < c$, and the set of *stream constraints* of the form $V = [\,]$, $V = [X|W]$ and $V = [c|W]$, where X and Y belong to $\mathcal{V}$, V and W are in $\mathcal{LV}$, and the constant c belongs to $\mathbb{D}$.

We define the set AP of atomic propositions as the set of tokens of the cylindric constraint system above. In the rest of the paper, we identify the notion of (finite) constraint with atomic propositions.

Let us now recall the syntax of tccp, defined in [2] as follows:[3]

Definition 2 (tccp Language). Let C be a cylindric constraint system. The syntax of agents of the language is given by the following grammar:

$$A ::= \mathsf{stop} \mid \mathsf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ A \mid A \,||\, A \mid \exists x\, A \mid \mathsf{p}(x)$$

where c, c_i are *finite constraints* of C. A tccp *process* P is an object of the form $D.A$, where D is a set of procedure declarations of the form $\mathsf{p}(x) :- A$, and A is an agent.

The stop agent terminates the execution whereas the $\mathsf{tell}(c)$ agent adds the constraint c to the store. Nondeterminism is modeled by the choice agent (written $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$) that executes nondeterministically one of the choices whose guard is satisfied by the store. The agent $A \,||\, A$ represents the concurrent component of the language, and $\exists x\, A$ is the existential quantification, that makes the variable x local to the agent A. The agent for the procedure call is $\mathsf{p}(x)$.

Finally, the now c then A else B agent (called conditional agent) is the new agent (w.r.t. cc) which allows us to describe notions such as *timeout* or *preemption*. This agent executes A if the store entails c, otherwise it executes B.

2.2 The tccp Structure

The reference point of this work is a model of tccp programs introduced in [10], which essentially consists of a graph structure. The main difference w.r.t. a Kripke Structure is in the definition of the states. A state in a tccp Structure represents a set of states of a Kripke Structure since it contains a conjunction of constraints instead of a valuation of the system variables. Formally, a tccp Structure is as follows. In [10] the reader can find how to automatically obtain the tccp Structure from a given tccp program.

Definition 3 (tccp *structure*). Let AP be a set of atomic propositions. We define a tccp Structure M over AP as a 5-tuple $M = (S, S_0, R, C, T)$, where

[2] We follow the Prolog notation for lists.

[3] The operational and denotational semantics of the language can be found in [2].

1. S is a finite set of states,
2. $S_0 \subseteq S$ is the set of initial states,
3. $R \subseteq S \times S$ is a transition relation,
4. $C : S \rightarrow 2^{AP}$ is the function that returns the set of atomic propositions which hold in a given state, and
5. $T : S \rightarrow 2^{L}$ is the function that returns the set of labels in a given state.

Informally, labels are used to identify the point of execution of the program. Each occurrence of every agent of a program is labelled, thus the set of labels in a given state represents the set of agents that must run in such execution point.

2.3 The scheduler example

In Figure 1 we show a tccp program which we use to motivate different points of the paper. The program consists of a predicate with three output variables. We use streams to simulate the values of the system variables along the time, since the constraint system in tccp is monotonic (see [2] for details).

Intuitively, the program gets the value of variables `D1`, `T1` and `E1` by calling the auxiliary process `get_constraints`. These variables represent the duration of three different tasks of the process of building a house. This is executed in parallel with an ask agent which simply checks if the values of the variables are integer numbers and, in that case, some constraints are added to the global store which contains the available information of the system. Finally, a recursive call to the building process is made which would allow to recalculate the planning schedule.

```
build([PD|PD_],[PT|PT_],[PE|PE_]) ::=
     ∃ D1,T1,E1 (get_constraints(D1,T1,E1) ||
     ask(atom(D1),atom(T1),atom(E1)) →
        (tell(PD+D1 =< PT) ||
        tell(PT+T1 =< PE) ||
        tell(PE+E1 =< PA)) ||
        build(PD_,PT_,PE_)).
```

Figure 1. Example of a tccp program

The tccp Structure associated with this code is shown in Figure 2. The black circle indicates the initial state of the graph. We have simplified the structure by showing, in each state, only the new information added to the store. At each state, we also show the set of labels (beginning with the character 'l') representing the agents that must be executed, and the local variables.

The most important point of this example is the fact that we have added to the store only constraints of the form `V1+C=<V2` which can also be written as $V1 - V2 \leq C$ being C an integer or real constant. This kind of constraints appears in applications where, for example, we compare two clocks of a system to control the timing between tasks, or in scheduling applications such as this example. In the following sections, we show how we can symbolically represent this kind of constraints in a similar way as *Binary Decision Diagrams* (BDDs) do in the basic symbolic model checking approach.

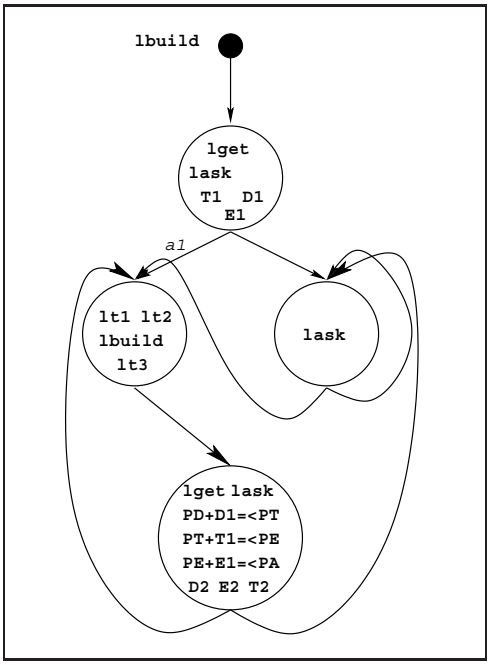


Figure 2. tccp Structure of `build`

3 Symbolic Model Checking

The idea of symbolic model checking is to represent the graph structure (the model) as a boolean formula, and then transform it into the efficient structure of BDD. In our approach, we aim to represent the tccp Structure as a formula with difference constraints and logical streams, and then transform it into a suitable extension of BDDs.

In [10,19], a logic dealing with constraints was proposed as the basis to develop a classical LTL model checking algorithm based on a tableau algorithm. The most important advantage of this approach is that the use of constraints leads to a compact representation of the system which we also exploit to effectively check properties on the model. Unfortunately, the expected state-explosion problem shows up when we combine the model with the property that we want to verify.

By considering the constraint system defined in Section 2.1 for the tccp language, the model which can be automatically obtained by following [10,19] only contains difference and stream constraints. Thus, our aim is to represent the tccp *Structure* by means of a new symbolic structure called DDD+LSs, then we use the efficient algorithms for checking DDD+LSs in order to verify tccp programs. In the following, we formalize our verification strategy and illustrate it by means of an example. We use the temporal logic with constraints of [3] to specify the properties we are interested to verify.

3.1 tccp *Structures* as logic formulas

A tccp *Structure* can be translated into a formula of the logic underlying our constraint system similarly as it is done for boolean functions in the classical symbolic approach. The idea is to encode states and to represent the relation R (i.e., the arcs of the graph) with a logic formula which is defined from the labels and the store of states.

Once we have the formula, we can construct a symbolic BDD-like structure corresponding to the formula, which represents an encoding of the system.

Let us explain how to obtain the formula by using the graph example shown in Figure2. Each arc of this graph corresponds to an element in the relation R. Now we can encode each arc as a conjunction of constraints. For example, the formula

$$\texttt{lget} \wedge \texttt{lask} \wedge \texttt{T1} \wedge \texttt{D1} \wedge \texttt{E1} \wedge \texttt{lt1}' \wedge \texttt{lt2}' \wedge \texttt{lbuild}' \wedge \texttt{lt}' \quad (1)$$

represents the arc labelled with $a1$.In the following, we call *arc-formula* the logic formula representing an arc of the tccp Structure. Note that we have used primed versions of agent labels in order to express their value in the following time instant. This is equivalent to the use of program counters in (imperative) classical model checking approaches.

It is easy to see that the R relation can be represented by a disjunction of arc-formulas. The resulting formula is the subject of our next task: we have to symbolically represent this formula and, for this purpose, we define a new structure (similar to a BDD) and the algorithms which automatically construct it from the formula.

4 The Symbolic Structure

Difference Decision Diagrams (DDDs) are an extension of the Binary Decision Diagrams defined in [4] to symbolically represent *difference constraint expressions*. Difference constraint expressions are formulas of a logic extended with difference constraints. Difference constraints are inequalities of the form $x - y \leq c$ where x and y are integer or real-valued variables, and c is a constant. A difference constraint expression consists of difference constraints combined with boolean connectives. $d \rightarrow a, b$ where d is a difference constraint and a and b are difference constraint expressions means that, if d holds, then a, else b.

DDDs and BDDs share some common features. For example, both BDDs and DDDs can be ordered and reduced, and the algorithms to handle them are quite similar. A drawback of DDDs is the fact that maintaining them as a canonical data structure is more expensive than for BDDs.

It is important to remark that, in order to correctly represent tccp Structures, we cannot directly use boolean structures such as BDDs. Nodes in DDDs contain constraints which can encode some implicit information whereas nodes in BDDs contain only boolean variables [14]. This implicit information is the main reason why a DDD-like structure is necessary. This is also the reason why, if we reduce a DDD following the ideas of BDDs, then we do not obtain a canonical representation for the considered difference constraint expression, as opposed to the case of Ordered BDDs. However, it is still possible to obtain a semi-canonical[4] structure which can be used to decide satisfiability, validity, falsifiability and unsatisfiability of expressions. There is also an algorithm to obtain a canonical representation of DDDs which is quite expensive ([15]).

Even though we can use DDDs to represent difference constraints, we need to model also constraints over streams (represented as logical lists in tccp). Therefore, we need to extend the expressivity of DDDs and to redefine the algorithms which automatically construct the DDD Structure from a given formula.

4.1 Extending Difference Decision Diagrams with Logical Streams

As we have shown in Section 3, we need to represent symbolically a graph structure which contains difference constraints and stream constraints.

[4] A DDD is semi-canonical if (i) an expression ϕ is represented by **1** iff ϕ is valid, and (ii) an expression ϕ is represented by **0** iff ϕ is unsatisfiable.

Formally, we define *Difference Decision Diagrams + Logical Streams* (DDD+LSs) to handle the logic defined by the following grammar:

$$\phi ::= x - y \leq c \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x.\phi \mid X = [x|Y] \mid X = [c|Y] \mid X = [\,]$$

where the constant c belongs to $\mathbb{D}$, and $X, Y \in \mathcal{V}$ denote variables. The grammar is extended as usually with the derived operators $x - y < c$, $\phi_1 \vee \phi_2$ and $\forall x.\phi$. Similarly to DDDs, equality can be modeled by using the $<$ and $\leq$ operators.

Similarly to DDDs, a DDD+LS is a directed acyclic graph (V, E) where V is a set of vertices and E a set of arcs connecting pairs of vertices. The set V contains two terminal vertices with out-degree zero (called **0** and **1**). In addition, V contains a set of non-terminal vertices with out-degree two. Each non-terminal vertex v has nine attributes which can be classified in three subsets: (i) the first three attributes ($pos(v)$, $neg(v)$ and $const(v)$) which are defined in the case when the node v represents a difference constraint (otherwise they are set to $\perp$), (ii) the attributes $left(v)$, $head(v)$ and $tail(v)$ that stand for the case when v represents a stream constraint (otherwise they are set to $\perp$), and (iii) the last three attributes $op(v)$, $high(v)$ and $low(v)$ which are defined in both cases. Intuitively, $op(v)$ determines which kind of expression represents v: if $op(v) \in \{\text{LE}, \text{LEQ}\}$, then v represents the expression $pos(v) - neg(v)\ op(v)\ const(v)$; otherwise (if $op(v) = \text{LIST}$), then v represents the stream constraint $left(v) = [head(v)|tail(v)]$ The remaining two attributes ($high(v)$ and $low(v)$) represent the two branches that can be followed from the non-terminal vertex v in the graph.

Some shorthands are defined to reference combinations of attributes. Notation $var(v)$ represents the pair $(pos(v), neg(v))$ whereas we use $bnd(v)$ to refer to the pair $(op(v), const(v))$. By $varl(v)$ we represent the pair $(left(v), tail(v))$ and $listExp(v)$ is the pair $(head(v), varl(v))$. Finally, we denote by $attr(v)$ the set of attributes of the node v.

The set of edges E is defined as the set of pairs of the form $(v, low(v))$ and $(v, high(v))$, where $v \in V$ and v is not a terminal vertex.

A node of a DDD+LS Structure represents an expression which can be either a difference constraint (as in DDDs) or a stream constraint. The semantics of DDD+LS nodes is formalized in Definition 4 which just adds to DDDs the semantics derived from the new DDD+LS attributes. $\mathbf{Exp}$ stands for difference constraint expressions and stream expressions. The auxiliary function ($op(v)$) is used to check whether the node represents a difference constraint expression (returning values LE or LEQ), or a list expression (returning value LIST).

Definition 4. Let v be a vertex of a DDD+LS Structure. We define the function $\mathcal{S} : V \rightarrow \mathbf{Exp}$:

$$\begin{array}{l}
\mathcal{S}[\![\mathbf{0}]\!] \stackrel{\text{def}}{=} \mathit{false} \\
\mathcal{S}[\![\mathbf{1}]\!] \stackrel{\text{def}}{=} \mathit{true} \\
\mathcal{S}[\![v]\!] \stackrel{\text{def}}{=} \begin{cases}
(pos(v) - neg(v) < const(v)) \rightarrow \ \mathcal{V}[\![high(v)]\!], \mathcal{V}[\![low(v)]\!] & \text{if } op(v) = \text{LE}, \\
(pos(v) - neg(v) \leq const(v)) \rightarrow \ \mathcal{V}[\![high(v)]\!], \mathcal{V}[\![low(v)]\!] & \text{if } op(v) = \text{LEQ}, \\
(left(v) = [head(v)|tail(v)]) \rightarrow \ \mathcal{V}[\![high(v)]\!], \mathcal{V}[\![low(v)]\!] & \text{if } op(v) = \text{LIST}
\end{cases}
\end{array}$$

For example, the meaning of the first row in the semantics of v means that, if $pos(v) - neg(v) < const(v)$ holds, then $\mathcal{V}[\![high(v)]\!]$ is true, otherwise $\mathcal{V}[\![low(v)]\!]$ holds.

We show in Figure 3 a DDD+LS graph representing the formula in (2).

$$\texttt{PD} - \texttt{PT} =< 4 \wedge \texttt{PT} - \texttt{PE} =< 7 \wedge \texttt{P} = [\texttt{PT} \mid \texttt{PT\_}] \tag{2}$$

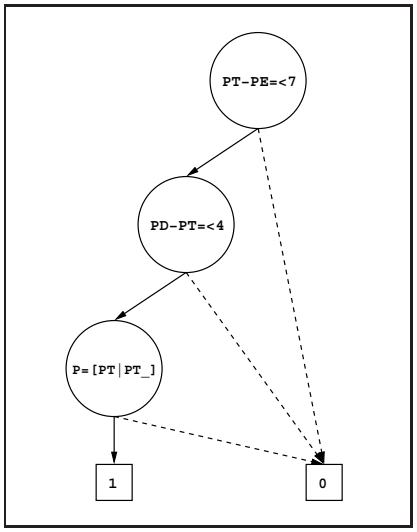


Figure 3. Example: DDD+LS from the formula in (2).

In order to obtain an ordered graph structure, we extend the total order on the vertices of the graph defined in [14] to consider the new attributes in DDD+LS. First of all, we assume an order between variables. We require that both, pairs of variables of a vertex corresponding to the difference expression, as well as pairs of list variables in the list expression, are *normalized*. This means that $pos(v) > neg(v)$ and $left(v) < tail(v)$[5]. Then we assume that LIST $<$ LE $<$ LEQ. Finally, tuples formed by the set of attributes in a specific vertex u, i.e., tuples of the form $(pos(v), neg(v), op(v), const(v), left(v),$
$head(v), tail(v))$, are ordered lexicographically. Note that $\forall X, \perp \leq X$ where $X \in \{\mathcal{V}, \mathcal{LV}\}$. We also have that for any constant c, $\perp \leq c$.

By using the order $<$ above, the notion of *Ordered* DDD+LS (called ODDD+LS in short) is formalized:

Definition 5 (Ordered DDD+LS). Let O be a DDD+LS Structure. We say that O is an *Ordered DDD+LS* (ODDD+LS) Structure if each non-terminal vertex v defined in O satisfies the following properties:

1. $neg(v) < pos(v)$,
2. $left(v) < tail(v)$,
3. $var(v) < var(high(v))$
4. $varl(v) < varl(high(v))$,
5. $var(v) < var(low(v)) \vee (var(v) = var(low(v)) \wedge bnd(v) < bnd(low(v)))$,
6. $varl(v) < varl(low(v)) \vee (varl(v) = varl(low(v)) \wedge head(v) < head(low(v))$

Intuitively, nodes containing difference expressions will appear in the graph structure before the nodes containing stream expressions. The first two conditions in Definition5 require that variables in a given node are normalized. The third and fourth requirements establish that, given a node v, variables of the child in the high branch of v must be greater than the variables in v. The last two points ensure that variables of the child in the low branch must be greater or equal than the variables

[5] We note that this property is reasonable when we use this kind of constraints to model streams, as it simply establishes that the list on the right hand side of the constraint is greater than the other one.

of v; if they are equal, then $bnd(low(v))$ ($head(low(v))$), must be greater than $bnd(v)$ ($head(v)$), respectively. The structure in Figure 3 is an ODDD.

In order to verify properties, we can consider semi-canonical structures as mentioned before. To get them, we propose some local and path reductions for ODDD+LSs, which are inspired in the reductions defined for Ordered DDDs.

Definition 6 (Locally Reduced DDD). Let D be an ODDD+LS with domain $\mathbb{D} \in \{\mathbb{N}, \mathbb{Z}\}$, and let u and v be non-terminal vertices of D. Then D is a *Locally Reduced DDD+LS* (LRDDD+LS) if it satisfies:

1. if $\mathbb{D} = \mathbb{Z}$ then, for all v, $op(v) = \text{LEQ}$ or $op(v) = \text{LIST}$,
2. for all v and u, if the set of attributes of u is identical to the set of attributes of v, then $u = v$,
3. for all v, $low(v) \neq high(v)$,
4. for all v, if $var(v) = var(low(v))$ then $high(v) \neq high(low(v))$,
5. for all v, if $list(v) = list(low(v))$ then $high(v) \neq high(low(v))$

The intuition of the first item is that, if the domain of the structure are integers, then we can eliminate any occurrence of the LE operator since it can be reduced to the LEQ operator by decreasing in one unit the value of the constant and then comparing with LEQ. The second condition ensures that there is no pair of different nodes with the same attributes. The rest of requirements avoid redundant tests on the same variables.

The next step towards the semi-canonical representation of DDD+LSs is the formalization of the notion of path reduction. We first need to define the semantics of edges and paths. By abuse, we define the negation of lists as the absence of information. That is, when we negate a stream expression we mean that the current store does not entail it.

Definition 7. Let u, v be vertices of a DDD+LS Structure. Let u and v be two adjacent vertices. The function $\mathcal{E} : E \rightarrow \mathbf{Exp}$ is defined as follows:

$$\mathcal{E}[\![(u,v)]\!] \stackrel{\text{def}}{=} \begin{cases} (pos(u) - neg(u) < const(u)) & \text{if } v = high(u) \text{ and } op(v) = \text{LE}, \\ (pos(u) - neg(u) \leq const(u)) & \text{if } v = high(u) \text{ and } op(v) = \text{LEQ}, \\ \neg(pos(u) - neg(u) < const(u)) & \text{if } v = low(u) \text{ and } op(v) = \text{LE}, \\ \neg(pos(u) - neg(u) \leq const(u)) & \text{if } v = low(u) \text{ and } op(v) = \text{LEQ} \\ left(v) = [head(v)|tail(v)] & \text{if } v = high(u) \text{ and } op(v) = \text{LIST}, \\ \neg(left(v) = [head(v)|tail(v)]) & \text{if } v = low(u) \text{ and } op(v) = \text{LIST}. \end{cases}$$

The notion of *path* in a DDD+LS Structure is defined as a finite sequence of edges of the form $\langle (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \rangle$. We say that such path has length k. The semantics of a path is defined as the conjunction of all difference constraints, negated difference constraints, stream constraints, and negated stream constraints in the path.

Now we are ready to define the notion of path reduction which provides us the semi-canonical representation. We denote by PRDDD+LS the structure resulting from applying this reduction step to a LRDDD+LS. A semi-canonical representation has exactly one DDD+LS to denote valid expressions and also a single DDD+LS for unsatisfiable expressions.

Essentially, we can identify redundant edges regarding difference constraint expressions by checking how expressions divide the domain. Each edge e_i splits the domain into two disjoint subsets. If one of these subsets is empty, then we know that the edge is redundant. This is the method applied in [14]. Regarding nodes representing stream expressions, since we have defined the negation as the absence of information, then the domain is split into two *possibly non* disjoint subsets, and no path-reduction can be done.

Theorem 1 allows us to check properties in the PRDDD+LS in a safe way. We know that the expression ϕ_u represented by the node u is valid if and only if $u = \mathbf{1}$. If $u = \mathbf{0}$, then the expression is unsatisfiable. If u is a non terminal vertex, then we know that the expression is both satisfiable *and* falsifiable. The proof of this result can be found in [1].

Theorem 1 (semi-canonicity). *In a PRDDD+LS, the terminal vertex **1** is the only representation of valid expressions, and the terminal vertex **0** is the only representation of unsatisfiable expressions.*

In Figure 4 we show a DDD+LS Structure representing the same formula in (2), which has a redundant edge (the second one from the top). We know that this node is redundant since the part of the domain for which the constraint is not satisfied is empty. Thus we could eliminate it obtaining the DDD+LS shown in Figure 3

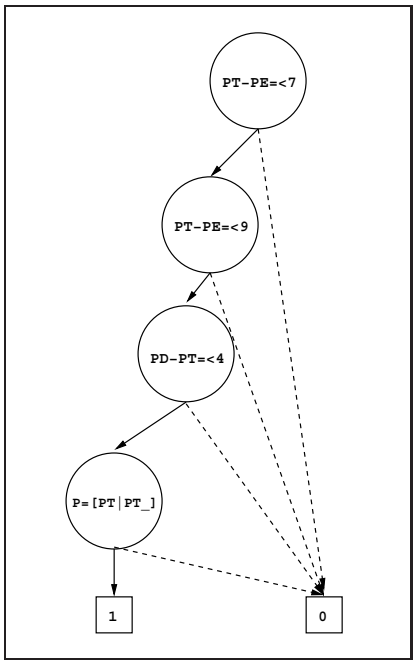


Figure 4. Example: Non Path-reduced DDD+LS representing the formula in (2).

5 Construction of DDD+LSs

In this section we show the algorithms which automatically construct a Locally Reduced DDD+LS from a given formula. In the rest of the section we assume that we are always considering Locally Reduced DDD+LS. Vertices and edges of the DDD+LS are stored in a graph data structure simply called *Graph*. Let G be a *Graph*. Initially, G contains only the two terminal vertices **0** and **1**. The set of edges of G are implicitly stored via the attributes of its vertices.

Let us introduce some functions which allow us to access the information or modify the structure. First, $\mathsf{insert}(G, a)$ creates a new vertex v in G with attribute a, and returns v. The function $\mathsf{member}(G, a)$ returns true if there exists a vertex in G with attribute a. Finally, $\mathsf{lookup}(G, a)$ returns the vertex in G with attribute a.

We also need some operators which obtain information about the attributes of a graph. Since names and behavior of these operations are very intuitive, we will use the name of attributes directly in our pseudocode.

In the following, we extend the algorithms in [14] (defined to construct DDDs), to construct the DDD+LS Structure. We also develop the new algorithms to deal with stream expressions.

```
vertex MKD(G: graph, x ∈ V, y ∈ V, o: operator,
           c ∈ D, h: vertex, l: vertex)
  if D = Z ∧ o = LE then c := c − 1
                         o := LEQ
  if member(G, (x, y, o, c, ⊥, ⊥, ⊥, h, l)) then
     return lookup(G, (x, y, o, c, ⊥, ⊥, ⊥, h, l))
  else if l = h then return l
       else if (x, y) = var(l) ∧ h = high(l) then return l
            else return insert(G, (x, y, o, c, ⊥, ⊥, ⊥, h, l))
```

Figure 5. MKD Algorithm

In Figure 5, the algorithm MKD for difference constraints is given as an extension of the algorithm presented in [14]. We have modified the attributes of nodes to take into account that nodes in DDD+LSs have three additional attributes.

The algorithm MKL is presented in Figure 6. It builds the vertex representing the stream expression $x = [y|z] \rightarrow h, l$.

```
vertex MKL(G: graph, x ∈ LV, y ∈ V, z ∈ LV, o: operator,
           h: vertex, l: vertex)
  if member(G, (⊥, ⊥, LIST, ⊥, x, y, z, h, l)) then
     return lookup(G, (⊥, ⊥, LIST, ⊥, x, y, z, h, l))
  else if l = h then
          return l
       else if (x, z) = plist(l) ∧ h = high(l) then
               return l
            else
               return insert(G, (⊥, ⊥, LIST, ⊥, x, y, z, h, l))
```

Figure 6. Algorithm MKL that creates a vertex for a list expression

These two algorithms have some preconditions which are similar to those for DDDs. For example, pairs of variables must be normalized. In [14], some functions are given which normalize pairs of variables before constructing vertices. We can use similar procedures for the difference expressions contained in our DDD+LS Structures. A novel precondition for the MKD algorithm is that we must ensure that we are dealing with difference constraints (namely, $op(v) \neq \text{LIST}$). Similarly, for the MKL algorithm we require that $op(v) = \text{LIST}$.

The next step for the construction of the DDD+LS Structure, is to define the algorithms which combine difference and stream expressions with boolean operators. The idea is to recursively apply a specific operator to all the vertices in the DDD Structure. In [4], this procedure is called APPLY. The same idea can be used for our DDD+LS Structure. The APPLY algorithm returns a DDD which is locally reduced, hence it is still necessary to path reduce the resulting DDD.

We have called APPLYLS the corresponding algorithm for DDD+LSs. We show in Figure 7 this algorithm which follows closely the design of APPLY with some little adjustment to include the handling of the list expressions. In the pseudocode, 'Connective' denotes a boolean connective of the logic. Moreover, eval is a function which takes the two terminal vertices and a boolean connective as input and returns the truth value depending on the boolean connective.

```
Vertex ApplyLS(G: graph c: Connective, u: Vertex, v: Vertex)
r: Vertex
if u, v ∈{0,1} then return eval(c, u, v)
else if member(G, (c, u, v)) then return lookup(G, (c, u, v))
  else if var(u) < var(v) then
    if op(u) = LIST then
        r ← MkL(left(u), head(u), tail(u),ApplyLS(c, high(u), v), ApplyLS(c, low(u), v))
      else r ← MkD(var(u), bnd(u), ApplyLS(c, high(u), v), ApplyLS(c, low(u), v))
      return r
  else if var(u) = var(v) then
    if bnd(u) < bnd(v) ∧ op(u) = LIST then
      r ← MkL(left(u), head(u), tail(u), ApplyLS(c, high(u), high(v)), ApplyLS(c, low(u), v))
    else if bnd(u) < bnd(v) ∧ op(u) ≠ LIST then
      r ← MkD(var(u), bnd(u), ApplyLS(c, high(u), high(v)), ApplyLS(c, low(u), v))
    else if bnd(u) = bnd(v) ∧ op(u) = LIST then
      r ← MkL(left(u), head(u), tail(u), ApplyLS(c, high(u), high(v)),
            ApplyLS(c, low(u), low(v)))
    else if bnd(u) = bnd(v) ∧ op(u) ≠ LIST then
      r ← MkD(var(u), bnd(u), ApplyLS(c, high(u), high(v)), ApplyLS(c, low(u), low(v)))
    else if bnd(u) > bnd(v) ∧ op(v) = LIST then
      r ← MkL(left(v), head(v), tail(v), ApplyLS(c, high(u), high(v)),
            ApplyLS(c, u, low(v)))
    else if bnd(u) > bnd(v) ∧ op(v) ≠ LIST then
      r ← MkD(var(u), bnd(u), ApplyLS(c, high(u), high(v)), ApplyLS(c, u, low(v)))
    else if var(u) > var(v) then
      if op(u) = LIST then
        r ← MkL(left(v), head(v), tail(v), ApplyLS(c, u, high(v)), ApplyLS(c, u, high(v)))
      else r ← MkD(var(v), bnd(v), ApplyLS(c, u, high(v)), ApplyLS(c, u, high(v)))
```

Figure 7. Algorithm APPLYLS

6 Verification

In this section we show how the symbolic structure can be used to formalize a symbolic model checking method for tccp programs. Assume that we express the property that we want to verify by using a CTL logic [6], where the atomic propositions of the logic are the same set of atomic propositions of the constraint system considered above. Note that we can use the defined entailment relation to obtain the truth value of formulas (see [3]).

We illustrate the method by a simple example. Assume that we want to verify that whatever we check that the variables have been assigned, there exists a successor where the same check is done. This property is expressed by the following formula. We use the standard notation for the temporal

operators, thus $\mathbf{AG}(f)$ is the logic operator meaning that the formula f holds at each state in the future and $\mathbf{EX}(g)$ means that there exists a successor state where g is satisfied.

$$\mathbf{AG}(\neg lask \vee \mathbf{EX}(lask)) \tag{3}$$

The classical symbolic model checking algorithm would take the formula in (3) as input and would return an OBDD representing the set of states of the system satisfying that formula. Temporal operators of the logic are represented as fix-points ([6]) and then, symbolic structures are manipulated. In our approach we would substitute OBDDs by DDD+LSs and the CTL logic is interpreted over constraints.

The formula $\mathbf{AG}(f)$ is equivalent to $f \wedge \mathbf{AX}(f)$ where **AX** means that the formula holds at each successive state. [6] shows that is possible to associate a fix-point operator to each CTL formula. Thus, we consider the operator associated to $f \wedge \mathbf{AX}(f)$. This operator allows us to compute a (greatest) fix-point which corresponds to the set of states starting from which the property to be proven holds. Finally, if all initial states of the model (the tccp Structure) are included in the fix-point, then the formula holds in the system. In our example, this algorithm [6] proves that the formula holds.

7 Conclusions

We have generalized DDDs to a new structure which allows us to represent tccp programs symbolically. We have introduced the corresponding notions and algorithms for automatically construct the symbolic structures and we have shown how they can be used within a symbolic model checking method. We think that this novel symbolic methodology improves the automatic verification of reactive systems specified as tccp programs as it reduces the search space significantly.

As future work, we plan to extend the language to consider constraint expressions more general than difference constraints.

[8] presents a different data structure called CST, which allows one to represent integer linear constraints symbolically and is used to define a parameterized verification method for (infinite state) Petri nets.

We also plan to implement our method and compare it with the exhaustive algorithm defined in previous works in order to quantify the improvement of the symbolic method w.r.t. the exhaustive one.

References

1. M. Alpuente, M. Falaschi, and A. Villanueva. Symbolic Model Checking for Timed Concurrent Constraint Programs. Technical Report DSIC-II/19/03, DSIC, Technical University of Valencia, 2003. Available at www.dsic.upv.es/users/elp/villanue/papers/ techrep03.ps.
2. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161:45–83, 2000.
3. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In G. Smolka, editor, *Proceedings of 8th International Symposium on Temporal Representation and Reasoning*, pages 227–233. IEEE Computer Society Press, 2001.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2002.

6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.
7. E. M. Clarke, K. M. McMillan, S. Campos, and V. Hartonas-GarmHausen. Symbolic Model Checking. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer-Verlag, July/August 1996.
8. G. Delzanno, J.-F. Raskin, and L. Van Begin. CSTs (Covering Sharing Trees): compact Data Structures for Parametrized Verification. *Software Tools for Technology Transfer*, 2003. . To appear.
9. M. Falaschi, A. Policriti, and A. Villanueva. Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language - I. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Selected papers from 2000 Joint Conference on Declarative Programming*, volume 48 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000.
10. M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. 2003. Submitted for publication.
11. L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras Part I*. North–Holland, Amsterdam, 1971.
12. P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(fd). Technical report, Computer Science Department, Brown University, 1992.
13. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.
14. J. Møller and J. Lichtenberg. Difference Decision Diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark, 1999.
15. J. Møller, J. Lichtenberg, H.R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proceedings of the 13th International Workshop on Computer Logic Science*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, 1999.
16. V. A. Saraswat. Concurrent Constraint Programming Languages. In *PhD Thesis, Carnegie-Mellon University*, 1989.
17. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Programming in Timed Concurrent Constraint Languages. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI*, pages 361–410, Berlin, 1994. Springer-Verlag.
18. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of 18th Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, New York, 1991. ACM Press.
19. A. Villanueva. *Model Checking for the Concurrent Constraint Paradigm*. PhD thesis, University of Udine and Technical University of Valencia, May 2003.

A Fixed Point Semantics for an Extended *CLP* Language

Miguel García-Díaz and Susana Nieva *

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
{miguel,nieva}@sip.ucm.es

Abstract. In [10] an extension of traditional Logic Programming was introduced combining two improving approaches. On one hand, extending Horn logic to hereditary Harrop formulas, in order to enhance the expressive power, and on the other incorporating constraints, in order to improve the efficiency. The constraint logic programming language obtained from such combination was in need of a declarative semantics. In this paper, we present a fixed point semantics for it. Taking as starting point the technique used by Miller to interpret intuitionistic implication in goals, we have formulated a novel extension to deal with universal quantifiers and constraints. The corresponding theorems of soundness and completeness are proved.

1 Introduction

One of the main features of Logic Programming (LP) is that, in a logic program, the operational interpretation and the mathematical (declarative) meaning agree each other, in the sense that the declarative meaning of a program can be interpreted operationally as a goal-oriented search for solutions. In [13] the notion of abstract logic programming language is formulated as a formalization of this idea. There the declarative meaning of a program is identified with the set of goals that can be proved from it by means of uniform proofs in a deduction system. Several logic extensions of Horn logic, both of first and higher order, have been proved to be abstract logic programming languages that enhance the weak expressive power of logic programs based on Horn clauses ([13,14]). This is also the case of the language $HH(\mathcal{C})$, on which the present paper focuses. It was introduced in [10] as a combination of the logic of Hereditary Harrop formulas (HH) and Constraint Logic Programming (CLP), obtaining a scheme $HH(\mathcal{X})$ that may be particularized with any constraint system $\mathcal{C}$, providing for an instance $HH(\mathcal{C})$. This language is not only an extension of traditional LP (based on Horn logic) improving its expressivity, but also incorporating the efficiency advantages of CLP [7]. HH extends Horn logic including disjunctions, intuitionistic implications and universal quantifiers in goals. These constructions are essential in capturing module structure, hypothetical queries and data abstraction. On the other hand, the purpose of the incorporation of the CLP approach is to overcome the inherent limitations in dealing efficiently with elements of domains different from Herbrand terms. Satisfiability of constraints of particular domains may be checked in an efficient way, apart from the logic. In [6] an interesting and useful constraint system that combines real numbers with Herbrand terms is presented as an instance of our scheme.

* The authors are partially supported by the Spanish project TIC2002-01167 'MELODIAS'.

In addition, in [10] a goal solving procedure for the scheme $HH(\mathcal{C})$ was presented and it is proved to be sound and complete w.r.t. the intuitionistic deduction system $\mathcal{UC}$, previously defined. This goal solving procedure could be considered an operational semantics of $HH(\mathcal{C})$.

Of course, an operational interpretation is needed in order to specify programs which can be executed with certain efficiency. But a clear declarative semantics would simplify the programmer's work. If the deduction system is supported by model-theoretic semantics involving more abstract elements, then additional properties of programs can be analyzed in a formal way. The attempts to provide declarative semantics for LP languages based on mathematical foundations are extensive and fruitful (see i.e. [11,2,3]). This is also the case of CLP [8,5]. In both, LP and CLP, most of the studies are based on fixed point theories, in which it is easy to define program analysis frameworks.

The aim of the present work is to define a fixed point semantics for $HH(\mathcal{C})$. That definition is inspired in the semantics for a fragment of HH described in [12]. Our purpose is to find a model such that for any program Δ, finite set of constraints Γ and goal G, G can be proved in the deduction system $\mathcal{UC}$, if and only if, G is satisfied in that model in the context $\langle \Delta, \Gamma \rangle$. However, in order to build such model, it is important to realize that, during the search of a proof for a goal from a program Δ and a set of constraints Γ, both Δ and Γ may grow. So we will identify the notion of interpretation with functions that associate to every pair $\langle \Delta, \Gamma \rangle$ a set of "true" atoms, in such a way that, if Δ or Γ are augmented, the set of true atoms cannot decrease. The model we are looking for will be the least fixed point of a continuous operator that transforms such interpretations.

The rest of this paper is organized as follows: Section 2 gathers the syntax of constraint systems, as well as the syntax of programs and goals of $HH(\mathcal{C})$, and shows some examples of its use as logic programming language. In Section 3 we recapitulate the definition of the proof system $\mathcal{UC}$ in $HH(\mathcal{C})$, which permits only uniform proofs of goals from programs and constraints. Section 4 contains the main new results of the paper. A fixed point semantics for $HH(\mathcal{C})$ is presented, and soundness and completeness results are obtained. The proofs are compressed, but they can be found extended in the Appendix. Finally a new version of this semantics for an interesting class of constraint systems is summarized. In Section 5 related works and future research lines are commented.

2 The programming language $HH(\mathcal{C})$

The purpose of the present section is to briefly describe the syntax of $HH(\mathcal{C})$, introduced in [10]. $HH(\mathcal{C})$ can be regarded as a constraint logic programming language, not founded in Horn logic, as usual, but in the extended logic of hereditary Harrop formulas. As most CLP languages, it is in fact a parameterized scheme that can be instantiated by particular constraint systems. The requirements imposed to such generic constraint systems are gathered below.

Given a signature Σ, containing constants, function and predicate symbols, including the equality predicate $\approx$, a constraint system $\mathcal{C}$ over Σ is a

pair $(\mathcal{L}_\mathcal{C}, \vdash_\mathcal{C})$, where $\mathcal{L}_\mathcal{C}$ is the set of formulas that play the role of constraints, and $\vdash_\mathcal{C} \subseteq \mathcal{P}(\mathcal{L}_\mathcal{C}) \times \mathcal{L}_\mathcal{C}$ [1] is the entailment or deduction relation between sets of constraints and constraints. $\mathcal{C}$ must fulfill the following conditions:

- $\mathcal{L}_\mathcal{C}$ is a set of first-order formulas built up using the signature Σ, which must specifically include $\top$ (true), $\bot$ (false), and the equations $t \approx t'$ for any Σ-terms t and t'.
- $\mathcal{L}_\mathcal{C}$ is closed under $\wedge, \Rightarrow, \exists, \forall$ and the application of substitutions of terms for variables.
- $\vdash_\mathcal{C}$ is *compact*, i.e., $\Gamma \vdash_\mathcal{C} C$ iff $\Gamma_0 \vdash_\mathcal{C} C$ for some finite $\Gamma_0 \subseteq \Gamma$. $\vdash_\mathcal{C}$ is also *generic*, i.e., $\Gamma \vdash_\mathcal{C} C$ implies $\Gamma\sigma \vdash_\mathcal{C} C\sigma$ for any substitution σ. $\Gamma\sigma$ is the result of applying the substitution σ to each formula in Γ, avoiding the capture of free variables.
- All the inference rules related to $\wedge, \Rightarrow, \exists, \forall$ and $\approx$ valid in the intuitionistic fragment of first-order logic are also valid to infer entailments in the sense of $\vdash_\mathcal{C}$.

Hereafter, we will consider a fixed signature Σ and a constraint system $\mathcal{C}$ over Σ. C will stand for $\mathcal{C}$-constraints and Γ for finite sets of $\mathcal{C}$-constraints. $\bigwedge \Gamma$ stands for the conjunction of constraints of Γ.

Let the *set of program predicate symbols* Π_P be a set of predicate symbols such that $\Sigma \cap \Pi_P = \emptyset$. In the rest of the paper Σ and Π_P are assumed fixed. Let At be the set of atomic formulas built up with the predicate symbols in Π_P and Σ-terms. The set $\mathcal{G}$ of *goals* G, and the set $\mathcal{D}$ of *clauses* D over Σ and Π_P are defined by the mutually-recursive rules below. Notice that constraints can be found embedded in goals and clauses.

$G ::= A \mid C \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \Rightarrow G \mid C \Rightarrow G \mid \exists x G \mid \forall x G,$

$D ::= A \mid G \Rightarrow A \mid D_1 \wedge D_2 \mid \forall x D,$

where $A \in At$.

A program over Σ and Π_P is a finite subset of $\mathcal{D}$. The symbol Δ will be used for programs. Let $\mathcal{W}$ be the set of programs over Σ and Π_P.

The following definition will be useful in order to simplify the usage of program clauses.

Definition 1. Given a clause D, the set of its elaborations, $elab(D)$, is the set of clauses defined by the following rules:

- $elab(A) \stackrel{\text{def}}{=} \{\top \Rightarrow A\}$.
- $elab(D_1 \wedge D_2) \stackrel{\text{def}}{=} elab(D_1) \cup elab(D_2)$.
- $elab(G \Rightarrow A) \stackrel{\text{def}}{=} \{G \Rightarrow A\}$.
- $elab(\forall x D) \stackrel{\text{def}}{=} \{\forall x D' \mid D' \in elab(D)\}$.

This definition is naturally extended to sets $S \subseteq \mathcal{D}$: $elab(S) \stackrel{\text{def}}{=} \bigcup_{D \in S} elab(D)$. The clauses of $elab(S)$ for any S are said to be *elaborated*. Notice that elaborated clauses have always the form $\forall \overline{x}(G \Rightarrow A)$ [2]. A *variant* of $\forall \overline{x}(G \Rightarrow A)$ is a clause $\forall \overline{y}((G \Rightarrow A)[\overline{y}/\overline{x}])$, where no $y \in \overline{y}$ occurs in $G \Rightarrow A$. $F[\overline{y}/\overline{x}]$ is the result of applying to F the substitution that replaces x_i by y_i for each $x_i \in \overline{x}$.

[1] Here and in the rest of the paper, given a set S, $\mathcal{P}(S)$ denotes its powerset.

[2] $\forall \overline{x}$ is an abbreviation for $\forall x_1 \ldots \forall x_n$, and analogously for $\exists \overline{x}$.

One of the outstanding features of the logic programming language $HH(\mathcal{C})$ is its high expressive power. In order to illustrate it, a couple of examples is presented below, in a Prolog-like notation, enriched with the logic symbols $\forall$ and $\Rightarrow$.

Example 1. Let us consider the constraint system $\mathcal{R}$, consisting of the field of real numbers with the usual arithmetic, and predicate symbols $=, <, >, \leq$ and $\geq$. Taking $\Pi_P = \{\texttt{triangle}, \texttt{isosceles}\}$, let us consider the singleton program Δ_1 with the clause:

```
triangle(A, B, C):- A > 0, B > 0, C > 0,
                    A < C + B, B < A + C, C < A + B.
```

The variables `A`, `B` and `C` are intended to be lengths, so that the predicate `triangle(A,B,C)` becomes true when it is possible to build a triangle with sides `A`, `B` and `C`. Let Δ_2 be the program:

```
isosceles(A, B, C):- triangle(A, B, C), A = B.
isosceles(A, B, C):- triangle(A, B, C), A = C.
isosceles(A, B, C):- triangle(A, B, C), B = C.
```

Suppose we want to know which conditions over Y guarantee that, for any $X > 1$, it is possible to build an isosceles triangle with sides $\langle X, X, Y \rangle$. The goal which captures that query is:

$$G \equiv (\Delta_2 \Rightarrow \forall \texttt{X (X>1} \Rightarrow \texttt{isosceles(X, X, Y)))}.$$

In G, similarly as in [12], the program Δ_2 is being used as a *module* that is loaded over Δ_1 when solving G. Notice that such goal cannot be written in *CLP* languages based on Horn clauses, because the connectives $\Rightarrow$ and $\forall$ would not be allowed in goals. Given the program Δ_1 and the goal G, according to the proof theory that will be described in Section 3, `0<Y`$\wedge$`Y<=2` is a correct answer constraint for G from Δ_1.

Example 2. This example shows an efficient and reversible program to compute Fibonacci numbers. It is borrowed from [10]. The constraint system used is $\mathcal{R}$ again.

```
fib(N,X):- memfib(0, 1) ⇒
     (memfib(1, 1) ⇒ getfib(N, X, 1)).
getfib(N, X, M):- 0 <= N, N <= M, memfib(N, X).
getfib(N, X, M):- N > M, memfib(M-1, F1), memfib(M, F2),
     (memfib(M + 1, F1 + F2) ⇒ getfib(N, X, M + 1)).
```

The goal `getfib(N,X,M)` computes the `N`-th Fibonacci number in `X`, assuming that the Fibonacci numbers fib_i, with $0 \leq i \leq$ `M`, are stored in the local program as atoms for `memfib`. During the computation, atoms `memfib` for fib_i, with `M`$< i \leq$`N`, are locally memorized.

Other examples can be found in [10,9,6]. The ones in [9] belong to the higher-order version of $HH(\mathcal{C})$, and those in [6] to the instance $HH(\mathcal{RH})$.

3 The calculus $\mathcal{UC}$

We follow the ideas of Miller et al. [13], in which logic programming languages are identified with those such that non-uniform proofs of goals in a deduction system can be discarded. Those languages are called abstract logic programming languages. This characterization captures the fact that in *LP* the search of a proof for a goal is directed by the structure of such goal. For the classical and intuitionistic logics there are known deduction systems, based on sequent calculus, such that certain fragments of those logics have been proved to be abstract logic programming languages, w.r.t. them. For the case of $HH(\mathcal{C})$, the existence of constraints implies the necessity of a different calculus. In [10] a sequent calculus that combines intuitionistic rules for the logic connectives with the entailment relation $\vdash_{\mathcal{C}}$ is presented, and it is proved to be equivalent to another calculus, designated by $\mathcal{UC}$, such that every $\mathcal{UC}$-proof is uniform, therefore demonstrating that $HH(\mathcal{C})$ is also an abstract logic programming language.

The calculus $\mathcal{UC}$ is now briefly described. $\mathcal{UC}$ consists of the set of deduction rules below. For any program Δ, finite set of constraints Γ, and goal G, the notation $\Delta; \Gamma \vdash_{\mathcal{UC}} G$ stands for the assertion that there is a proof for the sequent $\Delta; \Gamma \vdash G$ using, in a bottom-up fashion, the rules of the calculus $\mathcal{UC}$. So $\mathcal{UC}$-proofs will be regarded as trees.

$\mathcal{UC}$-Rules

Rules for constraints and atomic goals:

$$\frac{\Gamma \vdash_{\mathcal{C}} C}{\Delta; \Gamma \vdash C}\ (C_R) \qquad \frac{\Delta; \Gamma \vdash \exists \overline{x}(A \approx A' \wedge G)}{\Delta; \Gamma \vdash A}\ (Clause)$$

where $\forall \overline{x}(G \Rightarrow A')$ is a variant of some clause in $elab(\Delta)$; the variables of $\overline{x}$ do not occur free in the lower sequent; $A \equiv P(t_1, \ldots, t_n)$, $A' \equiv P(s_1, \ldots, s_n)$, and $A \approx A'$ denotes the conjunction $t_1 \approx s_1 \wedge \ldots \wedge t_n \approx s_n$.

Rules introducing connectives:

$$\frac{\Delta; \Gamma \vdash G_1 \quad \Delta; \Gamma \vdash G_2}{\Delta; \Gamma \vdash G_1 \wedge G_2}\ (\wedge_R) \qquad \frac{\Delta; \Gamma \vdash G_i}{\Delta; \Gamma \vdash G_1 \vee G_2}\ (\vee_R), i \in \{1, 2\}$$

$$\frac{\Delta, D; \Gamma \vdash G}{\Delta; \Gamma \vdash D \Rightarrow G}\ (\Rightarrow_R) \qquad \frac{\Delta; \Gamma, C \vdash G}{\Delta; \Gamma \vdash C \Rightarrow G}\ (\Rightarrow_{C_R})$$

$$\frac{\Delta; \Gamma, C \vdash G[y/x] \quad \Gamma \vdash_{\mathcal{C}} \exists y C}{\Delta; \Gamma \vdash \exists x G}\ (\exists_R) \qquad \frac{\Delta; \Gamma \vdash G[y/x]}{\Delta; \Gamma \vdash \forall x G}\ (\forall_R)$$

In rules $(\exists_R)$ and $(\forall_R)$ the variable y does not occur free in any formula of the lower sequent.

When $\Delta; C \vdash_{\mathcal{UC}} G$ holds, if C is satisfiable, it is said to be a *correct answer constraint* for G from Δ.

In [10] a goal solving procedure for $HH(\mathcal{C})$ is introduced and proved to be sound and complete w.r.t the deducibility $\vdash_{\mathcal{UC}}$.

4 Fixed Point Semantics

The goal solving procedure defined in [10] may be regarded as an operational semantics for $HH(\mathcal{C})$. However, from the theoretical point of view, the programming language $HH(\mathcal{C})$ presented lacks a declarative semantics. The only meanings that we may associate to programs, so far, are sets of proofs. In addition, having in mind that $\mathcal{UC}$ is not a traditional sequent calculus due to the presence of constraints, its correspondence with any of the known logical inference relations ($\models$) cannot be direct, and the definition of a specific model-theoretic semantics merging the intuitionistic behavior of HH and the interpretation of constraints is a hard task.

In this section, alternative semantics based on a fixed point construction, widely utilized in LP and CLP, are introduced. For the traditional LP language, given a program P there is a continuous operator T_P transforming models (sets of atoms) such that a goal G can be proved from P, if and only if, G "is true" in the least fixed point of T_P [17]. As analyzed in [12], for the fragment of HH that includes implications in goals, the situation is more complex, since while building a proof for a goal G the program Δ may be augmented. Therefore programs play the role of contexts, and interpretations become monotonous functions mapping each program into a set of atoms. Instead of a family $\{T_\Delta\}_{\Delta\in\mathcal{W}}$ of continuous operators, there is a unique operator T, and the main result is that G can be proved from Δ, if and only if, G "is true" in the least fixed point of T at the context Δ. In the present paper we extend this approach for the language $HH(\mathcal{C})$. New difficulties arise since the universal quantifier, as well as constraints, are allowed in goals, and then embedded into programs. When proving a goal G from a program Δ there is also the presence of a set of constraints Γ; both Δ and Γ may result augmented, therefore the notion of context is extended to pairs $\langle\Delta,\Gamma\rangle$. So an interpretation of Δ and Γ should depend on interpretations of $\langle\Delta',\Gamma'\rangle$ with $\Delta'\subseteq\Delta$, $\Gamma'\subseteq\Gamma$. In this reason, interpretations are defined as monotonous functions able to interpret every pair $\langle\Delta,\Gamma\rangle$. A continuous operator transforming such interpretations is defined. We prove that for any Δ,Γ and G, $\Delta;\Gamma\vdash_{\mathcal{UC}} G$ if and only if G is satisfied by the least fixed point of this operator at the context $\langle\Delta,\Gamma\rangle$.

4.1 Interpretations

Let us assume that Σ, Π_P, a constraint system $\mathcal{C}$ over Σ and a set Π_P of program predicate symbols have been chosen.

Definition 2. An *interpretation* I is a function $I:\mathcal{W}\times\mathcal{P}(\mathcal{L}_\mathcal{C})\to\mathcal{P}(At)$ that is monotonous, i. e. for any Δ_1,Δ_2 and Γ_1,Γ_2 such that $\Delta_1\times\Gamma_1\subseteq\Delta_2\times\Gamma_2$, $I(\Delta_1,\Gamma_1)\subseteq I(\Delta_2,\Gamma_2)$ holds. Let $\mathcal{I}$ denote the set of interpretations.

So, interpretations are notions of truth. $I(\Delta,\Gamma)$ is the set of atoms that are true for I in the context $\langle\Delta,\Gamma\rangle$. The property that, when $\langle\Delta,\Gamma\rangle$ becomes greater, the set of true atoms cannot decrease, is guaranteed by the monotonicity of interpretations.

Definition 3. For any $I_1, I_2 \in \mathcal{I}$, $I_1 \sqsubseteq I_2$ if for each Δ and Γ, $I_1(\Delta, \Gamma) \subseteq I_2(\Delta, \Gamma)$ holds.

It is straightforward to check that $(\mathcal{I}, \sqsubseteq)$ is a poset, i. e. $\sqsubseteq$ is a partial order.

However, in order to define fixed point semantics, the set of interpretations needs to be a complete lattice, not just a poset.

Lemma 1. *The poset $(\mathcal{I}, \sqsubseteq)$ is a complete lattice. Furthermore, given $S \subseteq \mathcal{I}$, its least upper bound and greatest lower bound, denoted by $\bigsqcup S$ and $\bigsqcap S$, are characterized by the following equations:*

$$(\bigsqcup S)(\Delta, \Gamma) = \bigcup_{I \in S} I(\Delta, \Gamma) \text{ for any } \Delta \text{ and } \Gamma,$$
$$(\bigsqcap S)(\Delta, \Gamma) = \bigcap_{I \in S} I(\Delta, \Gamma) \text{ for any } \Delta \text{ and } \Gamma.$$

Proof. The claim follows from the fact that $\mathcal{I}$ is a set of monotonic functions whose range, $\mathcal{P}(At)$, is a complete lattice. □

As a particular case, $(\mathcal{I}, \sqsubseteq)$ has an infimum $\bigsqcap \mathcal{I}$, denoted by $I_\perp$, the constant function $\emptyset$.

The following definition formalizes the notion of a goal G being "true" for an interpretation I in a context $\langle \Delta, \Gamma \rangle$.

Definition 4. Given $I \in \mathcal{I}$, Δ and Γ, a goal G *is forced* by I, Δ and Γ if $I, \Delta, \Gamma \Vdash G$, where $\Vdash$ is the relation recursively defined depending on the structure of G, as follows:

- $I, \Delta, \Gamma \Vdash C \overset{\text{def}}{\Longleftrightarrow} \Gamma \vdash_{\mathcal{C}} C$.
- $I, \Delta, \Gamma \Vdash A \overset{\text{def}}{\Longleftrightarrow} A \in I(\Delta, \Gamma)$.
- $I, \Delta, \Gamma \Vdash G_1 \wedge G_2 \overset{\text{def}}{\Longleftrightarrow} I, \Delta, \Gamma \Vdash G_i$ for each $i \in \{1, 2\}$.
- $I, \Delta, \Gamma \Vdash G_1 \vee G_2 \overset{\text{def}}{\Longleftrightarrow} I, \Delta, \Gamma \Vdash G_i$ for some $i \in \{1, 2\}$.
- $I, \Delta, \Gamma \Vdash D \Rightarrow G \overset{\text{def}}{\Longleftrightarrow} I, \Delta \cup \{D\}, \Gamma \Vdash G$.
- $I, \Delta, \Gamma \Vdash C \Rightarrow G \overset{\text{def}}{\Longleftrightarrow} I, \Delta, \Gamma \cup \{C\} \Vdash G$.
- $I, \Delta, \Gamma \Vdash \exists x G \overset{\text{def}}{\Longleftrightarrow}$ there is a constraint C and a variable y such that:
  - y does not occur free in Δ, Γ, $\exists x G$.
  - $\Gamma \vdash_{\mathcal{C}} \exists y C$.
  - $I, \Delta, \Gamma \cup \{C\} \Vdash G[y/x]$.
- $I, \Delta, \Gamma \Vdash \forall x G \overset{\text{def}}{\Longleftrightarrow}$ there is a variable y such that:
  - y does not occur free in Δ, Γ, $\forall x G$.
  - $I, \Delta, \Gamma \Vdash G[y/x]$.

Since the deduction system $\mathcal{UC}$ has a constraint-oriented formulation, when a proof of an existential quantified goal $\exists x G$ must be found, instead of guessing a witness of x, by means of a substitution $[t/x]$ or by the counterpart constraint $x \approx t$, some extra generality is necessary. Any satisfiable constraint C may be considered,that represents a property characterizing x, v.g. $x^2 \approx 2$. The semantics of a program provides for information regarding the goals which can be proved from it. So, the definition of the forcing relation for the case $\exists x G$ should exhibit the same generality of the rule $(\exists_R)$.

Defined this way, the relation $\Vdash$ has several properties that will help us to prove other more significant results.

Lemma 2. *If $I_1, I_2 \in \mathcal{I}$ and $I_1 \sqsubseteq I_2$, then for any G, Δ, and Γ, $I_1, \Delta, \Gamma \models G$ implies $I_2, \Delta, \Gamma \models G$.*

Proof. The proof is inductive on the structure of G. Only a few cases are considered here, the rest can be found in the Appendix.

- $G \equiv A$, atomic goal. $I_1, \Delta, \Gamma \models A \iff A \in I_1(\Delta, \Gamma)$. $I_1 \sqsubseteq I_2$ implies that $I_1(\Delta, \Gamma) \subseteq I_2(\Delta, \Gamma)$, so $A \in I_2(\Delta, \Gamma)$ and therefore $I_2, \Delta, \Gamma \models A$.
- $G \equiv \forall x G'$. $I_1, \Delta, \Gamma \models \forall x G' \iff$ there is a variable y such that: y does not occur free in Δ, Γ, $\forall x G'$ and $I_1, \Delta, \Gamma \models G'[y/x]$. By induction hypothesis, it holds that $I_2, \Delta, \Gamma \models G'[y/x]$, so $I_2, \Delta, \Gamma \models \forall x G'$. □

The following lemma states a usual property of this kind of semantic approach.

Lemma 3. *Let $\{I_i\}_{i \geq 0}$ be a denumerable family of interpretations such that $I_0 \sqsubseteq I_1 \sqsubseteq I_2 \sqsubseteq \ldots$, and let G be a goal. Then, for any Δ and Γ,*
$$\bigsqcup_{i \geq 0} I_i, \Delta, \Gamma \models G \Rightarrow \exists k \geq 0 \text{ such that } I_k, \Delta, \Gamma \models G.$$

Proof. We already know that $\bigsqcup_{i \geq 0} I_i(\Delta, \Gamma) = \bigcup_{i \geq 0} I_i(\Delta, \Gamma)$. The proof is inductive on the structure of G. Here we deal with a few cases, the rest can be found in the Appendix.

- $G \equiv C \in \mathcal{L}_{\mathcal{C}}$. $\bigsqcup_{i \geq 0} I_i, \Delta, \Gamma \models C \iff \Gamma \vdash_{\mathcal{C}} C \iff I_k, \Delta, \Gamma \models C$, independently of k.
- $G \equiv C \Rightarrow G'$. $\bigsqcup_{i \geq 0} I_i, \Delta, \Gamma \models C \Rightarrow G' \iff \bigsqcup_{i \geq 0} I_i, \Delta, \Gamma \cup \{C\} \models G'$. By induction hypothesis, there is $k \geq 0$ such that $I_k, \Delta, \Gamma \cup \{C\} \models G'$. Therefore, $I_k, \Delta, \Gamma \models C \Rightarrow G'$. □

As it has been mentioned before, the semantic approach we are formulating is based in searching for a model such that $\Delta; \Gamma \vdash_{\mathcal{UC}} G$ iff G is true in such model in the context $\langle \Delta, \Gamma \rangle$. We have shown that each interpretation provides for a version of truth of goals in such contexts. The next step is to define an operator over interpretations whose least fixed point supplies the desired version of truth.

Definition 5. The *operator* $T : \mathcal{I} \longrightarrow \mathcal{I}$ transforms interpretations as follows. For any $I \in \mathcal{I}$, Δ, Γ and $A \in At$, $A \in T(I)(\Delta, \Gamma)$ if there is a variant $\forall \overline{x}(G \Rightarrow A')$ of a clause in $elab(\Delta)$ such that the variables $\overline{x}$ do not occur free in Δ, Γ, A, and $I, \Delta, \Gamma \models \exists \overline{x}(A \approx A' \wedge G)$.

Lemma 4 (Monotonicity of T). *Let $I_1, I_2 \in \mathcal{I}$ such that $I_1 \sqsubseteq I_2$. Then, $T(I_1) \sqsubseteq T(I_2)$.*

Lemma 5 (Continuity of T). *Let $\{I_i\}_{i \geq 0}$ be a denumerable family of interpretations such that $I_0 \sqsubseteq I_1 \sqsubseteq I_2 \sqsubseteq \ldots$. Then $T(\bigsqcup_{i \geq 0} I_i) = \bigsqcup_{i \geq 0} T(I_i)$.*

Proof. Let us deal with both inclusions.

$\supseteq$) This inclusion is always a consequence of the monotonicity of T.

$\subseteq$) Let Δ, Γ and $A \in T(\bigsqcup_{i\geq 0} I_i)(\Delta, \Gamma)$. Due to the definition of T, there is a variant $\forall \overline{x}(G \Rightarrow A')$ of a clause of $elab(\Delta)$ such that the variables $\overline{x}$ do not occur free in Δ, Γ, A, and $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash \exists \overline{x}(A \approx A' \wedge G)$. Thanks to Lemma 3, there exists $k \geq 0$ such that $I_k, \Delta, \Gamma \Vdash \exists \overline{x}(A \approx A' \wedge G)$, and therefore $A \in T(I_k)(\Delta, \Gamma)$. So, we have proved that $T(\bigsqcup_{i\geq 0} I_i)(\Delta, \Gamma) \subseteq \bigcup_{k\geq 0} T(I_k)(\Delta, \Gamma) = (\bigsqcup_{k\geq 0} T(I_k))(\Delta, \Gamma)$, for any Δ and Γ, thus $T(\bigsqcup_{i\geq 0} I_i) \sqsubseteq \bigsqcup_{k\geq 0} T(I_k)$. $\square$

Theorem 1. *The operator T has a least fixed point, which is $\bigsqcup_{i\geq 0} T^i(I_\perp)$.*

Proof. The claim is an immediate consequence of Lemmas 4 and 5, and the Knaster-Tarski fixed point theorem. $\square$

From now on, $lfp(T)$ denotes the least fixed point of T.

Example 3. Let Δ_3 be the program in Example 2. Figure 1 shows some of the goals that are forced by the first interpretations $T^i(I_\perp)$ in the contexts $\langle \Delta, \Gamma \rangle$, where $\Gamma = \{z_1 = 1, z_2 = 1, x = z_1 + z_2\}$, $\Delta_3' = \Delta_3 \cup \{mf(0,1), mf(1,1)\}$ and $\Delta_3'' = \Delta_3 \cup \{mf(0,1), mf(1,1), mf(2, z_1 + z_2)\}$.

$\langle \Delta, \Gamma \rangle$	$T(I_\perp)$	$T^2(I_\perp)$	$T^3(I_\perp)$	$T^4(I_\perp)$	$T^5(I_\perp)$
$\langle \Delta_3, \Gamma \rangle$	...	...	...	$mf(0,1) \Rightarrow$	$fib(2,x)$
	...	...	...	$(mf(1,1) \Rightarrow gf(2,x,1))$	...
$\langle \Delta_3', \Gamma \rangle$	$mf(0,z_1)$,	...	$mf(2, z_1 + z_2)$	$gf(2,x,1)$	...
	$mf(1,z_2)$	...	$\Rightarrow gf(2,x,2)$	...	...
$\langle \Delta_3'', \Gamma \rangle$	$mf(2,x)$	$gf(2,x,2)$	...	...	...

Fig. 1. Steps leading to $T^5(I_\perp), \Delta_3, \Gamma \Vdash fib(2,x)$.

The chart contains the main steps leading to $T^5(I_\perp), \Delta_3, \Gamma \Vdash fib(2,x)$. *memfib* is abbreviated with *mf*, and *getfib* with *gf*.

4.2 Soundness and Completeness

The following theorem states the soundness and completeness we were looking for, establishing the full connection between the fixed point semantics presented and the calculus $\mathcal{UC}$. The definitions below will be used in its proof.

Let $\mathcal{S} = \{\langle \Delta, \Gamma, G \rangle \in \mathcal{W} \times \mathcal{P}(\mathcal{L}_\mathcal{C}) \times \mathcal{G} \mid lfp(T), \Delta, \Gamma \Vdash G\}$. We define the function $ord : \mathcal{S} \longrightarrow \mathbb{N}$ as follows. Given any $\langle \Delta, \Gamma, G \rangle \in \mathcal{S}$, Lemma 3 guarantees that the set of natural numbers k such that $T^k(I_\perp), \Delta, \Gamma \Vdash G$ is nonempty. Therefore, it is possible to define $ord(\langle \Delta, \Gamma, G \rangle)$ as the least element of such set. Let us consider the partial order $(\mathcal{S}, <)$ defined as follows. Given any $\langle \Delta_1, \Gamma_1, G_1 \rangle, \langle \Delta_2, \Gamma_2, G_2 \rangle \in \mathcal{S}$, $\langle \Delta_1, \Gamma_1, G_1 \rangle < \langle \Delta_2, \Gamma_2, G_2 \rangle$ if

- $ord(\langle \Delta_1, \Gamma_1, G_1 \rangle) < ord(\langle \Delta_2, \Gamma_2, G_2 \rangle)$, or
- $ord(\langle \Delta_1, \Gamma_1, G_1 \rangle) = ord(\langle \Delta_2, \Gamma_2, G_2 \rangle)$ and G_1 is a strict subformula of a goal G_2', where G_2' is obtained by renaming the free variables in G_2.

Such partial order is well-founded, because $(\mathbb{N}, <)$ is also well-founded and formulas are finite sequences of symbols.

Theorem 2. *For any Δ, Γ and G goal, $lfp(T), \Delta, \Gamma \models G \iff \Delta; \Gamma \vdash_{\mathcal{UC}} G$.*

Proof. The whole proof can be found in the Appendix. Only the cases when G is atomic or an existential quantification are considered below.

$\Leftarrow$) Let h be the height of a $\mathcal{UC}$-proof for $\Delta; \Gamma \vdash_{\mathcal{UC}} G$. The claim is proved inductively on h. For the base case $h = 1$ and $G \equiv C$ (see the Appendix). For the inductive case, we suppose that $\Delta; \Gamma \vdash G$ has a proof of height h. Let us prove that $lfp(T), \Delta, \Gamma \models G$ by case analysis on the $\mathcal{UC}$-rule employed in the bottom of such proof.

- $(Clause)$. It must be the case that there exist a variant $\forall \overline{x}(G \Rightarrow A')$ of a clause of $elab(\Delta)$, such that the variables in $\overline{x}$ do not occur free in Δ, Γ, A, and that the sequent $\Delta; \Gamma \vdash \exists \overline{x}(A \approx A' \wedge G)$ has a proof of height $h - 1$. By induction hypothesis, $lfp(T), \Delta, \Gamma \models \exists \overline{x}(A \approx A' \wedge G)$. Using the definition of the operator T, the latter implies $A \in T(lfp(T))(\Delta, \Gamma)$, which is equivalent to $T(lfp(T)), \Delta, \Gamma \models A$. But since $T(lfp(T)) = lfp(T)$, the proof is complete.
- $(\exists_R)$. G must be of the form $\exists x G'$, and there must be a constraint C and a variable y not occurring free in Δ, Γ, $\exists x G'$, such that $\Delta; \Gamma, C \vdash G'[y/x]$ has a proof of height $h-1$ and $\Gamma \vdash_{\mathcal{C}} \exists y C$. By induction hypothesis, $lfp(T), \Delta, \Gamma \cup \{C\} \models G'[y/x]$, and therefore $lfp(T), \Delta, \Gamma \models \exists x G'$.

$\Rightarrow$) By induction on the order $(\mathcal{S}, <)$. Let us take $\langle \Delta, \Gamma, G \rangle \in \mathcal{S}$ and assume that, for any other $\langle \Delta', \Gamma', G' \rangle \in \mathcal{S}$, $\langle \Delta', \Gamma', G' \rangle < \langle \Delta, \Gamma, G \rangle$ implies $\Delta'; \Gamma' \vdash_{\mathcal{UC}} G'$. Then, let us conclude $\Delta; \Gamma \vdash_{\mathcal{UC}} G$ by case analysis on the structure of G.

- $G \equiv A$. $\langle \Delta, \Gamma, A \rangle \in \mathcal{S}$ implies that $lfp(T), \Delta, \Gamma \models A$.
  Let $k = ord(\langle \Delta, \Gamma, A \rangle)$, so $T^k(I_\perp), \Delta, \Gamma \models A$, which is equivalent to $A \in (T^k(I_\perp))(\Delta, \Gamma)$. This implies that there is a variant $\forall \overline{x}(G \Rightarrow A')$ of a clause of $elab(\Delta)$ such that the variables $\overline{x}$ do not occur free in Δ, Γ, A, and $T^{k-1}(I_\perp), \Delta, \Gamma \models \exists \overline{x}(A \approx A' \wedge G)$. So $\langle \Delta, \Gamma, \exists \overline{x}(A \approx A' \wedge G) \rangle < \langle \Delta, \Gamma, A \rangle$, and the induction hypothesis can be applied, obtaining that $\Delta; \Gamma \vdash_{\mathcal{UC}} \exists \overline{x}(A \approx A' \wedge G)$. Using the rule $(Clause)$ with the elaborated clause $\forall \overline{x}(G \Rightarrow A')$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} A$.
- $G \equiv \exists x G'$. Then $\langle \Delta, \Gamma, G \rangle \in \mathcal{S}$ implies that there is a constraint C and a variable y such that y does not occur free in Δ, Γ, $\exists x G'$, $\Gamma \vdash_{\mathcal{C}} \exists y C'$ and $lfp(T), \Delta, \Gamma \cup \{C\} \models G'[y/x]$. Clearly, $ord(\langle \Delta, \Gamma, G \rangle) = ord(\langle \Delta, \Gamma \cup \{C\}, G'[y/x] \rangle)$ and $G'[y/x]$ is a renaming of a strict subformula of G, so $\langle \Delta, \Gamma \cup \{C\}, G'[y/x] \rangle < \langle \Delta, \Gamma, G \rangle$. Therefore, by the induction hypothesis we obtain $\Delta; \Gamma, C \vdash_{\mathcal{UC}} G'[y/x]$. Thanks to the rule $(\exists_R)$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} G$. □

This fixed point semantics supplies a framework in which properties of programs can be easily analyzed. For instance, two programs can be compared using the interpretation $lfp(T)$. Let us consider that two programs Δ and Δ' are said to be equivalent if, for any Γ and G, $\Delta; \Gamma \vdash_{\mathcal{UC}} G \iff \Delta'; \Gamma \vdash_{\mathcal{UC}} G$. In other words, for every Γ, the same goals can be deduced from them. Then the problem of check the equivalence between Δ and Δ' can be reduced to prove that $lfp(T)(\Delta, \Gamma) = lfp(T)(\Delta', \Gamma)$, for every Γ. This is due to the

previous results, since intuitively $lfp(T)$ provides the atoms that can be proved from a program in the context of a set of constraints.

Example 4. Let $\Delta = \{\forall x(x \geq 0 \Rightarrow p(x)), \forall x(x < 0 \Rightarrow p(x))\}$, and $\Delta' = \{\forall x(x \geq 0 \vee x < 0 \Rightarrow p(x))\}$, two programs for the instance $HH(\mathcal{R})$. Δ and Δ' are not equivalent because $lfp(T)(\Delta, \emptyset) = \emptyset$, but $p(y) \in lfp(T)(\Delta', \emptyset)$. This happens because the entailment relation in the constraint system $\mathcal{R}$ is classical deduction, but, for programs, an intuitionistic interpretation approach is considered.

On the contrary, if Δ and Δ' are such that $\{\forall x(q(x) \Rightarrow p(x)), \forall x(q'(x) \Rightarrow p(x))\} \subseteq \Delta$, and $\{\forall x(q(x) \vee q'(x) \Rightarrow p(x))\} \subseteq \Delta'$, these programs could be equivalent.

4.3 Models

At this stage, $lfp(T)$ has already been proved to be a sound and complete semantics with respect to $\mathcal{UC}$ in a sense. However, instead of having a unique model, it would also be desirable to provide for a more general notion of model such that $\Delta, \Gamma \vdash_{\mathcal{UC}} G$ iff G is true in the context $\langle \Delta, \Gamma \rangle$ for every model. Such notion of model is provided below, together with the expected results.

Definition 6. Given $D \equiv \forall \overline{x}(G \Rightarrow A)$, an interpretation I is a *model* of D, denoted by $I \Vdash D$, if for any Δ, Γ and $A' \in At$ such that D is a variant of a clause in $elab(\Delta)$ and no variable $x \in \overline{x}$ occurs free in Δ, Γ, A', if $I, \Delta, \Gamma \Vdash \exists \overline{x}(G \wedge A \approx A')$ then $A' \in I(\Delta, \Gamma)$.

Intuitively, an interpretation I is model of an elaborated clause D if, whenever D is available, I gathers all the atoms possibly inferred by using the clause D.

Definition 7. An interpretation I is said to be a *model* if $I \Vdash D$ holds for every elaborated clause D.

Lemma 6. *For any interpretation I, $I \in \mathcal{I}$ is a model $\iff$ $T(I) \sqsubseteq I$.*

Proof. $T(I) \sqsubseteq I \iff$ for any Δ and Γ, $T(I)(\Delta, \Gamma) \subseteq I(\Delta, \Gamma) \iff$ for any Δ, Γ, A and any variant $\forall \overline{x}(G \Rightarrow A')$ of a clause in $elab(\Delta)$ such that the variables $\overline{x}$ do not occur free in Δ, Γ, A, if $I, \Delta, \Gamma \Vdash \exists \overline{x}(A \approx A' \wedge G)$ then $A \in I(\Delta, \Gamma)$. However, by Definition 6, this is equivalent to say that $I \Vdash D$ for any elaborated clause D, i. e., I is a model. □

Lemma 7. *For any $I \in \mathcal{I}$, if $T(I) \sqsubseteq I$ then $lfp(T) \sqsubseteq I$.*

Proof. It is well known that, for continuous operators in complete lattices, any postfixed point is greater (or equal to) the least fixed point. □

Theorem 3. *For any Γ, Δ and G,*

$$\Delta; \Gamma \vdash_{\mathcal{UC}} G \iff I, \Delta, \Gamma \Vdash G \text{ holds for every model } I.$$

Proof. $I, \Delta, \Gamma \Vdash G$ for every model $I \iff I, \Delta, \Gamma \Vdash G$ for every I such that $T(I) \sqsubseteq I$, thanks to Lemma 6 $\iff lfp(T), \Delta, \Gamma \Vdash G$, from Lemmas 2 and 7 $\iff \Delta; \Gamma \vdash_{\mathcal{UC}} G$, by virtue of Theorem 2. □

4.4 Considering particular classes of constraint systems

A fixed point semantics has just been presented for $HH(\mathcal{C})$ for any general constraint system $\mathcal{C}$. In it, the constraint system has been used as a black box, through the entailment relation $\vdash_{\mathcal{C}}$, which is a syntactic tool. See, for example, the cases C and $\exists x G$ of Definition 4. The conditions imposed in Section 2 are meant as minimal requirements for a $\mathcal{C}$ to be a constraint system, but in many useful cases $\mathcal{C}$ satisfies additional properties. For example, it is many times the case when $\mathcal{L}_{\mathcal{C}}$ is the whole set of first-order formulas over a signature, and $\Gamma \vdash_{\mathcal{C}} C$ holds iff $Ax_{\mathcal{C}} \cup \Gamma \vdash C$, where $Ax_{\mathcal{C}}$ is a suitable set of first-order axioms and $\vdash$ is the entailment relation of classical first-order logic with equality. There are many well known constraint systems of this form. For instance, $\mathcal{CFT}$, where $Ax_{\mathcal{CFT}}$ is Smolka and Treinen's axiomatization of the domain of *feature trees* [15]; or $\mathcal{R}$, where $Ax_{\mathcal{R}}$ is Tarski's axiomatization of the real numbers [16]. See also the system $\mathcal{RH}$ defined in [6]. In these cases, the syntactic relation $\vdash_{\mathcal{C}}$ has a clear connection with the inference relation $\models$ in classical logic, and the requirements specified by $\vdash_{\mathcal{C}}$ in the preceding semantics can be replaced by conditions over $\models$.

As in the frame of *CLP* we are interested in generalized conditions for the constraint systems that would guarantee the existence of semantics for constraints, based on a model theory, and that could be incorporated to the fixed point semantics of logic programs.

Usually the semantics of constraint logic programs are based on the assumption that the domain of computation (model), which is the structure used to interpret the constraints; the solver, which checks if a constraint is satisfiable; and the constraint theory, that describes the logical semantics of the constraints, *agree*. See [8] for details.

Now we will focus on constraint systems for which an additional condition is required. This condition represents the same idea of agreement mentioned before, and it is specified now. Some notation is introduced for that purpose.

Given a constraint system $\mathcal{C}$ over a signature Σ, the $\mathcal{C}$-constraints will be interpreted by means of a Σ-structure $\mathcal{A}_{\mathcal{C}}$ consisting of a carrier set and an interpretation over it for every symbol of Σ. An assignment for $\mathcal{A}_{\mathcal{C}}$, denoted ν, is a function mapping variables into elements of the carrier of $\mathcal{A}_{\mathcal{C}}$. $[\![\_]\!]_{\nu}^{\mathcal{A}_{\mathcal{C}}}$ is a boolean function that applied to a constraint C produces the classical interpretation of C under $\mathcal{A}_{\mathcal{C}}$ and ν.

$\mathcal{A}_{\mathcal{C}}$ and $\mathcal{C}$ are said to *agree* if for any Γ, C and ν, $\Gamma \vdash_{\mathcal{C}} C$ if and only if $[\![\bigwedge \Gamma \Rightarrow C]\!]_{\nu}^{\mathcal{A}_{\mathcal{C}}} = \mathit{true}$.

We have defined a semantics for this class of constraint systems, based on a notion of forcing similar to that in Definition 4, but in which the sets of constraints are replaced by the assignments making them true in $\mathcal{A}_{\mathcal{C}}$. The introduction of the whole theory leads to several intermediate definitions and technical lemmas that are not developed here. However, the main ideas and results are summarized.

Let us assume that $\mathcal{A}_{\mathcal{C}}$ and $\mathcal{C}$ agree, and that ν henceforth denotes assignments for $\mathcal{A}_{\mathcal{C}}$. Constraints will be interpreted by the sets of such assignments that make them true. Formally, given a constraint C, the set $[\![C]\!]$ is defined as

$\{\nu : dom(\nu) = free(C) \mid [\![C]\!]_\nu = true\}$[3], where $dom(\nu)$ is the set of variables mapped by ν. Such definition is directly extended to finite sets of constraints, i. e. $[\![\Gamma]\!] = [\![\bigwedge \Gamma]\!]$. Notice that $\Gamma \vdash_{\mathcal{C}} C$ iff $[\![\Gamma]\!] \subseteq [\![C]\!]$.

Example 5. Let $\mathcal{C} = \mathcal{R}$ and $\mathcal{A}_{\mathcal{R}}$ be the Σ-structure whose carrier is $\mathbb{R}$ and that interprets constants for real numbers and arithmetic symbols in the natural way. If $C \equiv x * x + y * y = 1$, then $[\![C]\!] = \{\nu : \{x, y\} \to \mathbb{R}^2 \mid (\nu(x))^2 + (\nu(y))^2 = 1\}$. Once each variable is associated to a coordinate axis, this can be assimilated to the set $\{\langle x, y\rangle \in \mathbb{R}^2 \mid x^2+y^2 = 1\}$, the circle of radius 1 centered at the origin of the real plane. Thus, the syntactic object $x * x + y * y = 1$ is replaced by the circle which is its intended meaning in $\mathcal{A}_{\mathcal{R}}$.

As in the case of $\models$, we are looking for a model $\overline{I}$ and a relation $\models^{\mathcal{C}}$ such that for any Δ, Γ and G, $\Delta; \Gamma \vdash_{\mathcal{UC}} G$ iff $\overline{I}, \Delta, [\![\Gamma]\!] \models^{\mathcal{C}} G$. Several technicalities are required in the proof of such result, but the foundations are in some sense similar to those for the preceding semantics. A new notion of interpretation $\overline{I}$ is defined. Interpretations are monotonous functions applied to pairs $\langle \Delta, \nu\rangle$. An operator $\overline{T}$ that transforms such interpretations is defined, whose least fixed point is the model we were looking for. The main result is the following theorem.

Theorem 4. *For any Δ, Γ and G, $lfp(T), \Delta, \Gamma \models G \Longleftrightarrow lfp(\overline{T}), \Delta, [\![\Gamma]\!] \models^{\mathcal{C}} G$. Therefore, $\Delta; \Gamma \vdash_{\mathcal{UC}} G \iff lfp(\overline{T}), \Delta, [\![\Gamma]\!] \models^{\mathcal{C}} G$.*

5 Conclusions

In previous papers [10,9] combinations of *HH* and *CLP* were proposed, producing first and higher order schemes *HH*($\mathcal{C}$) parametric w.r.t. the constraint system. These amalgamated languages gather the expressivity and the efficiency advantages of *HH* and *CLP*, respectively. A proof system that merges inference rules from intuitionistic sequent calculus with the entailment relation of a constraint system was defined. This proof system guarantees uniform proofs, which are the basis of abstract logic programming languages [13]. A goal solving procedure that is sound and complete w.r.t. the proof system was also presented. Such procedure could be seen as an operational semantics of *HH*($\mathcal{C}$), however the absence of a more declarative semantics for this new language was evident. In this paper we have defined semantics for *HH*($\mathcal{C}$) based on fixed point constructions as is usually done in the *LP* and *CLP* fields [11,2,3,8,5].

As far as we know, our work is the first published attempt to give declarative semantics to an amalgamated logic that combines the Hereditary Harrop fragment of intuitionistic first-order logic with a constraint system. Due to the embedding of implications and universal quantifiers inside goals (and so inside programs), finding a fixed point semantics becomes a hard task, further obstructed by the presence of constraints.

In [12] a model theory is presented for an extension of Horn clauses including implications in goals based on a fixed point construction, and it is proved

[3] *free*(O) is the set of free variables in O, where O stands for a formula or set of formulas.

that the operational meaning of implication is sound and complete w.r.t. this semantics. Our approach is close to this framework, but it incorporates the semantics of universal quantifiers in goals and solves the new difficulties due to the presence of constraints.

A semantics for the fragment of λ-prolog —that is based on the higher-order logic *HH* without constraints—, in which classical and intuitionistic theories coincide, is presented in [18]. But this is not the case if implications and universal quantifiers are considered.

Referring to *CLP*, most of the defined semantics use different fixed point constructions. For instance in [8] fixed point semantics constitute a bridge between operational and algebraic semantics. This is also our aim. But notice that in traditional *CLP* the programs are limited to be Horn clauses with constraints. So in the frame of constraint systems which are complete w.r.t. a theory, programs (with embedded constraints) may be interpreted using classical logical inference. However, this is not the case in our language. A classical theory can be considered for the constraint system, but anyway the intuitionism remains, even in the interpretation of pure programs.

We are now researching for more abstract model theories based on indexed categories or uniform algebras [4,1], that could provide a pure model-theoretic semantics, not so directly connected with the operational semantics. Models should provide for meanings of constraints, programs and goals in a homogeneous way, and the expected general result would claim that C is a correct answer constraint for G from Δ, if and only if, every model satisfying Δ and C satisfies G.

Acknowledgements We appreciate the comments and suggestions from James Lipton concerning the presented work.

References

1. G. Amato and J. Lipton. Indexed categories and bottom-up semantics of logic programs. In R. Nieuwenhuis and A. Voronkov, editors, *LPAR'01*, LNCS 2250, pages 438–454. Springer, 2001.
2. A. Bossi, M. Gabrielli, G. Levi, and M. C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.
3. M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Information and Computation*, 69(1–2):23–80, 2001.
4. S. E. Finkelstein, P. Freyd, and J. Lipton. A new framework for declarative programming. *Theoretical Computer Science*, 300(1–3):91–160, 2003.
5. M. Gabbrielli, G. M. Dore, and G. Levi. Observable semantics for constraint logic programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.
6. M. García-Díaz and S. Nieva. Solving mixed quantified constraints over a domain based on real numbers and Herbrand terms. In Z. Hu and M. Rodríguez-Artalejo, editors, *FLOPS'02*, LNCS 2441, pages 103–118. Springer, 2002.
7. J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
8. J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
9. J. Leach and S. Nieva. A higher-order logic programming language with constraints. In H. Kuchen and K. Ueda, editors, *FLOPS'01*, LNCS 2024, pages 108–122. Springer, 2001.

10. J. Leach, S. Nieva, and M. Rodríguez-Artalejo. Constraint logic programming with hereditary Harrop formulas. *Theory and Practice of Logic Programming*, 1(4):409–445, 2001.
11. J. W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 1987.
12. D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, 1989.
13. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
14. G. Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11:111–145, 1993.
15. G. Smolka and R. Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, 1994.
16. A. Tarski. *A decision method for elementary algebra and geometry*. University of California Press, 1951.
17. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.
18. D. A. Wolfram. A semantics for λ-prolog. *Theoretical Computer Science*, 136:277–288, 1994.

Appendix

Lemma 2. *If $I_1, I_2 \in \mathcal{I}$ and $I_1 \sqsubseteq I_2$, then for any goal G, Δ, and Γ, $I_1, \Delta, \Gamma \models G$ implies $I_2, \Delta, \Gamma \models G$.*

Proof. The proof is inductive on the structure of G:

- $G \equiv C \in \mathcal{L}_{\mathcal{C}}$.
  $I_1, \Delta, \Gamma \models C \iff \Gamma \vdash_{\mathcal{C}} C \iff I_2, \Delta, \Gamma \models C$.
- $G \equiv A$, atomic goal.
  $I_1, \Delta, \Gamma \models A \iff A \in I_1(\Delta, \Gamma)$. $I_1 \sqsubseteq I_2$ implies that $I_1(\Delta, \Gamma) \subseteq I_2(\Delta, \Gamma)$, so $A \in I_2(\Delta, \Gamma)$ and therefore $I_2, \Delta, \Gamma \models A$.
- $G \equiv G_1 \wedge G_2$.
  $I_1, \Delta, \Gamma \models G_1 \wedge G_2 \iff I_1, \Delta, \Gamma \models G_i$ for each $i \in \{1, 2\}$. In both cases the induction hypothesis can be used, so $I_2, \Delta, \Gamma \models G_i$ for each $i \in \{1, 2\}$, which implies that $I_2, \Delta, \Gamma \models G_1 \wedge G_2$.
- $G \equiv G_1 \vee G_2$.
  $I_1, \Delta, \Gamma \models G_1 \vee G_2 \iff$ there is $i \in \{1, 2\}$ such that $I_1, \Delta, \Gamma \models G_i$. By induction hypothesis, $I_2, \Delta, \Gamma \models G_i$, which implies that $I_2, \Delta, \Gamma \models G_1 \vee G_2$.
- $G \equiv D \Rightarrow G'$.
  $I_1, \Delta, \Gamma \models D \Rightarrow G' \iff I_1, \Delta \cup \{D\}, \Gamma \models G'$. By induction hypothesis, $I_2, \Delta \cup \{D\}, \Gamma \models G'$ holds, which implies that $I_2, \Delta, \Gamma \models D \Rightarrow G'$.
- $G \equiv C \Rightarrow G'$.
  $I_1, \Delta, \Gamma \models C \Rightarrow G' \iff I_1, \Delta, \Gamma \cup \{C\} \models G'$. By induction hypothesis, $I_2, \Delta, \Gamma \cup \{C\} \models G'$ holds, which implies that $I_2, \Delta, \Gamma \models C \Rightarrow G'$.
- $G \equiv \exists x G'$.
  $I_1, \Delta, \Gamma \models \exists x G' \iff$ there is a $\mathcal{C}$-constraint C and a variable y such that:
  - y does not occur free in Δ, Γ, $\exists x G'$.
  - $\Gamma \vdash_{\mathcal{C}} \exists y C$.
  - $I_1, \Delta, \Gamma \cup \{C\} \models G'[y/x]$.

  By induction hypothesis, for the same C and y it holds that $I_2, \Delta, \Gamma \cup \{C\} \models G'[y/x]$, therefore $I_2, \Delta, \Gamma \models \exists x G'$.
- $G \equiv \forall x G'$.
  $I_1, \Delta, \Gamma \models \forall x G' \iff$ there is a variable y such that: y does not occur free in Δ, Γ, $\forall x G'$ and $I_1, \Delta, \Gamma \models G'[y/x]$. By induction hypothesis, it holds that $I_2, \Delta, \Gamma \models G'[y/x]$, therefore $I_2, \Delta, \Gamma \models \forall x G'$. □

Lemma 3. *Let $\{I_i\}_{i \geq 0}$ be a denumerable family of interpretations such that $I_0 \sqsubseteq I_1 \sqsubseteq I_2 \sqsubseteq \ldots$, and let G be a goal. Then, for any Δ and Γ, $\bigsqcup_{i \geq 0} I_i, \Delta, \Gamma \models G$ implies that there exists $k \geq 0$ such that $I_k, \Delta, \Gamma \models G$.*

Proof. We already know that $(\bigsqcup_{i \geq 0} I_i)(\Delta, \Gamma) = \bigcup_{i \geq 0} I_i(\Delta, \Gamma)$. By induction on the structure of G:

- $G \equiv C \in \mathcal{L}_{\mathcal{C}}$.
  $\bigsqcup_{i \geq 0} I_i, \Delta, \Gamma \models C \iff \Gamma \vdash_{\mathcal{C}} C \iff I_k, \Delta, \Gamma \models C$ is true independently of $k \geq 0$.

- $G \equiv A$, atomic formula.
  $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash A \iff A \in (\bigsqcup_{i\geq 0} I_i)(\Delta, \Gamma) = \bigcup_{i\geq 0} I_i(\Delta, \Gamma)$. Therefore, there exists $k \geq 0$ such that $A \in I_k(\Delta, \Gamma)$, hence, for that k, $I_k, \Delta, \Gamma \Vdash A$.
- $G \equiv G_1 \wedge G_2$.
  $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash G_1 \wedge G_2 \iff \bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash G_j$ for each $j \in \{1, 2\}$. In both cases the induction hypothesis can be used, so there exist $k_1, k_2 \geq 0$ such that $I_{k_j}, \Delta, \Gamma \Vdash G_j$ for each $j \in \{1, 2\}$. Let $k = max(k_1, k_2)$. Then $I_k, \Delta, \Gamma \Vdash G_j$ for each $j \in \{1, 2\}$ in virtue of Lemma 2, and therefore $I_k, \Delta, \Gamma \Vdash G_1 \wedge G_2$.
- $G \equiv G_1 \vee G_2$.
  $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash G_1 \vee G_2 \iff$ there is $j \in \{1, 2\}$ such that $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash G_j$. The induction hypothesis can be used, so there exist $k \geq 0$ such that $I_k, \Delta, \Gamma \Vdash G_j$, and therefore $I_k, \Delta, \Gamma \Vdash G_1 \vee G_2$.
- $G \equiv D \Rightarrow G'$.
  $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash D \Rightarrow G' \iff \bigsqcup_{i\geq 0} I_i, \Delta \cup \{D\}, \Gamma \Vdash G'$. By induction hypothesis, there is $k \geq 0$ such that $I_k, \Delta \cup \{D\}, \Gamma \Vdash G'$. Therefore, $I_k, \Delta, \Gamma \Vdash D \Rightarrow G'$.
- $G \equiv C \Rightarrow G'$.
  $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash C \Rightarrow G' \iff \bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \cup \{C\} \Vdash G'$. By induction hypothesis, there is $k \geq 0$ such that $I_k, \Delta, \Gamma \cup \{C\} \Vdash G'$. Therefore, $I_k, \Delta, \Gamma \Vdash C \Rightarrow G'$.
- $G \equiv \exists x G'$.
  $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash \exists x G' \iff$ there is a $\mathcal{C}$-constraint C and a variable y such that:
  - y does not occur free in Δ, Γ, $\exists x G'$.
  - $\Gamma \vdash_{\mathcal{C}} \exists y C$.
  - $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \cup \{C\} \Vdash G'[y/x]$.

  By induction hypothesis, it holds that there is a $k \geq 0$ such that $I_k, \Delta, \Gamma \cup \{C\} \Vdash G'[y/x]$. Therefore $I_k, \Delta, \Gamma \Vdash \exists x G'$.
- $G \equiv \forall x G'$.
  $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash \forall x G' \iff$ there is a variable y such that:
  - y does not occur free in Δ, Γ, $\forall x G'$.
  - $\bigsqcup_{i\geq 0} I_i, \Delta, \Gamma \Vdash G'[y/x]$.

  By induction hypothesis, it happens that there exists $k \geq 0$ such that $I_k, \Delta, \Gamma \Vdash G'[y/x]$. Therefore $I_k, \Delta, \Gamma \Vdash \forall x G'$. □

Lemma 4 (Monotonicity of T). *Let $I_1, I_2 \in \mathcal{I}$ such that $I_1 \sqsubseteq I_2$. Then, $T(I_1) \sqsubseteq T(I_2)$.*

Proof. Let us consider any Δ, Γ and $A \in T(I_1)(\Delta, \Gamma)$. The latter implies that there is a variant $\forall \overline{x}(G \Rightarrow A')$ of a clause of $elab(\Delta)$, such that the variables $\overline{x}$ do not occur free in Δ, Γ, A, and $I_1, \Delta, \Gamma \Vdash \exists \overline{x}(A \approx A' \wedge G)$. Using Lemma 2 and the fact that $I_1 \sqsubseteq I_2$, we obtain $I_2, \Delta, \Gamma \Vdash \exists \overline{x}(A \approx A' \wedge G)$, which implies $A \in T(I_2)(\Delta, \Gamma)$. Since no particular choice was made for A, Δ, Γ, this argument proves $T(I_1)(\Delta, \Gamma) \subseteq T(I_2)(\Delta, \Gamma)$ for any Δ and Γ, therefore $T(I_1) \sqsubseteq T(I_2)$. □

Theorem 2. *For any Δ, Γ and G, $lfp(T), \Delta, \Gamma \Vdash G \iff \Delta; \Gamma \vdash_{\mathcal{UC}} G$.*

Proof. Both implications are proved by induction.

$\Leftarrow$) Let h be the height of a $\mathcal{UC}$-proof for $\Delta; \Gamma \vdash_{\mathcal{UC}} G$. The claim is proved inductively on h.

- Base case: $h = 1$. The only possibility is that $G \equiv C \in \mathcal{L}_{\mathcal{C}}$. Then $\Delta; \Gamma \vdash_{\mathcal{UC}} C$ implies that $\Gamma \vdash_{\mathcal{C}} C$, and therefore $lfp(T), \Delta, \Gamma \models C$ holds.
- Inductive case. Assuming that $\Delta; \Gamma \mid\!\!- G$ has a proof of height h, let us prove that $lfp(T), \Delta, \Gamma \models G$ by case analysis on the $\mathcal{UC}$-rule employed in the bottom of such proof.

($Clause$) So there is a variant $\forall \overline{x}(G \Rightarrow A')$ of a clause of $elab(\Delta)$ such that the variables in $\overline{x}$ do not occur free in Δ, Γ, A, and that the sequent $\Delta; \Gamma \mid\!\!- \exists \overline{x}(A \approx A' \wedge G)$ has a proof of height $h-1$. By induction hypothesis, $lfp(T), \Delta, \Gamma \models \exists \overline{x}(A \approx A' \wedge G)$. Using the definition of the operator T, the latter implies that $A \in (T(lfp(T)))(\Delta, \Gamma)$, which is equivalent to $T(lfp(T)), \Delta, \Gamma \models A$. But since $T(lfp(T)) = lfp(T)$, the proof is complete.

($\wedge_R$) It must be the case that there are G_1, G_2 such that $G \equiv G_1 \wedge G_2$ and the sequents $\Delta; \Gamma \mid\!\!- G_i$ have a proof of height less than h for each $i \in \{1, 2\}$. By induction hypothesis, $lfp(T), \Delta, \Gamma \models G_i$ holds for $i \in \{1, 2\}$, which implies $lfp(T), \Delta, \Gamma \models G$.

($\vee_R$) Then there are G_1, G_2 such that $G \equiv G_1 \vee G_2$ and the sequent $\Delta; \Gamma \mid\!\!- G_i$ has a proof of height $h-1$ for some $i \in \{1, 2\}$. By induction hypothesis, $lfp(T), \Delta, \Gamma \models G_i$, hence $lfp(T), \Delta, \Gamma \models G$.

($\Rightarrow_R$) In this case, $G \equiv D \Rightarrow G'$ for some D and G', and the sequent $\Delta, D; \Gamma \mid\!\!- G'$ has a proof of height $h-1$. By induction hypothesis, $lfp(T), \Delta \cup \{D\}, \Gamma \models G'$. Therefore $lfp(T), \Delta, \Gamma \models D \Rightarrow G'$.

($\Rightarrow_{C_R}$) It must be the case that there are C and G' such that $G \equiv C \Rightarrow G'$ and the sequent $\Delta; \Gamma, C \mid\!\!- G'$ has a proof of height $h-1$. By induction hypothesis, $lfp(T), \Delta, \Gamma \cup \{C\} \models G'$. So, $lfp(T), \Delta, \Gamma \models C \Rightarrow G'$.

($\exists_R$) G must be of the form $\exists x G'$, and there must exist C and a variable y not occurring free in Δ, Γ, $\exists x G'$, such that $\Delta; \Gamma, C \mid\!\!- G'[y/x]$ has a proof of height $h-1$ and $\Gamma \vdash_{\mathcal{C}} \exists y C$. By induction hypothesis, $lfp(T), \Delta, \Gamma \cup \{C\} \models G'[y/x]$, and therefore $lfp(T), \Delta, \Gamma \models \exists x G'$.

($\forall_R$) G must be of the form $\forall x G'$, and there must exist a variable y not occurring free in Δ, Γ, $\forall x G'$ such that $\Delta; \Gamma \mid\!\!- G'[y/x]$ has a proof of height $h-1$. By induction hypothesis, $lfp(T), \Delta, \Gamma \models G'[y/x]$, and therefore $lfp(T), \Delta, \Gamma \models \forall x G'$.

$\Rightarrow$) By induction on the order $(\mathcal{S}, <)$. Let us take $\langle \Delta, \Gamma, G \rangle \in \mathcal{S}$ and assume that, for any other $\langle \Delta', \Gamma', G' \rangle \in \mathcal{S}$, $\langle \Delta', \Gamma', G' \rangle < \langle \Delta, \Gamma, G \rangle$ implies $\Delta'; \Gamma' \vdash_{\mathcal{UC}} G'$. Then, let us conclude $\Delta; \Gamma \vdash_{\mathcal{UC}} G$ by case analysis on the structure of G.

- $G \equiv C \in \mathcal{L}_{\mathcal{C}}$. Then $\langle \Delta, \Gamma, C \rangle \in \mathcal{S}$ implies that $\Gamma \vdash_{\mathcal{C}} C$, and therefore $\Delta; \Gamma \vdash_{\mathcal{UC}} C$.
- $G \equiv A$. In this case $\langle \Delta, \Gamma, A \rangle \in \mathcal{S}$ implies that $lfp(T), \Delta, \Gamma \models A$. Let $k = ord(\langle \Delta, \Gamma, A \rangle)$, so $T^k(I_\perp), \Delta, \Gamma \models A$, which is equivalent to $A \in T^k(I_\perp)(\Delta, \Gamma)$. This implies that there is $\forall \overline{x}(G \Rightarrow A') \in elab(\Delta)$ such that the variables $\overline{x}$ do not occur free in Δ, Γ, A, and in addition

$T^{k-1}(I_\perp), \Delta, \Gamma \models \exists \overline{x}(A \approx A' \wedge G)$. Due to the way in which the order in $\mathcal{S}$ is defined, $\langle \Delta, \Gamma, \exists \overline{x}(A \approx A' \wedge G)\rangle < \langle \Delta, \Gamma, A\rangle$, so the induction hypothesis can be applied, obtaining that $\Delta; \Gamma \vdash_{\mathcal{UC}} \exists \overline{x}(A \approx A' \wedge G)$. Using the rule $(Clause)$ with the clause $\forall \overline{x}(G \Rightarrow A')$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} A$.

- $G \equiv G_1 \wedge G_2$. Then $\langle \Delta, \Gamma, G\rangle \in \mathcal{S}$ implies $lfp(T), \Delta, \Gamma \models G_1$ and $lfp(T), \Delta, \Gamma \models G_2$. It is obvious that $ord(\langle \Delta, \Gamma, G\rangle) = ord(\langle \Delta, \Gamma, G_1\rangle = ord(\langle \Delta, \Gamma, G_2\rangle)$ and G_1, G_2 are strict subformulas of G, and hence $\langle \Delta, \Gamma, G_1\rangle, \langle \Delta, \Gamma, G_2\rangle < \langle \Delta, \Gamma, G\rangle$. Therefore, by the induction hypothesis we obtain $\Delta; \Gamma \vdash_{\mathcal{UC}} G_1$ and $\Delta; \Gamma \vdash_{\mathcal{UC}} G_2$. Thanks to the rule $(\wedge_R)$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} G$.
- $G \equiv G_1 \vee G_2$. Then $\langle \Delta, \Gamma, G\rangle \in \mathcal{S}$ implies that there is $i \in \{1, 2\}$ such that $lfp(T), \Delta, \Gamma \models G_i$. Clearly, $ord(\langle \Delta, \Gamma, G\rangle) = ord(\langle \Delta, \Gamma, G_i\rangle)$ and G_i is a strict subformula of G, so $\langle \Delta, \Gamma, G_i\rangle < \langle \Delta, \Gamma, G\rangle$. Therefore, by the induction hypothesis we obtain $\Delta; \Gamma \vdash_{\mathcal{UC}} G_i$ for some $i \in \{1, 2\}$. Thanks to the rule $(\vee_{R_i})$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} G$.
- $G \equiv D \Rightarrow G'$. Then $\langle \Delta, \Gamma, G\rangle \in \mathcal{S}$ implies that $lfp(T), \Delta \cup \{D\}, \Gamma \models G'$. Clearly, $ord(\langle \Delta, \Gamma, G\rangle) = ord(\langle \Delta \cup \{D\}, \Gamma, G'\rangle)$ and G' is a strict subformula of G, so $\langle \Delta \cup \{D\}, \Gamma, G'\rangle < \langle \Delta, \Gamma, G\rangle$. Therefore, by the induction hypothesis we obtain $\Delta, D; \Gamma \vdash_{\mathcal{UC}} G'$. Thanks to the rule $(\Rightarrow_R)$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} G$.
- $G \equiv C \Rightarrow G'$. Then $\langle \Delta, \Gamma, G\rangle \in \mathcal{S}$ implies that $lfp(T), \Delta, \Gamma \cup \{C\} \models G'$. Clearly, $ord(\langle \Delta, \Gamma, G\rangle) = ord(\langle \Delta, \Gamma \cup \{C\}, G'\rangle)$ and G' is a strict subformula of G, so $\langle \Delta, \Gamma \cup \{C\}, G'\rangle < \langle \Delta, \Gamma, G\rangle$. Therefore, by the induction hypothesis we obtain $\Delta; \Gamma, C \vdash_{\mathcal{UC}} G'$. Thanks to the rule $(\Rightarrow_{C_R})$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} G$.
- $G \equiv \exists x G'$. Then $\langle \Delta, \Gamma, G\rangle \in \mathcal{S}$ implies that there is a constraint C and a variable y such that:
  - y does not occur free in Δ, Γ, $\exists x G'$.
  - $\Gamma \vdash_{\mathcal{C}} \exists y C$.
  - $lfp(T), \Delta, \Gamma \cup \{C'\} \models G'[y/x]$.

  Clearly, $ord(\langle \Delta, \Gamma, G\rangle) = ord(\langle \Delta, \Gamma \cup \{C'\}, G'[y/x]\rangle)$ and $G'[y/x]$ is a renaming of a strict subformula of G, so $\langle \Delta, \Gamma \cup \{C\}, G'[y/x]\rangle < \langle \Delta, \Gamma, G\rangle$. Therefore, by the induction hypothesis we obtain $\Delta; \Gamma, C \vdash_{\mathcal{UC}} G'[y/x]$. Thanks to the rule $(\exists_R)$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} G$.
- $G \equiv \forall x G'$. Then $\langle \Delta, \Gamma, G\rangle \in \mathcal{S}$ implies that there is a variable y such that:
  - y does not occur free in Δ, Γ, $\forall x G'$.
  - $lfp(T), \Delta, \Gamma \models G'[y/x]$.

  Clearly, $ord(\langle \Delta, \Gamma, G\rangle) = ord(\langle \Delta, \Gamma, G'[y/x]\rangle)$ and $G'[y/x][x/y] \equiv G'$ is a strict subformula of G, so $\langle \Delta, \Gamma, G'[y/x]\rangle < \langle \Delta, \Gamma, G\rangle$. Therefore, by the induction hypothesis we obtain $\Delta; \Gamma \vdash_{\mathcal{UC}} G'[y/x]$. Thanks to the rule $(\forall_R)$, it follows that $\Delta; \Gamma \vdash_{\mathcal{UC}} G$. $\square$

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult `http://aib.informatik.rwth-aachen.de/` **or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,**
Email: `biblio@informatik.rwth-aachen.de`

95-11 * M. Staudt / K. von Thadden: Subsumption Checking in Knowledge Bases

95-12 * G.V. Zemanek / H.W. Nissen / H. Hubert / M. Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

95-13 * M. Staudt / M. Jarke: Incremental Maintenance of Externally Materialized Views

95-14 * P. Peters / P. Szczurko / M. Jeusfeld: Business Process Oriented Information Management: Conceptual Models at Work

95-15 * S. Rams / M. Jarke: Proceedings of the Fifth Annual Workshop on Information Technologies & Systems

95-16 * W. Hans / St. Winkler / F. Sáenz: Distributed Execution in Functional Logic Programming

96-1 * Jahresbericht 1995

96-2 M. Hanus / Chr. Prehofer: Higher-Order Narrowing with Definitional Trees

96-3 * W. Scheufele / G. Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates

96-4 K. Pohl: PRO-ART: Enabling Requirements Pre-Traceability

96-5 K. Pohl: Requirements Engineering: An Overview

96-6 * M. Jarke / W. Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools

96-7 O. Chitil: The ς-Semantics: A Comprehensive Semantics for Functional Programs

96-8 * S. Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics

96-9 M. Hanus (Ed.): Proceedings of the Poster Session of ALP'96 — Fifth International Conference on Algebraic and Logic Programming

96-10 R. Conradi / B. Westfechtel: Version Models for Software Configuration Management

96-11 * C. Weise / D. Lenzkes: A Fast Decision Algorithm for Timed Refinement

96-12 * R. Dömges / K. Pohl / M. Jarke / B. Lohmann / W. Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models

96-13 * K. Pohl / R. Klamma / K. Weidenhaupt / R. Dömges / P. Haumer / M. Jarke: A Framework for Process-Integrated Tools

96-14 * R. Gallersdörfer / K. Klabunde / A. Stolz / M. Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

96-15 * H. Schimpe / M. Staudt: VAREX: An Environment for Validating and Refining Rule Bases

96-16 * M. Jarke / M. Gebhardt, S. Jacobs, H. Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

96-17 M. Jeusfeld / T. X. Bui: Decision Support Components on the Internet

96-18 M. Jeusfeld / M. Papazoglou: Information Brokering: Design, Search and Transformation

96-19 * P. Peters / M. Jarke: Simulating the impact of information flows in networked organizations

96-20 M. Jarke / P. Peters / M. Jeusfeld: Model-driven planning and design of cooperative information systems

96-21 * G. de Michelis / E. Dubois / M. Jarke / F. Matthes / J. Mylopoulos / K. Pohl / J. Schmidt / C. Woo / E. Yu: Cooperative information systems: a manifesto

96-22 * S. Jacobs / M. Gebhardt, S. Kethers, W. Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

96-23 * M. Gebhardt / S. Jacobs: Conflict Management in Design

97-01 Jahresbericht 1996

97-02 J. Faassen: Using full parallel Boltzmann Machines for Optimization

97-03 A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

97-04 M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

97-05 * S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

97-06 M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

97-07 P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

97-08 D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting

97-09 C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets

97-10 M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

97-13 M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

97-14 R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

97-15 G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

98-01 * Jahresbericht 1997

98-02 S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes

98-03 S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

98-04 * O. Kubitz: Mobile Robots in Dynamic Environments

98-05 M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems

98-07 M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

98-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen

98-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases

98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML

98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

99-01 * Jahresbericht 1998

99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager

99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

99-07 Th. Wilke: CTL+ is exponentially more succinct than CTL

99-08 O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge / Marcin Jurdzinski: A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-04 Andreas Becks / Stefan Sklorz / Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 Mareike Schoop: Cooperative Document Management

2000-06 Mareike Schoop / Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 * Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

2000-08 Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus

2001-05 Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC Languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark / Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl / Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig / Martin Leucker / Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl / Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

2002-05 Horst Lichter / Thomas von der Maßen / Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines

2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities

2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java

2002-09 Markus Mohnen: Interfaces with Default Implementations in Java

2002-10 Martin Leucker: Logics for Mazurkiewicz traces

2002-11 Jürgen Giesl / Hans Zantema: Liveness in Rewriting

2003-01 * Jahresbericht 2002

2003-02 Jürgen Giesl / René Thiemann: Size-Change Termination for Term Rewriting

2003-03 Jürgen Giesl / Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl / René Thiemann / Peter Schneider-Kamp / Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding / Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter / Thomas von der Maßen / Alexander Nyßen / Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl / René Thiemann / Peter Schneider-Kamp / Stephan Falke: Mechanizing Dependency Pairs

2004-02 Benedikt Bollig / Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner / Femke van Raamsdonk / Joe Wells (eds.): Proceedings of the Second International Workshop on Higher-Order Rewriting (HOR 2004)

2004-04 Slim Abdennadher / Christophe Ringeissen (eds.): Proceedings of the Fifth International Workshop on Rule-Based Programming (RULE 2004)

2004-05 Herbert Kuchen (ed.): Proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)

2004-06 Sergio Antoy / Yoshihito Toyama (eds.): Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004)

2004-07 Michael Codish / Aart Middeldorp (eds.): Proceedings of the 7th International Workshop on Termination (WST 2004)

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.