INSTITUT FÜR INFORMATIK

Programmiersprachen und Rechenkonzepte

24. Workshop der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“

Bad Honnef, 2.-4. Mai 2007

Michael Hanus, Bernd Braßel (Hrsg.)

Bericht Nr. 0707

August 2007

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Programmiersprachen und Rechenkonzepte

24. Workshop der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“ Bad Honnef, 2.-4. Mai 2007

Michael Hanus, Bernd Braßel (Hrsg.)

Bericht Nr. 0707
August 2007

e-mail: mh@informatik.uni-kiel.de, bbr@informatik.uni-kiel.de

Dieser Bericht enthält eine Zusammenstellung der Beiträge des 24. Workshops Programmiersprachen und Rechenkonzepte, Physikzentrum Bad Honnef, 2.-4. Mai 2007.

Vorwort

Seit 1984 veranstaltet die GI–Fachgruppe „Programmiersprachen und Rechenkonzepte“, die aus den ehemaligen Fachgruppen 2.1.3 „Implementierung von Programmiersprachen“ und 2.1.4 „Alternative Konzepte für Sprachen und Rechner“ hervorgegangen ist, regelmäßig im Frühjahr einen Workshop im Physikzentrum Bad Honnef. Das Treffen dient in erster Linie dem gegenseitigen Kennenlernen, dem Erfahrungsaustausch, der Diskussion und der Vertiefung gegenseitiger Kontakte.
In diesem Forum werden Vorträge und Demonstrationen sowohl bereits abgeschlossener als auch noch laufender Arbeiten vorgestellt, unter anderem (aber nicht ausschließlich) zu Themen wie

- Sprachen, Sprachparadigmen
- Korrektheit von Entwurf und Implementierung
- Werkzeuge
- Software-/Hardware-Architekturen
- Spezifikation, Entwurf
- Validierung, Verifikation
- Implementierung, Integration
- Sicherheit (Safety und Security)
- eingebettete Systeme
- hardware-nahe Programmierung

In diesem Technischen Bericht sind einige der präsentierten Arbeiten zusammen gestellt. Allen Teilnehmern des Workshops möchten wir danken, dass sie durch ihre Vorträge, Papiere und Diskusion den jährlichen Workshop erst zu einem spannenden Ereignis machen. Ein besonderer Dank gilt den Autoren die mit ihren vielfältigen Beiträgen zu diesem Band beigetragen haben. Ein abschließender Dank gebührt noch den Mitarbeitern des Physikzentrums Bad Honnef, die durch ihre umfassende Betreuung für eine angenehme und anregende Atmosphäre gesorgt haben.

Kiel, im August 2007 Michael Hanus, Bernd Braßel

Inhaltsverzeichnis

Glass-Box Testing of Functional Logic Programs*

Sebastian Fischer[1] and Herbert Kuchen[2]

[1] Department of Computer Science
University of Kiel, Germany
sebf@informatik.uni-kiel.de
[2] Department of Information Systems
University of Münster, Germany
kuchen@uni-muenster.de

Abstract. We employ the narrowing-based execution mechanism of the functional logic programming language Curry in order to automatically generate a system of test cases for glass-box testing of Curry programs. The test cases for a given function are computed by narrowing a call to that function with initially uninstantiated arguments. The generated test cases are produced w.r.t. a selected code-coverage criterion such as control-flow coverage. Besides an adaption of the notion of control-flow coverage to functional (logic) programming, we present a novel coverage criterion for this programming paradigm.

1 Introduction

Curry [1] is a programming language that aims at integrating different declarative programming paradigms into a single programming language. Its syntax is similar to Haskell [2] but it uses a different evaluation mechanism [3] and supports free variables and nondeterministic computations like logic languages. Declarative programming languages offer a high degree of abstraction which eases the development of complex systems. They help to write more readable and re-usable code [4] which, however, can still contain errors. Ultimate confidence in the correctness of a program can only be achieved by proofs with regard to a complete and formal specification. Declarative languages are usually based on a formal semantics, which helps to formally certify some properties of declarative programs. However, proving complex systems correct turns out to be too difficult and time consuming, even in the context of declarative programming. Therefore, heuristic approaches to expose errors gain importance also in this area. Recently, techniques have been proposed to systematically debug programs that are known to be erroneous [5–9]. However, little can be found in the literature about the systematic testing of functional logic programs. This paper intends to help filling this gap. As the functional part of Curry is basically Haskell98 without type classes, we can also apply our tool to generate test cases for Haskell programs.

* This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-2.

1.1 Testing

Approaches to software testing can be divided into black-box testing, where test cases are deduced from a specification without taking the concrete implementation into account, and glass-box testing[3], which aims at a systematic coverage of the code. Both approaches do not exclude each other, but can be combined. Black-box testing is often used to evaluate larger parts of an application. Glass-box testing is preferred for testing small program units with a complex algorithmic structure, which makes it hard to deduce all possible behaviors from the specification. Since declarative programs typically consist of a sequence of small function definitions, glass-box testing is even more suited for declarative languages than for imperative languages. Moreover, the lack of side effects renders it possible to test parts of an algorithm independently. We will focus on glass-box testing in this paper.

Regardless of the way test cases have been generated, they can not only be used for testing a program unit once. More importantly, they can be used for so-called regression testing, i.e., the suite of collected test cases is automatically processed in order to check whether some change in the program affects the already implemented functionality.

1.2 Related Work

Existing tools for testing functional programs generate random test cases or deduce them from a specification [10, 11], i.e., they are based on black-box testing. Since they do not take the implementation into account, they cannot ensure that all parts of the program are actually executed by the test cases. Hence, errors in uncovered parts of the program may remain undetected.

In [12, 13] an approach for generating glass-box test cases for Java is presented. Techniques known from logic programming are incorporated into a symbolic Java virtual machine for code-based test-case generation. A similar approach based on a Prolog simulation of a Java Virtual Machine is presented in [14]. A related approach to test-case generation for logic programs is discussed in [15]. Here, test cases are not generated by executing the program but by first computing constraints on input arguments that correspond to an execution path and then solving these constraints to obtain test inputs that cover the corresponding path.

We transfer the approach for Java presented in [12, 13] to the functional logic programming language Curry. However, instead of extending an abstract machine by components for backtracking and handling logic variables, we can just employ the usual execution mechanism of Curry, since it already provides these features. Actually, this approach to test-case generation seems to be tailor-made for functional logic languages. Our approach is based on a transformation of Curry programs and, hence, not restricted to a specific Curry implementation.

[3] Glass-box testing is often called white-box testing, although one can of course not look inside a white box.

Besides testing, there are different approaches for debugging functional logic programs, which are remotely related to testing. *Trace debuggers* [16, 6, 7] record information about a specific (usually erroneous) program run and present it to the user in a structured way. Different kinds of views can be employed to analyze the recorded program trace. Algorithmic debuggers ask the user a sequence of questions about the correctness of subcomputations in order to find an error in a program [9]. Algorithmic debugging can be implemented as an interactive view on a program trace. With tools for *observational debugging* [16, 8] the programmer can annotate her code with observer functions to record parts of the computation. With this approach, only selected parts considered interesting by the programmer are recorded, which saves memory. Both tracing and observational debugging are approaches to locate an already present error. They are not aiming at but can be combined with testing.

Due to space restrictions, we only discuss our approach informally. A full version of this paper including an introduction to Curry, implementation details and experimental results is presented in [17]. In Section 2 we introduce the concept of code coverage and consider two different code-coverage criteria for functional logic programs. In Section 3 we describe how test cases can be generated w.r.t. different code-coverage criteria before we conclude and point out directions for future work in Section 4.

2 Code Coverage

Before we explain our approach to generate test cases for Curry, let us briefly recall the basics of glass-box testing for imperative languages. For testing a function one collects a set of test cases which covers the possible behaviors of the function in a reasonable way. A *test case* is a pair of a function call and a corresponding expected result.

Since there are often infinitely many possible inputs and corresponding computation paths through a program, it is impossible to test all of them. But even if the number of paths was finite, many of them would cover the same control and data flow and would hence be equivalent from the point of view of testing. For cost effective testing, one is interested in a minimal set of test cases covering the code according to a selected coverage criterion.

Classical criteria known from testing imperative programs are coverage of the (nodes and) edges of the control-flow graph and the so-called def-use chain coverage [18]. The latter requires that each sequence of statements is covered, which starts with a statement computing a value and ends with a statement, where this value is used and where this value is not modified in between.

It is important to understand that no coverage criterion guarantees the absence of errors, including the criteria presented in this paper. They rather try to detect as many errors as possible with limited effort. It is always possible to find examples where a given coverage criterion fails to expose an error. Nevertheless coverage criteria are useful in order to find the majority of the errors, in particular in algorithmically complex code. Remaining errors can be eliminated, e.g., by black-box testing.

Lazy declarative languages like Curry have no assignments and a rather complicated control-flow (due to laziness), which cannot easily be represented by a control-flow graph. Therefore, we cannot simply transfer the notions of code coverage from the imperative to the declarative world, but need adapted notions. Here, we will present two different coverage criteria: *Global Branch Coverage* (GBC) and *Function Coverage* (FC) which correspond both to variants of control-flow coverage in imperative languages. However, let us point out that our approach works with any coverage criterion. We only require that the coverage can be described by a set of *coverable items*. These items can represent control- and/or data-flow information.

2.1 Global Branch Coverage

In glass-box testing for imperative languages, typically only code sequences that are part of a single function or procedure declaration are considered. Due to control structures like loops present in imperative languages, there is often no need to consider more than one function to obtain interesting test cases.

In declarative programming, (recursive) function calls are used to express control structures and to replace loops. Since the functions in a declarative program are typically very small and consist of a few lines only, it is not sufficient in practice to cover only the branches of the function to be tested. Thus, we will aim at covering the branches of all the directly and indirectly called functions, too.

The main idea of this approach is that we label all the alternatives in *or* and *case* expressions with pairwise different labels and that we try to make sure that every labelled alternative will be executed at least once by some test case. This means that each alternative of an *or* expression and each e_i in a *case* expression

$$case\ e\ of\ \{p_1 \to e_1; \ldots ; p_n \to e_n\}$$

will be evaluated to head normal form. *Or* expressions are used to model nondeterministic alternatives and *case* expressions denote pattern matching. See [19] for more details. To test the definition of a function `f`, we can compute the set of all functions reachable from `f` and generate test cases that cover all branches of these functions.

2.2 Function Coverage

Covering all branches of all reachable functions may produce a lot of overhead. Moreover, it may be impossible to cover all branches of the called functions, since the parameters passed to them do not permit this. Thus, we are interested in a criterion which focuses on the code of the function to be tested. As mentioned, a simple coverage of all branches of the considered function will expose too few errors in practice. Thus, we extend this approach slightly. We will ensure that in addition to all branches of the original call to the considered function, also all branches of all recursive calls to that function have to be executed. As shown in [17], the resulting criterion called *Function Coverage* (FC) works quite well in practice. If we assume that all functions called by the considered function `f` have been tested before, FC allows to find strictly more errors than GBC, since FC will then not only ensure that every branch is executed once, but that every branch will be checked in every call to the considered function.

As for GBC, it may be sometimes impossible to cover all branches for each recursive call, since the actual parameters do not permit this. In this case, we will confine ourselves to execute the reachable branches.

As a simple example consider the labeled definition of the append function `(++)` for lists:

```
(++) :: [a] -> [a] -> [a]
l ++ ys = case l of {
  [] -> ⟨1⟩ ys;
  (x:xs) -> ⟨2⟩ x : (xs++xs) }
```

Here, the test case `[0] ++ [] = [0]` suffices to reach global branch coverage without exposing the error in branch ⟨2⟩ (`xs++xs` should be `xs++ys`).

On the other hand, there are two calls of `(++)` to be covered with FC, the initial call and the recursive call in branch ⟨2⟩. We require both calls to execute all branches of `(++)`. With the above test case, the recursive call does not execute branch ⟨2⟩ and our tool also generates the test case `[0] ++ [1] = [0]` that exposes the error.

If we fail to fix the error and write `ys++xs`, then the test cases `[] ++ [] = []` and `[0] ++ [1] = [0,1]` suffice to fulfill both GBC and FC. None of the coverage criteria exposes this error which hints at the incomplete nature of every coverage criterion discussed at the beginning of this section.

3 Generating Test Cases

Let us now consider, how we can generate a system of test cases for some coverage criterion. For each test case, we need to find a sequence of parameters with a corresponding expected result. Moreover, we would like the set of test cases to cover all coverable items according to the selected criterion.

A naive way of producing a set of test cases in a functional logic language is to call the function `f` to be tested with a sequence of unbound logic variables `x1,...,xn` as parameters and to compute all possible solutions of `f x1...xn`. Each computation will bind the logic variables to some terms such that the function called with these terms as parameters causes the desired coverage. The result of this computation will be the desired expected result for the test case. Note, that we do not need to integrate a constraint solver like [13] because in a functional-logic language free variables can be bound by the built-in narrowing mechanism. In order to obtain a list of test cases for a function `f`, we could collect the results of the test-case generation with the primitive function `allValues` using encapsulated search [20]:

```
allValues (let x1,...,xn free
           in ((x1,...,xn), f x1 ... xn))
```

Unfortunately, this naive approach will in general fail to produce the desired minimal set of test cases. Typically, it will even generate an infinite number of them. This does not mean that this narrowing-based generation of test cases cannot be used at all. We rather have to make sure that the computation is controlled in such a way that not too many test cases are generated.

In our approach, we record the set of covered items during the computation along with every computed result. Note that this approach is independent of the selected coverage criterion. Given this additional information, we can demand further results until we obtain the desired coverage. Thus, we need to be able to compute the result of encapsulated search lazily. Moreover we have to rely on a fair search strategy, as, e.g., offered by KiCS [21], that ensures that all results are eventually computed.

With a non-fair depth-first search, the overall computation would try to find an infinite amount of solutions for some subexpression before considering an alternative which causes the missing items to be covered. In particular in situations, where the desired coverage cannot be achieved (as explained in Subsection 2.2), additional means for controlling the computation are required. One possibility is to limit the recursion depth based on an additional parameter which keeps track of it. Alternatively, the computation could be stopped, if the last n generated test cases do not cover any new coverable item (n has to be configured appropriately). The simplest means for controlling the computation is to limit the amount of generated test cases to some fixed n (which has to be configured appropriately). Our system can be combined with any of these alternatives.

The reader may wonder, why we need to use the generated test cases at all for testing, since they reflect the actual behavior of the system and one can hence observe an erroneous behavior by just looking at a generated test case. The reason is that the test cases are needed for regression testing, i.e., in order to check whether a change of the system does not destroy the already working functionality.

The approach described so far does not ensure a minimal set of test cases. In order to get a minimal set of test cases, we need an additional step which removes redundant test cases. Obviously, this problem is the set covering problem which is known to be NP-complete [22]. Since it is not essential in our context that we really find a minimal solution, we are happy with any heuristic producing a small solution. Sometimes, a larger set of smaller test cases may be preferred over a smaller set of larger test cases because small test cases are usually easier to verify by humans. However, for regression testing the smaller set is always cheaper to check.

Not all Curry implementations support the guessing of numbers in arithmetic operations. Currently only KiCS [21] does it – by implementing numbers as algebraic datatype [23]. In order to be able to handle guessing in arithmetic operations in other Curry implementations as well, we employ the system of constraint solvers presented in [13]. During the computation, generated constraints are checked for consistency against other already generated constraints in order to select valid computation paths. After the computation, we solve the generated constraints and instantiate numerical variables according to the computed solution in order to produce a test case.

4 Conclusions and Future Work

We have shown how glass-box testing based on systematic coverage of the code can be adapted from the imperative world to a functional logic programming language.

We have developed two coverage criteria for the functional (logic) programming paradigm and presented a tool which generates a system of test cases automatically according to a selected coverage criterion. This tool employs the narrowing-based execution mechanism of Curry in order to generate test-cases. The computation is controlled by the set of items to be covered and redundant test cases are eliminated by a heuristic for the set covering problem.

We have demonstrated that *Function Coverage* exposes errors that can remain undetected in test cases that satisfy *Global Branch Coverage*. On the other hand, it usually does not expose errors in reachable functions, so these need to be tested separately.

As future work, we plan to investigate the notion of data-flow coverage in the context of declarative programming. We also plan to integrate specifications that are employed to automatically verify the generated test cases.

References

1. Hanus, M., et al.: Curry: An integrated functional logic language (version 0.8.2). Available at URL `http://www.informatik.uni-kiel.de/~curry` (2006)
2. Peyton Jones, S., ed.: Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press (2003)
3. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. Journal of the ACM **47**(4) (2000) 776–822
4. Hughes, J.: Why Functional Programming Matters. Computer Journal **32**(2) (1989) 98–107
5. Chitil, O., Runciman, C., Wallace, M.: Freja, hat and hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In: Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000), Springer LNCS 2011 (2001) 176–193
6. Braßel, B., Hanus, M., Huch, F., Vidal, G.: A semantics for tracing declarative multi-paradigm programs. In: Proc. of the 6th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'04), ACM Press (2004) 179–190
7. Braßel, B., Fischer, S., Huch, F.: A program transformation for tracing functional logic computations. In: International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006). LNCS, Springer (2006) To appear.

8. Braßel, B., Chitil, O., Hanus, M., Huch, F.: Observing functional logic computations. In: Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04), Springer LNCS 3057 (2004) 193–208
9. Caballero, R., Rodríguez-Artalejo, M.: Ddt: a declarative debugging tool for functional-logic languages. In: Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004), Springer LNCS 2998 (2004) 70–84
10. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices **35**(9) (September 2000) 268–279
11. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic automated software testing. In Peña, R., ed.: The 14th International workshop on the Implementation of Functional Languages, IFL'02, Selected Papers. Volume 2670 of LNCS., Madrid, Spain, Springer (September 2002) 84–100
12. Müller, R., Lembeck, C., Kuchen, H.: A symbolic Java virtual machine for test-case generation. In: Proceedings IASTED. (2004)
13. Lembeck, C., Caballero, R., Müller, R., Kuchen, H.: Constraint solving for generating glass-box test cases. In: Proceedings of International Workshop on Functional and (Constraint) Logic Programming (WFLP). (2004) 19–32
14. Albert, E., Gómez-Zamalloa, M., Hubert, L., Puebla, G.: Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In: Ninth International Symposium on Practical Aspects of Declarative Languages. LNCS, Springer-Verlag (January 2007) To appear.
15. Mweze, N., Vanhoof, W.: Automatic generation of test inputs for Mercury programs. In: International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006). LNCS, Springer (2006) To appear.
16. ART: Hat – the Haskell tracer (version 2.04). Available at URL `http://haskell.org/hat/` (2005)
17. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: Proc. of the 9th International Symposium on Principles and Practice of Declarative Programming (PPDP 2007), ACM Press (2007) to appear.
18. Pressman, R.S.: Software Engineering: a Practitioner's Approach. McGraw-Hill, Inc. (1992)
19. Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G.: Operational semantics for declarative multi-paradigm languages. Journal of Symbolic Computation **40**(1) (2005) 795–829
20. Braßel, B., Hanus, M., Huch, F.: Encapsulating non-determinism in functional logic computations. Journal of Functional and Logic Programming **2004**(6) (2004)
21. Braßel, B., Huch, F.: Translating Curry to Haskell. In: Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005), ACM Press (2005) 60–65
22. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press, Cambridge, Mass. (1990)
23. Braßel, B., Fischer, S., Huch, F.: Declaring numbers. In: Proc. of the 16th International ACM SIGPLAN Workshop on Functional and (Constraint) Logic Programming (WFLP 2007). (2007) to appear.

Putting Declarative Programming into the Web: Translating Curry to JavaScript[⋆]

– Extended Abstract –

Michael Hanus

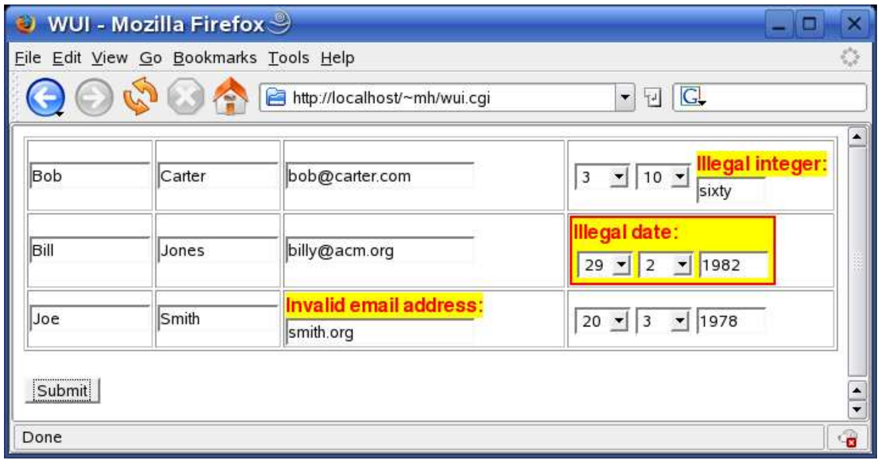
Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
`mh@informatik.uni-kiel.de`

Abstract. We propose a framework to construct web-oriented user interfaces in a high-level way by exploiting declarative programming techniques. Such user interfaces are intended to manipulate complex data in a type-safe way, i.e., it is ensured that only type-correct data is accepted by the interface, where types can be specified by standard types of a programming language as well as any computable predicate on the data. If the client's browser has JavaScript enabled, one could also check the correctness of the data on the client side providing immediate feedback to the user. In order to release the application programmer from the tedious details to interact with JavaScript, we propose an approach where the programmer must only provide a declarative description of the requirements of the user interface from which the necessary JavaScript programs and HTML forms are automatically generated.

1 Motivation

The implementation of software systems can be coarsely classified into two parts: the implementation of the application logic and the implementation of the user interface. If one uses declarative programming languages, the first part can be implemented with reasonable efforts. In contrast, the construction of the user interface is usually complex and tedious. In order to simplify the latter task, scripting languages with toolkits and libraries, like Tcl/Tk, Perl, or PHP, are one approach to support this goal. Since scripting languages often lack support for the development of complex and reliable software systems (e.g., no static type and interface checking, limited code reuse due to the lack of high-level abstractions), they are often used to implement the user interface whereas the application logic is implemented in some other language. This approach creates new problems since it is well known that such combinations could cause security leaks in web applications [9]. Therefore, an alternative solution is the embedding of such domain-specific languages in high-level languages able to provide appropriate abstractions. This paper follows the latter alternative and considers an approach to construct web user interfaces (WUIs) in a declarative way. The

⋆ This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-2. A full version of this extended abstract has appeared in [6].


WUI - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/~mh/wui.cgi
Submit
Done

Bob	Carter	bob@carter.com	3 10 Illegal integer: sixty
Bill	Jones	billy@acm.org	Illegal date: 29 2 1982
Joe	Smith	Invalid email address: smith.org	20 3 1978

Fig. 1. A WUI for a list of persons containing various input errors

core idea of the declarative construction of WUIs has been presented in [4]. In this paper we combine this construction with the existing features of scripting languages by compiling parts of the declarative interface specifications into a scripting language available in almost all web browsers: JavaScript.

In order to provide an example of the new approach presented in this paper, we give a short summary of the framework to construct WUIs presented in [4]. This approach to construct WUIs is useful in situations where data of an application program should be manipulated via standard web browsers (i.e., by HTML forms). The application program supplies the WUI with the current data of the application and an operation to store the modified data. Furthermore (and most important), it provides a type-oriented specification of the WUI structure that matches the type of the application data. For this purpose, the WUI framework contains a set of *basic WUIs* to manipulate data of basic types, e.g., integers, truth values, strings, finite sets, and a set of *WUI combinators* to construct WUIs for complex data types from simpler types similarly to type constructors in programming languages. For instance, there are combinators for tuples, lists, union types etc. The framework ensures that the user inputs only *type-correct data*, i.e., if the user tries to input illegal data (e.g., incorrect integer constants, empty strings, wrong dates or email addresses), the WUI does not accept the data and asks the user to correct the input. Figure 1 shows an example of a WUI for a list of persons containing various input errors. Note that errors can occur not only in individual input fields but also as illegal combinations of different fields, like the date in the second row. Thanks to this feature of the WUI framework, the application program need not check the input data and perform appropriate actions (e.g., providing error forms to correct the input etc). This considerably simplifies the task of programming the user interface.

In this paper we show how to exploit the existence of JavaScript interpreters in web browsers in order to increase the functionality of WUIs. By translating conditions in WUIs into JavaScript programs, one can check user inputs on the client side, i.e., forms with illegal inputs are not sent to the web server. This feature reduces the number of client/server interactions and provides instantaneous feedback on incorrect inputs on the client side. However, Curry is a powerful language with advanced programming features (e.g., higher-order functions, laziness, logic variables, constraint solving, concurrency). Thus, it is not reasonable to translate into JavaScript all possible conditions that can be implemented in Curry, since this might finally require to communicate a complete Curry implementation to the web client. This is not only inefficient (since JavaScript is usually interpreted) or impossible (due to space and time limitations of the JavaScript interpreter) but also not necessary: the correctness of the user input is always checked on the server side (due to the principle in web programming that one should *never trust user inputs from web browsers* even if they are checked by scripts on the client side, since one never knows whether the input comes from a human using a web browser or another malicious program [9]). Thus, one is free to select only particular conditions that are easy to translate into JavaScript. This is the idea used in the following in order to get a reasonable and practically applicable combination of two different worlds of programming.

2 Basics: Curry, HTML, JavaScript

We assume familiarity with functional logic programming and the language Curry (see [5] for a recent survey and [8] for details about Curry).

Writing programs that generate HTML documents requires a decision about the representation of HTML documents. A textual representation (as often used in CGI scripts written in Perl or with the Unix shell) is very poor since it does not avoid certain syntactical errors (e.g., unbalanced parenthesis) in the generated document. Thus, it is better to introduce an abstraction layer and model HTML documents as elements of a specific data type together with a wrapper function that is responsible for the correct textual representation of this data type. Since HTML documents have a tree-like structure, they can be represented in functional or logic languages in a straightforward way. For instance, the type of HTML expressions is defined in Curry as follows:

```
data HtmlExp = HtmlText   String
             | HtmlStruct String [(String,String)] [HtmlExp]
```

Thus, an HTML expression is either a plain string or a structure consisting of a tag (e.g., `b`,`em`,`h1`,`h2`,...), a list of attributes (name/value pairs), and a list of HTML expressions contained in this structure. Thus, the HTML code

```
<p>This is an <i>example</i></p>
```

is represented by the data term

```
HtmlStruct "p" [] [HtmlText "This is an ",
                   HtmlStruct "i" [] [HtmlText "example"]]
```

Since it is tedious to write HTML documents in this form, one can provide various functions as useful abbreviations (`htmlQuote` transforms characters with a special meaning in HTML into their HTML quoted form):

```
htxt   s     = HtmlText   (htmlQuote s)
par    hexps = HtmlStruct "p" [] hexps
italic hexps = HtmlStruct "i" [] hexps
...
```

Then we can write the example as

```
par [htxt "This is an ", italic [htxt "example"]]
```

A *dynamic web page* is an HTML document (with header information) that is computed by a program at the time when the page is requested by a client (usually, a web browser). For this purpose, there is a data type

```
data HtmlForm = HtmlForm String [FormParam] [HtmlExp]
```

to represent complete HTML documents, where the first argument to `HtmlForm` is the document's title, the second argument contains optional parameters (e.g., cookies, style sheets), and the third argument is the document's content. Since a dynamic web page should represent information that often depends on the environment of the web server (e.g., stored in databases), a dynamic web page has always the type "`IO HtmlForm`", i.e., it is an I/O action [10] that retrieves some information from the environment and produces a web document.

Dynamic web pages usually process user inputs, placed in various input elements (e.g., text fields, text areas, check boxes) of an HTML form, in order to generate a user-specific result. For this purpose, the HTML library of Curry [3] provides an abstract programming model that can be characterized as *programming with call-back functions*. A web page with user input and buttons for submitting the input to a web server is modeled by attaching an *event handler* to each submit button that is responsible for computing the answer document. We omit further details here (they can be found in [3]) since we consider a more abstract layer to construct web-based user interfaces that will be described in Section 3.

JavaScript [2] is an imperative scripting language that can be embedded in HTML documents. JavaScript programs are executed by the client's web browser and have access, via a document object model, to the resources of the browser, in particular, to the HTML document shown in the browser. For this purpose, the document is represented as a hierarchical object structure where the attributes of each object can be accessed or manipulated by the standard "dot notation". For instance, the class identifier (whose meaning is usually defined in a style sheet) of an object `elem` in an HTML document can be changed to `myStyle` by the assignment `elem.className = "myStyle"`.

JavaScript programs are usually executed by the web browser when some event occurs. For instance, if a text input field in an HTML form has an attribute `onblur="f(3)"`, the function call `f(3)` is evaluated whenever the user leaves this input field. We exploit this functionality of JavaScript to check the user input on the client side before the complete web form is submitted to the server.

3 Type-Oriented Web User Interfaces

In this section we review the type-oriented construction of WUIs, as proposed in [4], from a programmer's point of view, before we discuss its extension to include JavaScript in the next section.

The basic idea of our WUI framework is to provide an easily applicable method to implement an interface for the manipulation of data of an application domain. Thus, we assume that the application program supplies a WUI with the current state of the data and an operation to store the data modified by the user and acknowledge it to the user. Thus, the main operation to construct a WUI has the type signature

```
mainWUI :: WuiSpec a -> a -> (a -> IO HtmlForm) -> IO HtmlForm
```

so that an expression (`mainWUI` *wspec d store*) evaluates to a web page containing an editor that shows the current data d and executes (*store* d') when the user submits the modified data d'. The operation *store* (also called *update form*) usually stores the modified data in a file or database, returns a web page that informs the user about the successful (or failed) modification, and proceeds with a further interaction.

The first argument of type `WuiSpec a`, also called *WUI specification*, specifies the kind of interface used to manipulate data of type `a`. This is necessary since there are usually various alternative interaction forms for identical data types. For instance, integer values can be manipulated in text fields (see last column in the table of Fig. 1) or, if the set of possible values is small, via selection boxes (see the two columns before the last one in Fig. 1). Therefore, the WUI framework provides a couple of predefined interaction forms for various data types. For instance, there are predefined entities

```
wString :: WuiSpec String
wInt    :: WuiSpec Int
```

to manipulate strings and integer values in simple text input fields, respectively. Similarly, there is an entity

```
wSelectInt :: [Int] -> WuiSpec Int
```

to select a value from a list of integers by a selection box.

In order to construct WUIs for complex data types, there are *WUI combinators* that are mappings from simpler WUIs to WUIs for structured types. For instance, there is a family of WUI combinators for tuple types:

```
wPair   :: WuiSpec a -> WuiSpec b -> WuiSpec (a,b)
```

```
wTriple :: WuiSpec a -> WuiSpec b -> WuiSpec c -> WuiSpec (a,b,c)
w4Tuple :: WuiSpec a -> WuiSpec b -> WuiSpec c ->
           WuiSpec d -> WuiSpec (a,b,c,d)
...
```

Thus,

```
wDate = wTriple (wSelectInt [1..31]) (wSelectInt [1..12]) wInt
wPerson = w4Tuple wString wString wString wDate
```

define WUI specifications for dates and persons consisting of first name, surname, email address, and date of birth. To manipulate lists of data objects, there is a WUI combinator for list types:

```
wList :: WuiSpec a -> WuiSpec [a]
```

Thus, to manipulate lists of persons as shown in Fig.1, we apply the main construction operation `mainWUI` to the WUI specification `(wList wPerson)`, which is of type

```
WuiSpec [(String,String,String,(Int,Int,Int))] ,
```

and appropriate data values and update forms.

As discussed above, our type-oriented construction of WUIs leads to type-safe user interfaces, i.e., the user can only enter type-correct data so that the application does not need to perform any checks for this purpose. Up to now, type-correctness is interpreted w.r.t. the types of the underlying programming language. However, many applications require a more fine-grained definition of types. For instance, not every triple of natural numbers that can be entered with the WUI `wDate` above is acceptable, e.g., the triple `(29,2,1982)` is illegal from an application point of view. In order to support also correctness checks for such *application-dependent type constraints*, our framework allows to attach a computable predicate to any WUI: there is an operation (also defined as an infix operator)

```
withCondition :: WuiSpec a -> (a->Bool) -> WuiSpec a
```

that combines a WUI specification with a predicate on values of the same type so that the result specifies a WUI that accepts only values satisfying the given predicate. For instance,

```
wEmail :: WuiSpec String
wEmail =  wString `withCondition` isEmail
```

defines a WUI that accepts only syntactically correct email addresses provided that `isEmail` is a predicate on strings that is satisfied if its argument is a syntactically valid email address.

If application-specific conditions on input values are added, appropriate error messages should be provided. For this purpose, there is an operation (infix operator)

```
withError :: WuiSpec a -> String -> WuiSpec a
```

that combines a WUI specification with a specific message which is shown in case of inputs that do not satisfy the input constraints. For instance, we can improve the definition of `wEmail` with an appropriate error message as follows:

```
wEmail = wString `withCondition` isEmail
                 `withError`     "Invalid email address:"
```

Similarly, if `correctDate` is a predicate on triples of integers that checks whether this triple represents a legal date (e.g., `correctDate (29,2,1982)` evaluates to `False`), then the WUI specification `wDate` above should be better defined by

```
wDate = wTriple (wSelectInt [1..31]) (wSelectInt [1..12]) wInt
             `withCondition` correctDate
             `withError`     "Illegal date:"
```

Redefining the WUI for persons by

```
wPerson = w4Tuple wString wString wEmail wDate
```

the expression `(wList wPerson)` denotes a WUI specification for lists of persons that checks for valid inputs and provides the error messages shown in Fig. 1.

4 Combining WUIs and JavaScript

As mentioned in Section 1, we intend to exploit the existence of JavaScript interpreters in current web browsers to increase the functionality of WUIs. In particular, we want to transmit a JavaScript program, together with the HTML form implementing a WUI, that implements the validation of user inputs in the HTML form. With this approach, invalid inputs are detected by the web browser on the client side which provides instantaneous feedback to the user and reduces the number of client/server interactions. Note that it is not our intention to shift computations from the server side to the client side in order to reduce the load of the web server: since a web application should never trust user inputs received from a client (see Section 1), the validation of inputs by the web application is mandatory. This design decision has a number of advantages:

- It is not necessary to check all input conditions in a WUI on the client side.
- If the JavaScript program running on the client cannot compute a definite result, e.g., due to resource limitations, it causes no problem for the web application since the input is always validated on the server side.
- The same is true if JavaScript is disabled in the client's browser (e.g., due to security reasons). In this case, the web forms can still be used (in contrast to approaches that rely on JavaScript like PowerForms [1]). The only difference is that input errors are shown *after* the form has been submitted to the web server which sends back a new form with error-annotated input fields (identical to the example in Fig. 1).

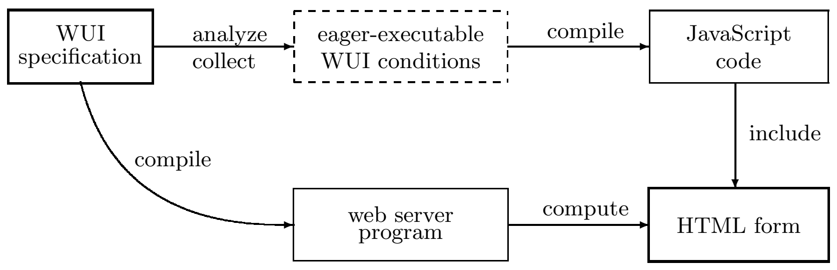


Fig. 2. Structure of the WUI/JavaScript implementation

The implementation of our framework is visualized in Fig. 2. The complete implementation is available with the current distribution of PAKCS [7]. The compilation of a web page containing WUIs is performed by the following steps:

1. The source program is analyzed and conditions in WUIs that are "easy to translate" into JavaScript are marked. These conditions are characterized as "eager executable", i.e., it is ensured that the eager (i.e., innermost) and deterministic rewriting of these conditions produce the same result as in Curry.
2. The eager-executable conditions are collected together with the functions on which they depend (which might be distributed in various imported modules) into a single Curry program.
3. This Curry program is translated into a JavaScript program. Thanks to the fact that this program can be executed in an eager and deterministic manner, the translation into JavaScript is straightforward (Curry functions are mapped into JavaScript functions, data constructors are mapped into arrays, pattern matching is mapped into switch statements etc). In order to reduce the size of the generated JavaScript programs (that is transmitted to the client together with the HTML code), a couple of code optimizations are applied by the compiler. The details of this compilation process and the subsequent optimizations can be found in [6].
4. The original Curry program is compiled with a standard Curry compiler and installed on the web server.
5. If a client demands a web page containing the WUI, the Curry program computes the corresponding HTML form and sends it to the client together with the generated JavaScript program.

Note that the implementation of WUIs, as described in [4], must also consider the integration of JavaScript code in the HTML form. For this purpose, the individual functions to generate HTML code from WUI specifications (they are implicitly contained in the WUI specifications but not directly accessible) generate also the calls to the JavaScript code for eager-executable conditions. These calls are attached to input fields if possible (in case of text fields with `onblur` events) and also collected for the complete WUI and attached to the submit

Program	Curry		JS		JSO	
	lines	bytes	lines	bytes	lines	bytes
posSum	1	21	19	278	3	63
isEmail	27	654	183	3050	77	1598
person	71	1624	404	6546	126	2777
exam	102	2333	629	10072	211	4613

Table 1. Code size of some programs

button. The check of the complete WUI follows an innermost strategy in case of hierarchical data structures (like list of persons containing dates): first, the basic input parts are checked and, if they do not contain an error, the parts constructed from these parts are checked. This strategy is reasonable since it avoids superfluous error messages related to global properties if the individual parts contain input errors. Furthermore, the possible error messages must also be included in the generated HTML code. Since they should be only visible when an input error is detected by the JavaScript code corresponding to the WUI conditions, the error messages are initially invisible and they are made visible, if necessary, by the JavaScript code. This is implemented by the use of different styles for the error elements that is changed by the JavaScript code (see Section 2).

In order to provide an impression of the size of the generated JavaScript code, Table 1 contains the results of compiling some example programs from Curry into JavaScript. The columns contain the sizes of the source Curry program (including all dependent functions but without comments), the generated JavaScript code without optimization (JS), and the generated JavaScript code including optimizations to reduce the code size (JSO). For each class of programs, the number of code lines and the code size in bytes is shown. The first three programs are WUI conditions mentioned in this paper, and program `exam` consists of the conditions of a web-based examination management tool. The difference between the entries in the columns JS and JSO clearly shows that the code optimizations are important and effective.

5 Conclusion

We have proposed a new framework to construct web-based user interfaces in a declarative way that is combined with features of JavaScript in order to exploit existing technologies without efforts for the application programmer. The construction of WUIs is type-oriented, i.e., the programmer selects basic WUI components and combine them with specific combinators in order to obtain a WUI that can be applied to manipulate the data of the application domain. An important feature of WUIs is the possibility to include computable conditions on input data. Since these conditions are checked before the data is transferred to the application program, the application must only specify such conditions but need not check their validity or implement the necessary interactions with the user to correct wrong inputs.

References

1. C. Brabrand, A. Møller, M. Ricky, and M.I. Schwartzbach. PowerForms: Declarative Client-side Form Field Validation. *World Wide Web Journal*, Vol. 3, No. 4, pp. 205–214, 2000.
2. D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 5th edition edition, 2006.
3. 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.
4. M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pp. 27–38. ACM Press, 2006.
5. M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.
6. M. Hanus. Putting Declarative Programming into the Web: Translating Curry to JavaScript. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 155–166. ACM Press, 2007.
7. M. Hanus, S. Antoy, B. Braßel, 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/`, 2007.
8. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.
9. S.H. Huseby. *Innocent Code: A Security Wake-Up Call for Web Programmers*. Wiley, 2003.
10. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

Typebasierte Analyse von JavaScript

Phillip Heidegger

Arbeitsbereich Programmiersprachen
Institut für Informatik
Universität Freiburg

Der AJAX Trend sorgt für eine starke Verbreitung von JavaScript und unterstreicht die Bedeutung der Sprache. Um die Zuverlässigkeit der entwickelten Software sicherzustellen und ein professioneller Einsatz von JavaScript zu ermöglichen, werden Analysewerkzeuge benötigt, wie diese für viele andere Programmiersprachen erhältlich sind.

Es gibt viele Ansätze, JavaScript sicherer zu machen. Eine Arbeit von Yu u.a. schlägt z.B. eine dynamische Methode vor [5]. Sie definiert eine Regelmenge, die JavaScript Quellcode dynamisch verändert. Dies ermöglicht dem Besucher einer Webseite, sich eine für ihn angepasste Regelmenge zu erstellen, die festlegt, welche Aktionen der JavaScript Quellcode ausführen darf. Bei einer solchen Lösung wird dem dynamischen Charakter von JavaScript Rechnung getragen und es werden Funktionen wie `eval`, `Document.write` u.a. unterstützt. Allerdings bietet eine solche Lösung für JavaScript Sicherheitsprobleme einem Programmierer keine Hilfestellung, Fehler in seinem Programm zu finden. Der Adressat des von Yu u.a. vorgestellten Ansatzes ist der Benutzer des Internetbrowsers.

Ist aber das Ziel, für einen JavaScript Programmierer Analysemethoden zu entwickeln, die ihm helfen seine Programme vor der Auslieferung an den Kunden auf Fehler zu analysieren, erfordert dies eine statische Analyse des Quellcodes. Als Beispiel sei hier ein Onlineportal eine Internetbank genannt. Dynamische Methoden würden erst nach Auslieferung oder bestenfalls beim Testen zu Ergebnissen führen. Bei einem Internetportal im sicherheitskritischen Umfeld ist dies nicht ausreichend.

Für statische Analysen muss angenommen werden, dass dem Programmierer der vollständige Quellcode seiner Anwendung statisch zur Verfügung steht. Dies bedeutet z.B., dass die Funktion `eval` mit statischen Analysen nicht zu behandeln ist.

Wie alle statischen Methoden, die nicht triviale Eigenschaften über Programmen berechnen, wird eine statische Analysen für JavaScript Näherungen durchführen müssen. Hierbei gibt es zwei Möglichkeiten. Lindahl u. Sagonas stellen in ihrem Artikel einen Ansatz mit Successtypen vor [3]. Hier werden nur Programme zurückgewiesen, bei denen garantiert wird, dass an einer Stelle des Codes ein Fehler besteht. Falls nicht sicher ist, ob der Code korrekt ausgeführt werden kann, wird bei Successtypen das Programm nicht vom Typsystem zurückgewiesen.

Die zweite Möglichkeit besteht in einer konservativen Analyse des Quellcodes, der nur korrekte Programme bzgl. eines Typsystems als gültig einordnet. Für gültige Programme ist eine gewisse Menge Eigenschaften garantiert. So stellt das Typsystem von Java z.B. sicher, dass bei einer Multiplikation keine Strings mit Zahlen multipliziert werden [1].

In Vortrag wird die zweite Methodik gewählt und eine korrekte Analyse für die Scriptsprache JavaScript kurz vorgestellt. Würde die Eigenschaft, keine Zahlen mit Strings multiplizieren zu können, auf ein Typsystem für JavaScript übertragen, so würde der JavaScript Ausdruck `x = "2" * 4;` vom Typsystem als ungültig abgelehnt. Das Multiplizieren des Strings `"2"` und der Zahl 4 wäre unzulässig. In JavaScript werden Werte aber kontextabhängig in passende Werte anderen Typs konvertiert und der Ausdruck liefert die Zahl 8. Ein Typsystem, das die Multiplikation von Strings und Floats generell ablehnt, würde somit Programme zurückweisen, die ein korrektes Laufzeitverhalten besitzen. Dies wird sich im Allgemeinen nicht komplett vermeiden lassen, wenn man auf Korrektheit Wert legt, aber es muss zumindest sichergestellt werden, dass die einfachen Konvertierungen vom Typsystem behandelt werden können. Der im Vortrag vorgestellte Ansatz basiert stark auf der Arbeit ,,Towards a Type System for Analysing JavaScript Programs“ von Peter Thiemann [4].

Vor der Vorstellung des Typsystems wird JavaScript anhand einiger Beispiele dargestellt. Hierdurch soll ein Gefühl für die Sprache vermittelt werden, und auf einige Fehlerquellen hingewiesen werden, die für einen Programmierer schwer zu erkennen sind.

Ein kurzes Beispiel für einen Fehler in JavaScript Programmen, der sehr schwer zu finden ist, und der auf dem Konvertierungsverhalten von JavaScript basiert, wird hier kurz erwähnt.

```
var y = 0;
y.x = "Hallo";
alert(y.x);
```

Der angegebene Programmausschnitt erstellt eine Variable `y`, und weist dieser den Floatwert `0.0` zu. Durch die automatische Konvertierung wird in der nächsten Zeile einem Objekt, das durch Konvertierung der Zahl in ein Objekt entstanden ist, die Eigenschaft `x` gegeben, und diese auf den Wert `"Hallo"` gesetzt. Leider wird dieser Wert in der dritten Zeile nicht ausgegeben, denn das Objekt wurde nicht an die Variable `y` gebunden. Aus diesem Grund wird in Zeile drei nochmals ein zum Floatwert 0.0 passendes, anderes Objekt erstellt. Dies besitzt keine Eigenschaft `x`. Somit erscheint `undefined` auf dem Bildschirm (Details siehe [4] oder [2]).

An dem Beispiel sieht man eine der zentralen Herausforderungen, denen sich ein Typsystem für JavaScript stellen muss, neben den Konvertierungen von Werten. Objekte von JavaScript haben die Fähigkeit, dynamisch zur Laufzeit Ihre Eigenschaften, sowohl den Wert auch auch die Existenz der Eigenschaften, zu verändern.

Es folgt eine kurze Vorstellung des Typsystems aus meiner Diplomarbeit [2]. Das Typsystem beinhaltet Union- und Intersectiontyps, behandelt die automatischen Konvertierungen von Werten in JavaScript, und kann mit den dynamischen Objekte und der Fehlerbehandlung vom JavaScript umgehen. Es stellt z.B. sicher, dass Fehler, wie im Programmausschnitt dargestellt, nicht möglich sind, aber es akzeptiert den Ausdruck `"2" * 4`.

Ein paar Beispiele gültiger JavaScript Programme, die das vorhandene Typsystem noch zurückweist, werden im Vortrag vorgestellt und bildet eine Überleitung zu der Frage, wie das Typsystem geeignet erweitert werden kann, um auch mit den in diesen Beispielen angesprochenen Problemen umgehen zu können. Wie und ob dies möglich ist, ist noch unklar. Es werden ein paar mögliche Ansätze diskutiert. Das vorhandene Typsystem erlaubt es bis jetzt lediglich, Constraints zu generieren. Lösungstrategien dieser Constraints stellt ein weiteres Diskussionsthema dar.

Literatur

1. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series).* Addison-Wesley Professional, July 2005.
2. Phillip Heidegger. Typbasierte Werkzeuge für Fehlersuche und Wartung von Java-Script Programmen, 2007. Deutschland, Universität Freiburg, Diplomarbeit.
3. Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.
4. Peter Thiemann. Towards a type system for analyzing javascript programs. In *European Symposium On Programming*, pages S. 408 – 422, 2005.
5. Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. 2007.

Akton-Algebra: Programmierung diskreter physikalischer Systeme

Hermann von Issendorff
Institut für Netzwerkprogrammierung
Hauptstr. 40, D-21745 Hemmoor

1. Einführung

Zwischen Software und Hardware besteht bisher ein Bruch. Räumliche Strukturen können bisher nicht in Form eines schrittweise ausführbaren Programms, d.h. konstruktiv beschrieben werden. Mit anderen Worten: Es gibt bisher keine Programmiersprache mit Raumzeitsemantik. Aus diesem Grunde war es bisher auch nicht möglich, die Aufbauordnung diskreter physikalischer Systeme als Folge von Konstruktionsschritten formal zu beschreiben. Ein wichtiges Beispiel diskreter physikalischer Systeme ist das Layout elektronischer Schaltungen.

Hardware-Design-Sprachen, beispielsweise VHDL oder Verilog, simulieren lediglich die Funktionen und Zeitverhalten einer Schaltung. Für das Layout, d.h. für die Erzeugung der Hardware-Struktur, werden nur die Verbindungen zwischen abhängigen Komponenten übernommen (Paarlisten), die räumliche Nachbarschaft der Komponenten bleibt unberücksichtigt. Ersatzweise muss daher mit sehr aufwändigen stochastischen Verfahren eine neue Layout-Struktur erzeugt werden, die generell suboptimal und viellagig ist.

Akton-Algebra ist eine Programmiersprache, mit der sich diskrete räumliche Strukturen, die dynamisch oder statisch sein können, konstruktiv beschreiben lassen. Akton-Algebra hat damit einen wesentlich allgemeineren und damit gänzlich anderen Ansatz als klassische Programmiersprachen, die auf die Struktur des zentralen Speichers und damit auf das Konzept des sequentiellen Zustandsautomaten ausgerichtet sind. Unter allen möglichen dynamischen oder statischen Strukturen bilden die klassischen Rechner nur eine kleine Klasse, und ebenso die ihr zugeordneten klassischen Programmiersprachen. Eine markante Eigenschaft dieser Programmiersprachen ist, dass alle raumbezogenen strukturellen Merkmale wie Kreuzungen oder Zyklen in Komplexkonstrukten verborgen sind. Kreuzungen sind z.B. in jeder bedingten Verzweigung und in jeder arithmetischen Operation unvermeidlich, Zyklen sind Bestandteil jeder Speicherung und jeder Rekursion. Mit Akton-Algebra dagegen können alle räumlichen Merkmale und alle Funktionen analytisch beschrieben werden und damit auch alle klassischen Programmiersprachen.

Auf abstrakter Ebene beschreibt Akton-Algebra die topologische Struktur gerichteter Knotennetze. Da Akton-Algebra kompositional ist, kann jeder Knoten endlich viele Eingangs- und endlich viele Ausgangskanten haben. Die Menge der Basiselemente der Akton-Algebra gliedert sich zum einen in eine Menge, die die Grundstrukturen gerichteter Knotennetze repräsentieren, zum anderen in zwei weitere

Mengen, die zur Abbildung der dreidimensionaler Strukturen auf die eindimensionale aktonalgebraischen Beschreibung erforderlich sind.

Auf konkreter Ebene können den Knoten Funktionen, Abmessungen oder beides zugeordnet werden. Durch Zuweisung von Funktionen zu den Knoten wird ein Knotennetz zu einem Datenverarbeitungssystem und die aktonalgebraische Beschreibung des Knotennetzes folglich zu einem DV-Programm. Da dieses DV-Programm aber die räumliche Struktur des Datenverarbeitungssystems enthält, unterscheidet es sich wesentlich von klassischen Programmen. Um auf einem klassischen Rechner ablauffähig zu werden, muss es auf dessen sequentielle Verarbeitungsstruktur transformiert werden. Durch Zuweisung von Abmessungen zu den Knoten wird ein Knotennetz zu einem materiellen Objekt, und seine aktonalgebraische Beschreibung folglich zu einem Layout-Programm, d.h. zu einer Bauanleitung für das materielle Objekt. Layout-Sprachen, mit denen die planare oder räumliche Struktur eines materiellen Objekts beschrieben werden kann, gibt es bisher nicht.

2. Eigenschaften und Aufbau der Akton-Algebra

Diskrete reale Systeme bestehen aus einer Menge materieller Komponenten, die statisch oder dynamisch sein können. Wenn die Komponenten dynamisch sind, d.h. wenn sie materielle Objekte produzieren oder Funktionen auswerten, dann haben sie ein zeitliches Verhalten und werden in einer zeitlichen Ordnung aktiviert. Sind die Komponenten statisch, dann kann man ihnen eine Assemblierungsordnung zuweisen, die ebenfalls eine zeitliche Ordnung induziert.

Abstrahiert man ein diskretes reales System von seiner Metrik, dann reduziert sich das System auf ein gerichtetes Knotennetz, dessen Knoten die Funktionen des Systems enthalten. Abstrahiert man von der Funktionalität, dann erhält man ein gerichtetes Netz von Bausteinen, die gewisse räumliche Abmessungen haben. Abstrahiert man von der Metrik und der Funktionalität, dann erhält man ein Knotennetz, das nur noch die Abhängigkeiten der Knoten wiedergibt. In einem solchen gerichteten Netz sind irgendwelche zwei Knoten entweder abhängig oder unabhängig, und jeder Knoten kann mit einer endlichen Zahl von Vorgängerknoten und einer endlichen Zahl von Nachfolgerknoten verbunden sein.

Konkretisiert man das Knotennetz wieder mit seiner ursprünglichen Metrik und Funktionalität, dann erhält man wieder das ursprüngliche System. Das gerichtete Knotennetz stellt damit eine Strukturklasse dar, die allen diskreten realen Systemen eigen ist. Eine Programmiersprache, die gerichtete Knotennetze beschreibt, wäre daher eine gemeinsame Grundlage für alle diskreten realen Systeme. Akton-Algebra hat diese Eigenschaft.

Es ist zu betonen, dass die Abstraktion von Metrik und Funktionalität keine Abstraktion von Raum und Zeit bedeutet. Jeder Knoten eines Knotennetzes hat immer eine endliche, wenn auch unbekannte, räumliche Ausdehnung. Der Durchlauf vom Knoteneingang zum Knotenausgang hat also immer eine endliche, wenn auch unbekannte, zeitliche Dauer. Das bedeutet, dass jede Aktion, die auf einem Netzknoten durchgeführt wird, ein diskreter Schritt in Raum und Zeit ist. Für diesen Schritt, gleich ob er eine konkrete Komponente oder einen abstrakten Netzknoten bezeichnet, wurde die Bezeichnung "Akton" gewählt.

Das Knotennetz eines realen Systems hat generell eine dreidimensionale Struktur. Ein Programm dagegen besteht aus einer geordneten Folge von Symbolen und hat damit eine eindimensionale Struktur. In diesem Abschnitt wird gezeigt, dass ein gerichtetes dreidimensionales Knotennetz eindeutig und vollständig auf die eindimensionale Beschreibung eines Programmtextes abgebildet werden kann.

Die Klasse der Knotennetze, mit der wir uns hier befassen, ist gerichtet und hat immer zumindest einen Eingangs- und zumindest einen Ausgangsknoten. Jeder Knoten eines Knotennetzes hat eine Eingangs- und eine Ausgangsschnittstellen. Die Schnittstellen können zwei verschiedene Elemente enthalten, die mit *pin* und *gap* bezeichnet werden. Ein *pin* kennzeichnet die Verbindung zu einem anderen Knoten; ein *gap* kennzeichnet eine Leerstelle. Jede Schnittstelle enthält mindestens ein *pin*- oder ein *gap*-Element.

Eine universelle Eigenschaft aller gerichteten Knotennetze ist, dass sie mit 4 fundamentalen Knotensorten aufgebaut werden können. Die erste, *Head* genannte Sorte, hat keine Vorgängerknoten, aber einen Nachfolgerknoten. Der Input von Head enthält daher nur ein *gap*. Die zweite, *Tail* genannte Sorte, hat keinen Nachfolger aber einen Vorgänger. Der Output von Tail enthält daher nur ein *gap*. Die dritte, *Body* genannte Sorte, hat einen Vorgänger und einen Nachfolger, einen Vorgänger und zwei Nachfolger oder zwei Vorgänger und einen Nachfolger, d.h. mindestens ein *pin* im Input und im Output. Die letzte, *CS* (*Closed System*) genannte Sorte, hat weder Vorgänger noch Nachfolger, d.h. nur *gaps* im Input und im Output.

$$\begin{tikzcd}
 & \mathit{Akton} \arrow[d] & \\
 & \{\mathit{Head, Body, Tail, CS}\} \arrow[dl] \arrow[d] \arrow[dr] & \\
\{\mathit{Entry, Up, Set}\} & \{\mathit{Fork, Join, Link}\} & \{\mathit{Exit, Down, Off}\}
\end{tikzcd}$$

Bild 1: Sortenhierarchie der Strukturknoten.

Die vier Knotensorten *Head*, *Tail*, *Body* und *CS* bilden das fundamentale Vokabular der Akton-Algebra. Sie sind die erste Ebene einer Sortenhierarchie (Bild 1), an deren Spitze die All-Sorte *Akton* steht.

Die Knotensorten nachfolgender Ebenen haben zusätzliche Eigenschaften. Als Untersorten von *Body* lassen sich unmittelbar die drei Sorten *Fork*, *Join* und *Link* einführen. Die *Fork*-Sorten haben einen Vorgänger und zwei Nachfolger, die *Join*-Sorten zwei Vorgänger und einen Nachfolger und *Link* einen Vorgänger und einen Nachfolger. Sie sind die elementaren Knoten mit den Eigenschaften von *Body*. Ebenso lässt sich zu *Head* die Untersorte *Entry* und zu *Tail* die Untersorte *Exit* einführen, die die Eigenschaften eines elementaren Systemeingangs bzw. elementaren Systemausgangs haben. Zwei weitere Untersorten von *Head* und *Tail* sind für die Abbildung gerichteter räumlicher Knotennetze auf die eine Dimension eines Programmtextes erforderlich.

Knotennetze haben eine Struktur, die wegen der Abstraktion von der Metrik nur noch topologisch ist. Topologische Strukturen erhalten die Nachbarschaftsbeziehungen der Knoten, wenn das Knotennetz in irgendeiner Weise stetig deformiert wird. Als Deformation sind auch Schnitte im Netz zugelassen, sofern die Zuordnung der Schnittenden gewährleistet ist. Macht man die Deformation auf die gleiche Weise rückgängig, dann erhält man wieder die Originalstruktur. Im Folgenden wird diese Erhaltungseigenschaft angewendet, um die räumliche Struktur gerichteter Knotennetzen bijektiv auf die eine Dimension eines Programmtextes abzubilden.

Zur Beschreibung der topologischen Raumstruktur eines Knotennetzes ist ein topologisches Referenzsystem erforderlich. Wegen der fehlenden Metrik kann dieses nur relational sein. Ein dreidimensionales relationales Referenzsystem lässt sich durch die Annahme eines Beobachters definieren, der zwischen links und rechts, oben und unten, sowie vorne und hinten unterscheiden kann. Unter diesen drei orthogonalen Achsen muss eine ausgewählt werden, der die Richtung des Knotennetzes aufgeprägt wird. Diese Achse erhält dadurch den Charakter einer Lese- oder Ausführungsrichtung. Dem üblichen westlichen Lesestandard folgend wählen wir hierfür die Richtung von links nach rechts. Da jeder Knoten eine Aktion repräsentiert und jede Aktion eine Dauer hat, induziert diese Festlegung auch eine Zeitrichtung; "links" kann daher auch als "früher" und "rechts" als "später" interpretiert werden.

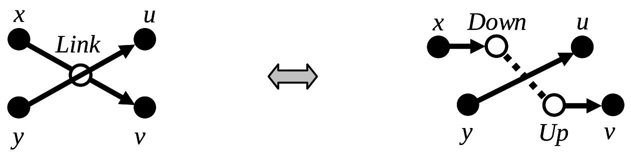


Bild 2: Planarisierung, d.h.. Beseitigung von Kreuzungen, durch Aufschneiden und Einführen eines *Down/Off*- Paars.

Die Abbildung des Knotennetzes auf den Programmtext erfolgt in mehreren Schritten:

1. Das Knotennetz wird so orientiert, dass alle Systemeingänge links und alle Systemausgänge rechts liegen.

2. Das Knotennetz wird auf eine Ebene projiziert, die durch links/rechts und oben/unten aufgespannt ist. Dabei werden generell Verbindungskreuzungen entstehen, die jeweils durch die Ersetzung der untenliegenden Knotenverbindung durch ein mit *Down/Up* bezeichneten Sortenpaars aufgelöst werden (Bild 2). *Down* ist Untersorte von *Tail* und *Up* ist Untersorte von *Head*.

3. Das resultierende Knotennetz ist planar, kann aber noch nichtorientierbare Teilstrukturen, d.h. Zyklen oder Traversen enthalten. Diese Strukturen werden durch Einführung eines weiteren Sortenpaars, das mit *Off/Set* bezeichnet wird, aufgeschnitten und sodann ebenfalls von links nach rechts

orientiert (Bild 3). *Off* ist Untersorte von *Tail*, und *Set* ist Untersorte von *Head*. Das *Off/Set*-Paar wird auch zum Aufspalten von Knotennetzen verwendet, die keinen Eingang (*Entry*) und keinen Ausgang (*Exit*) haben.

4. Das planare, vollständig ausgerichtete Knotennetz kann nun zu einer gerichteten Sequenz umgeformt werden, die partiell aus Teilsequenzen abhängiger Knoten besteht. Zur Kennzeichnung der Abhängigkeit wird eine "*Next*" genannte und mit dem Symbol ">" bezeichneten links/rechts-Relation eingeführt. In gleicher Weise wird zur Kennzeichnung der Unabhängigkeit von Teilsequenzen eine "*Juxta*" genannte und mit dem Symbol "/" bezeichneten oben/unten-Relation eingeführt. Um die Klammerung zu reduzieren wird zudem angenommen, dass *Juxta* stärker bindet als *Next*.

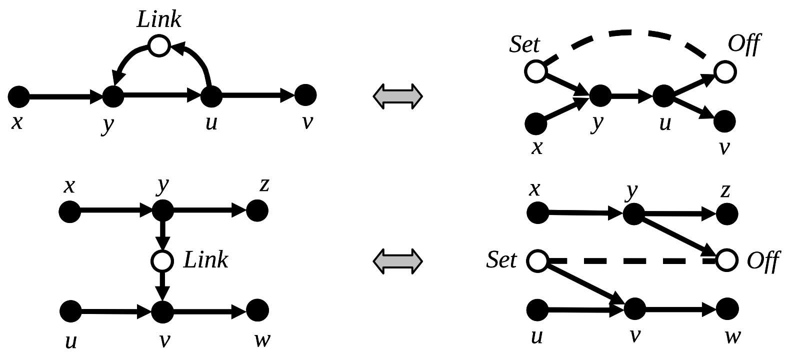


Bild 3: Orientierung von Zyklen und Traversen durch Aufschneiden und Einführen eines *Off/Set*-Paars.

3. Funktionale Beschreibung realer Systeme

Die Sprache der Aktonalgebra, mit der sich, wie gezeigt wurde, die topologische Struktur abstrakter Systemen programmieren lässt, kann nun in einfacher Weise zu Programmiersprachen erweitert werden, die die Funktionalität und die Metrik konkreter realer Systeme beinhalten. Die Akton-Algebra bildet dabei das einheitliche Skelett aller so erzeugbaren Programmiersprachen.

Wir beginnen mit der Einführung von Funktionalität. Sie prägt einem System ein zeitliches Verhalten auf: Die Vorgänge auf dem Knotennetz hängen von den Eingabeparametern ab.

Funktionen sind definitionsgemäss linkstotale und rechtseindeutige Relationen. Als Akton haben sie damit immer mindestens einen Eingangsparameter und genau einen Ausgangsparameter, d.h. eine *Link*- oder eine *Join*-Struktur. Binäre Funktionen sind direkt durch elementare *Link*- und *Join*-Strukturen darstellbar. Höhere Funktionen, d.h. solche mit mehr als zwei Eingangsparametern, lassen sich daraus durch Komposition aufbauen. Die Parameter können entweder als analoger Wert auf einem Pin oder als digitale Werte auf mehreren Pins dargestellt werden. Formal werden Funktionen als Untersorten zu den Sorten *Link* und *Join* definiert.

In diesem Kapitel wird lediglich ein einfaches digitales System als Beispiel behandelt. Die dabei gewonnenen Erkenntnisse können aber problemlos auf komplexe digitale, analoge oder gemischt digital/analoge Systeme angewendet werden.

Bekanntlich kann jedes funktionale digitale System unter Verwendung nur einer der Funktionssorten *Nand* oder *Nor* aufgebaut werden. Unerwähnt bleibt dabei meistens, dass immer auch ein Verbindungselement, d.h. ein Element mit 1-1-Funktion, erforderlich ist. Hier soll es mit *Wire* bezeichnet werden. Statt der Funktionssorten *Nand* oder *Nor* werden im Folgenden nur die Funktionssorte *And* und dazu die inverse Funktionssorte *Not* verwendet. Als Untersorten von *Join* und *Link* werden somit definiert

Join:={And, ..} und *Link:={Wire, Not}.*

Zur Schaltungssimulation muss jeder Funktionssorte eine Wahrheitstabelle zugeordnet werden.

Als Beispiel für ein digitales System dient hier das *SR*-Flipflop, d.h. die einfachste Schaltung eines digitalen Speichers (Bild 4). Die Schaltung mit den klassischen Schaltsymbolen gezeigt. Die Ein- und Ausgangsparameter sind spezifiziert durch:

$Entry:=\{E_a,E_b\}$ mit $out(E_a):= a$, $out(E_b):= b$,

$Exit:=\{X_c,X_d\}$ mit $in(X_c):= c$, $in(X_d):= d$ und

$a,b,c,d \in \{0,1\}$

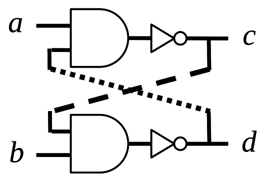


Bild 4: Schaltbild eines *SR*-Flipflops.

Die punktierte Linie stellt eine unterkreuzende Verbindung dar, im Akton-Ausdruck beschrieben durch ein *Down/Up*-Paar. Die gestrichelte Linie stellt eine Verbindung auf der Schaltungsebene dar, im Akton-Ausdruck beschrieben durch ein *Off/Set*-Paar. Die Schaltung SR_{Fkt} wird durch den Akton-Ausdruck

$$SR_{Fkt} :=$$

$$(Wire/Up>And>Not>Fork>Wire/Off)/(Set/Wire>And>Not>Fork>Down/Wire)$$

mit den Schnittstellen

$$in(SR_{Fkt}) = \{pin/pin\}$$
$$out(SR_{Fkt}) = \{pin/pin\}.$$

beschrieben. Der vollständige Ausdruck, der auch die Spezifikation der Ein- und Ausgänge des Systems einschliesst, ist damit

$$E_a/E_b > SR_{Fkt} > X_c/X_d.$$

4. Metrische Beschreibung realer Systeme (Layout)

Die Abbildung eines diskreten realen Systems auf ein abstraktes Knotennetz setzt Stetigkeit voraus, was bedeutet, dass die Komponenten stetig verformbar sein müssen. Die gleiche Verformbarkeit muss natürlich gegeben sein, wenn das abstrakte Knotennetz wieder konkretisiert werden soll, d.h. die ursprüngliche Metrik des realen Systems oder eine andere Metrik annehmen soll.

Die Forderung nach stetiger Verformbarkeit lässt sich durch die beiden Annahmen ersetzen, dass alle Komponenten eine konvexe Oberfläche haben und weitere, funktionsneutrale Komponenten eingefügt werden dürfen.

Ein einfaches Verfahren zur Konkretisierung lässt sich durch die Einführung eines einheitlichen quadratischen bzw. kubischen Rasters erreichen. Jede Systemkomponente ist dann in Vielfachen des Rastermasses beschreibbar.

Als Beispiel für die metrische Beschreibung wird wieder das *SR*-Flipflop verwendet. Zur Unterscheidung von der funktionalen Beschreibung verwenden wir einfache Grossbuchstaben für die Schaltungskomponenten. In Anlehnung an die realen Gegebenheiten wird festgelegt, dass *And*-Komponenten *A* in Breite und Länge die Abmessungen 2×3, Inverter *N* und *Down/Up*- Komponenten *V* (Via) die Abmessungen 2×2, und die Strukturkomponenten von *Fork F* und *Wire W* die Abmessungen 1×1 haben.

Soweit sich durch diese Festlegungen die Grösse der Schnittstellen ändert, werden diese durch *gap*-Elemente erweitert. Dies ist bei den Komponenten *A*, *N* und *V* der Fall. Da die *gap*-Elemente an beliebiger Stelle in die Schnittstellen eingefügt werden können, ergeben sich weitere Untersorten. Im Beispiel werden nur die Schnittstellen definiert, die tatsächlich Verwendung finden. Bezüglich *A* ist das $out(A) := gap/pin$ und bezüglich *N* $in(N) := gap/pin$ und $out(N) := gap/pin$. Bezüglich *V* ist zwischen V_u, $out(V_u) := pin/gap$ und V_d, $in(V_d) := pin/gap$ zu unterscheiden.

Für die Sorten *Fork* und *Wire* werden je drei Untersorten eingeführt, die unterschiedliche Positionen des Outputs relativ zum Input haben. Bei *Wire*-Komponenten *W* kann der Output relativ zum Input gegenüber, links oder rechts liegen, was durch die Indizes *s* (straight), *l* (left) und *r* (right) gekennzeichnet werden soll. Bei *Fork*-Komponenten *F* geschieht das durch ein Indexpaar, mit dem die relativen Positionen der beiden identischen Output-Teile angegeben werden. Als metrische Untersorten von *Fork* und *Wire* werden somit definiert:

$$F := \{F_{ls}, F_{sr}, F_{lr}\},\ W := \{W_s, W_l, W_r\}.$$

Zur Verkürzung der Schreibweise wird zusätzlich eingeführt, dass die Länge von Ketten einer Aktonsorte *x* durch einen präfixen Faktor *n* angegeben werden. Der Faktor ist definiert durch:

$$(n+1)x = nx{>}x,\ n>0,\ 1x = x,\ n \in \mathbf{N},\ x \in A^{+}.$$

Mit diesen Vorgaben kann das Layout des SR-Flipflops unter Anwendung der Termersetzungsregeln von Tabelle 1, insbesondere der *Link*-Regeln, in vielfacher Weise verändert werden. Ein besonders kompaktes Layout des SR-Flipflops ist in Bild 5 dargestellt und in dem Aktonausdruck SR_{Lay} beschrieben:

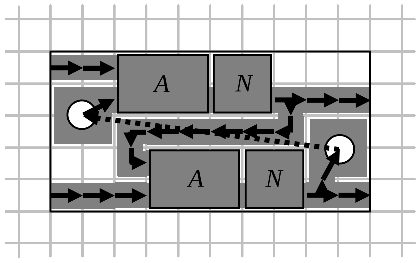


Bild 5: Layout eines *SR*-Flipflops

$$SR_{Lay} = (2W_s/V_u>A>N>F_{sr}>2W_s/((W_r>4W_s>2W_l\,)/3W_s>A>N>F_{sl}>V_d/W_s)$$

Das konkrete *SR*-Flipflop SR_{Lay} hat die Schnittstellen

$$in(SR_{Lay}) = \{pin/gap/gap/gap/pin\}$$
$$out(SR_{Lay}) = \{gap/pin/gap/gap/pin\}.$$

5. Zusammenfassung und Anmerkungen

In dieser Beschreibung wird eine Akton-Algebra genannte Programmiersprache vorgestellt, mit der sich nicht nur die Verarbeitung von Daten bzw. die Auswertung von Funktionen programmieren lässt, sondern auch die räumliche Struktur des Systems, auf dem die Verarbeitung stattfindet. Bisher gibt es keine andere Programmiersprache, die das leistet.

Beides, die Programmierung der Funktionen und die Programmierung der räumlichen Struktur bzw. Layouts wurde am Beispiel des *SR*-Flipflops gezeigt.

Zu ergänzen ist, dass Akton-Algebra nicht etwa auf die Schaltungsebene beschränkt ist, sondern mittels Komposition auf jeder höheren Ebene angewendet werden kann. Beispielsweise können in der gleichen Weise auch Rechnernetze beschrieben werden, d.h. Netze, in denen die Rechner die Komponenten sind.

Synthesizing Design Models from Scenarios by Learning

Benedikt Bollig[1], Joost-Pieter Katoen[2], Carsten Kern[2], and Martin Leucker[3]

[1] LSV, CNRS UMR 8643 & ENS de Cachan, France
[2] Software Modeling and Verification Group, RWTH Aachen University, Germany
[3] Institut für Informatik, TU München, Germany

Extended Abstract:

The elicitation of requirements is the main initial phase in the typical software engineering development cycle. A plethora of elicitation techniques for requirement engineering exist. Popular requirement engineering methods, such as the Inquiry Cycle and CREWS [NE00], exploit use cases and scenarios to specify the system's requirements. Sequence diagrams are also at the heart of the UML. A scenario is a partial fragment of the system's behavior, describing the system components, their message exchange and concurrency. Their intuitive yet formal nature has resulted in a broad acceptance. Scenarios can be positive or negative, indicating a desired or unwanted system behavior, respectively. Different scenarios together form a more complete description of the system behavior.

The following design phase in software engineering is a major challenge as it is concerned with a paradigm shift between the *requirement* specification—a partial, overlapping and possibly inconsistent description of the system's behavior—and a conforming *design model*, a complete behavioral description of the system (at a high level of abstraction). During the synthesis of design models, usually automata-based models that are focused on intra-agent communication, conflicting requirements will be detected and need to be resolved. Typical resulting changes to requirements specifications include adding or deleting scenarios, and fixing errors that are found by a thorough analysis (e.g., model checking) of the design model. Obtaining a complete and consistent set of requirements together with a related design model is thus a highly iterative process.

We propose a novel technique that is aimed to be an important stepping stone towards bridging the gap between scenario-based requirement specifications and design models. The novel aspect of our approach is to exploit *learning* algorithms for the synthesis of *distributed* design models from scenario-based specifications. Since message-passing automata (MPA, for short) [BZ83] are a commonly used model to realize the behavior as described by scenarios, we adopt MPA as design model. We present a procedure that interactively infers an MPA from a given set of positive and negative scenarios of the system's behavior provided as message sequence charts (MSCs) (cf. figure 1). This is achieved by generalizing Angluin's learning algorithm for deterministic finite-state automata (DFA) [Ang87] towards specific classes of bounded MPA, i.e., MPA that can be used to realize MSCs with channels of finite capacity. Moreover we investigated classes of regular MSC languages and corresponding MPA which can (not) be learned.

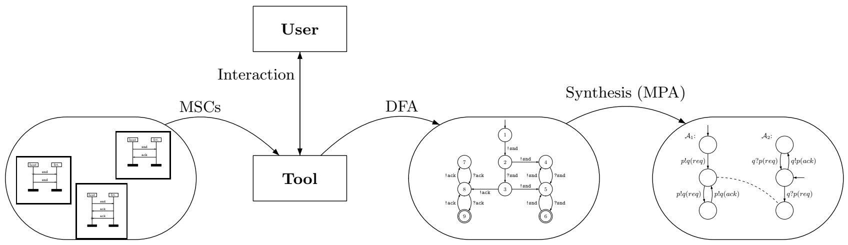


Fig. 1. Transforming MSCs into MPA

The full version of this work can be found in [BKKL07]. We also developed a tool named `Smyle` (Synthesizing models by learning from examples) which supports our approach. A version of `Smyle` can be freely downloaded from the tool's homepage at: http://smyle.in.tum.de.

References

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[BKKL07] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Replaying play in and play out: Synthesis of design models from scenarios by learning. In Orna Grumberg and Michael Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science*, pages 435–450, Braga, Portugal, March 2007. Springer.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. of the ACM*, 30(2):323–342, 1983.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *ICSE 2000*, pages 35–46. ACM, 2000.

Google's MapReduce Programming Model — Revisited[⋆]

Ralf Lämmel

Microsoft Corp., Data Programmability Team, Redmond, USA

Abstract. Google's MapReduce programming model serves for processing large data sets in a massively parallel manner. We deliver the first rigorous description of the model including its advancement as Google's domain-specific language Sawzall. To this end, we reverse-engineer the seminal papers on MapReduce and Sawzall, and we capture our findings as an executable specification. We also identify and resolve some obscurities in the informal presentation given in the seminal papers. We use typed functional programming (specifically Haskell) as a tool for design recovery and executable specification. Our development comprises three components: (i) the basic program skeleton that underlies MapReduce computations; (ii) the opportunities for parallelism in executing MapReduce computations; (iii) the fundamental characteristics of Sawzall's aggregators as an advancement of the MapReduce approach. Our development does not formalize the more implementational aspects of an actual, distributed execution of MapReduce computations.

Keywords: Data processing; Parallel programming; Distributed programming; Software design; Executable specification; Typed functional programming; MapReduce; Sawzall; Map; Reduce; List homomorphism; Haskell.

[⋆] The full paper appears in the *Science of Computer Programming* journal.

Lifting Curry's Monomorphism Restriction

Wolfgang Lux

University of Münster
wlux@uni-muenster.de

Abstract. The multi-paradigm declarative language Curry combines features from modern functional, logic, and functional-logic programming languages. Curry's syntax is similar to that of the functional language Haskell and, also like Haskell, Curry's type system is based on the Hindley-Milner typing discipline, which allows automatic type inference without declaring types explicitly. However, Curry's type system suffers from a monomorphism restriction that requires the types of all local variables to be monomorphic, whereas in the Hindley-Milner type system, the types of let-bound variables can be generalized polymorphically.
Curry's monomorphism restriction is a consequence of the presence of unbound logical variables whose type cannot be generalized. Yet, restricting all let-bound variables to monomorphic types unnecessarily causes rejection of some perfectly sound programs and also may be an obstacle to compiling Haskell programs with a Curry compiler. This paper shows how Curry's monomorphism restriction can be lifted for a broad class of definitions by means of a purely syntactic analysis. This analysis is related to the value restriction employed by ML-like languages in order to ensure type soundness of programs in the presence of mutable variables.

1 Introduction

The multi-paradigm declarative language Curry [Han06] aims at amalgamating the most important features from functional, logic, and functional-logic programming languages. Curry's syntax and semantics for ground expressions are similar to that of the functional language Haskell [Pey03]. In particular, Curry supports the introduction of local functions and variables inside let-declarations and with where clauses. Local declarations are also used in order to introduce logical variables, which in contrast to many other logic and functional-logic languages must be declared explicitly.

Like other modern declarative languages, Curry's type system is based on the Hindley-Milner typing discipline [Hin69,DM82], which combines parametric polymorphism with automatic type inference. As an extension to the basic Hindley-Milner type system, Curry supports polymorphic recursion for explicitly typed function declarations. On the other hand, Curry's type system also suffers a serious restriction: it does not support polymorphic generalization of the types of local variables declared in let expressions and where clauses. For instance, all definitions in Fig. 1 are not valid according to Curry's typing rules because the

```
f1 = (1:nil, 'a':nil) where nil = []
f2 = [z (), z False] where z = const 0
f3 = last (last "Curry")
  where last = (λxs → let y,ys free in xs =:= ys ++ [y] &> y)
```

Fig. 1. Functions using polymorphic local variables

local variables `nil`, `z`, and `last` are used at different types in the bodies of their respective functions.

The monomorphism restriction for let-bound variables in Curry is motivated by the fact that logical variables must have a monomorphic type. If polymorphic types were allowed for logical variables, it would be possible to define a polymorphic cast function with type $\forall\alpha\beta.\alpha \to \beta$

```
cast x | x=:=y = y where y free
```

This typing is unsound because it would allow casting the type of an expression into any other type[1]. Furthermore, logical variables may appear in the result of a function, e.g.,

```
unknown = x where x free
```

which is defined in the Curry Prelude. Note that a different logical variable is returned for each use of this function and therefore the constraint

```
(unknown,unknown) =:= (1,[])
```

is satisfiable.

Nevertheless, all definitions from Fig. 1 are perfectly sound. Furthermore, the definition of `f3` would be accepted if the local variable definition were replaced by a function definition

```
last xs = let y,ys free in xs =:= ys ++ [y] &> y
```

Thus, Curry's monomorphism restriction causes some perfectly sound definitions to be rejected and limits the usability of the so-called point-free programming style. In this paper we propose a modification to Curry's typing rules that allows inferring polymorphic types for let-bound variables when the bound expression cannot contain any logical variables.

The rest of this paper is structured as follows: In the following section we will first review Curry's present typing rules. In Sect. 3 we will identify a class of

[1] In fact, such a polymorphic cast function can be implemented in current Curry implementations by abusing an extension that allows evaluating equality constraints between partial applications. For instance, the type $\forall\alpha\beta.\alpha \to \beta$ is inferred for the function

```
cast' x | ign x =:= ign y = y where y free; ign x y = x
```

expressions for which it is easy to compute groundness information. Based on this class, we will propose an extension of the typing rules that allows polymorphic generalization for variables bound to ground expressions. The fourth section presents related work and in Sect. 5 we will consider the relation of Curry's monomorphism restriction to the often criticized monomorphism restriction in Haskell. The final section section then concludes and gives an outlook on future work.

2 Curry's Typing Rules

The typing rules for Curry as presented in the Curry report (cf. Sect. 4.2 of [Han06]) assume that all local declarations of a program except for let-free declarations introducing new logical variables have been lifted into arguments of auxiliary functions. Thus, all variables bound in let expressions become lambda-bound variables in the transformed programs and their types are not generalized according to the typing rules.

Since we are interested in generalizing the types of (some) let-bound variables, we will not perform such lifting. Note that keeping local variable declarations also corresponds with recent approaches to define the operational semantics of functional-logic programs [AHH+02]. In order to simplify the presentation of the typing rules we will consider only a core language whose syntax is given in Fig. 2. We use the notation $\overline{x_n}$ as a abbreviation for the sequence $x_1, \ldots, x_n$.

$$\begin{array}{rcll}
P & ::= & d_1; \ldots ; d_n & \text{Programs} \\
d & ::= & \mathtt{data}\ T\,\overline{x_n} = C_1\overline{\tau_{1n_1}} \mid \cdots \mid C_m\overline{\tau_{mn_m}} & \text{Declarations} \\
 & \mid & f\ x_1 \ldots x_n = e & \\
\tau & ::= & x \mid T\,\overline{\tau_n} \mid \tau_1 \to \tau_2 & \text{Types} \\
e & ::= & x \mid C \mid f & \text{Expressions} \\
 & \mid & e_1\, e_2 & \\
 & \mid & \lambda x.e & \\
 & \mid & \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 & \\
 & \mid & \mathtt{let}\ x\ \mathtt{free}\ \mathtt{in}\ e & \\
 & \mid & \mathtt{case}\ e\ \mathtt{of}\ \{\ t_1 \to e_1; \ldots ; t_n \to e_n\ \} & \\
t & ::= & C\,\overline{x_n} & \textit{Patterns}
\end{array}$$

Fig. 2. Syntax of the core language

Within our language, we distinguish variables (x, x_1, x_2, etc.), functions (f, f_1, f_2, etc.), and constructors (C, C_1, C_2, etc.).

The arity of a constructor ($ar(C)$) is equal to the number of type arguments in its declaration. The arity of a function ($ar(f)$) is determined by the number of arguments in its definition. Constructor applications in patterns must be saturated, i.e., $n = ar(C)$.

$\mathcal{FV}(e)$ denotes the free variables of a (type) expression e.

A type scheme σ is a type expression with universal quantification for some of its type variables, i.e., it has the form $\forall\alpha_1 \ldots \alpha_n.\tau$, where $\{\alpha_1, \ldots \alpha_n\} \subseteq \mathcal{FV}(\tau)$.

The types of all variables, functions, and constructors of a program are collected in a type environment A, which is a mapping from identifiers to type schemes. Fig. 3 shows the typing rules for expressions and patterns in the core language. A function $f\, x_1 \ldots x_n = e$ is well typed with respect to a type environment A if $A(f) = \forall\overline{\alpha_n}.\tau$ with $\{\overline{\alpha_n}\} = \mathcal{FV}(\tau)$ and $\lambda x_1 \ldots \lambda x_n.e :: \tau$ is derivable according to the typing rules. It is straight forward to show that these rules are in fact equivalent to those presented in the Curry report. The auxiliary function

$$\text{(INST)} \quad \begin{array}{c} A[x:\tau] \vdash x :: \tau \\ A[f:\sigma] \vdash f :: \tau \\ A[C:\sigma] \vdash C :: \tau \end{array} \quad \tau = \text{INST}(\sigma)$$

$$\text{(APP)} \quad \frac{A \vdash e_1 :: \tau_1 \to \tau_2 \quad A \vdash e_2 :: \tau_1}{A \vdash e_1\, e_2 :: \tau_2}$$

$$\text{(ABS)} \quad \frac{A[x/\tau_1] \vdash e :: \tau_2}{A \vdash \lambda x.e :: \tau_1 \to \tau_2}$$

$$\text{(LET)} \quad \frac{A \vdash e_1 :: \tau_1 \quad A[x/\tau_1] \vdash e_2 :: \tau_2}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: \tau_2}$$

$$\text{(EXIST)} \quad \frac{A[x/\tau_1] \vdash e :: \tau_2}{A \vdash \texttt{let}\ x\ \texttt{free in}\ e :: \tau_2}$$

$$\text{(CASE)} \quad \frac{A \vdash e :: \tau_1 \quad \overline{A \vdash (t_n \to e_n) :: \tau_1 \to \tau_2}}{A \vdash \texttt{case}\ e\ \texttt{of}\ \{\ t_1 \to e_1; \ldots; t_n \to e_n\ \} :: \tau_2}$$

$$\text{(ALT)} \quad \frac{A[\overline{x_n/\tau_n}] \vdash C\,\overline{x_n} :: \tau \quad A[\overline{x_n/\tau_n}] \vdash e :: \tau'}{A \vdash (C\,\overline{x_n} \to e) :: \tau \to \tau'}$$

Fig. 3. Typing rules for the core language

`INST` instantiates a polymorphic type scheme $\forall\overline{\alpha_n}.\tau$ with fresh type variables: $\text{INST}(\forall\overline{\alpha_n}.\tau) = \tau[\overline{\alpha_n/\beta_n}]$ where $\overline{\beta_n}$ fresh.

As mentioned before, and in contrast to the Hindley-Milner type system, the type of a let-bound variable is not generalized within the body of a let expression in the (LET) rule. Hence, all variables have monomorphic types, which is also reflected in the first case of the (INST) rule.

3 Identifying Ground Expressions

As explained in the introduction, restricting the types of all let-bound variables to monomorphic types is too restrictive. It would be sound to replace the (LET) rule by two rules that enable polymorphic generalization for ground expressions.

$$\text{(LET-M)}\ \frac{A \vdash e_1 :: \tau_1 \quad A[x/\tau_1] \vdash e_2 :: \tau_2}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: \tau_2} \quad \text{if } e_1 \text{ is not ground}$$

$$\text{(LET-P)}\ \frac{A \vdash e_1 :: \tau_1 \quad A[x/\sigma_1] \vdash e_2 :: \tau_2}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: \tau_2} \quad \begin{array}{l}\text{if } e_1 \text{ is ground,}\\ \sigma_1 = \text{GEN}(A, \tau_1)\end{array}$$

In addition, the first case of the (INST) rule has to be generalized in the obvious way. The auxiliary function GEN quantifies the free type variables of a type scheme with respect to a type environment: $\text{GEN}(A, \tau) = \forall \overline{\alpha_n}.\tau$ where $\{\overline{\alpha_n}\} = \mathcal{FV}(\tau) \setminus \mathcal{FV}(A)$.

The problem here is to distinguish ground and non-ground expressions. The information necessary for this distinction could be determined by a groundness analysis. However, we prefer to not making a complex semantic analysis a prerequisite or part of type checking. Another option would be augmenting the type system itself with groundness information. However, this comes with the cost of complicating the type system and also leads to incompatibilities with Haskell.

Still, even without a semantic analysis we can approximate groundness of expressions better than at present in Curry, where effectively every expression is considered non-ground. While it is impossible to determine the logical variables in the result of a function application without a semantic analysis, it is easy to determine the set of logical variables in expressions which are normal forms already. Therefore, we identify a subclass of expressions that we call *non-expansive.* In our core language, a *non-expansive* expression is either

– a variable x,
– an application of a n-ary constructor C to not more than n non-expansive expressions,
– an application of a n-ary function f or lambda abstraction $\lambda x_1 \ldots \lambda x_n.e$ to not more than $n-1$ non-expansive expressions, or
– an expression `let` $x = e_1$ `in` e_2 where e_1 and e_2 are both non-expansive expressions.

An expression is called *expansive* if it is not non-expansive.

It is easy to prove that the logical variables that can appear in the result of reducing a non-expansive expression e must come from the context of e: $\mathcal{LV}(e) \subseteq \mathcal{FV}(e)$ if e is non-expansive. Since the types of the variables in $\mathcal{FV}(e)$ are collected in the type environment A and the free type variables in A are not generalized, it is sound to replace the (LET) rule in Fig. 3 by the new rules (LET-M) and (LET-P) from Fig. 4.

With the new typing rules all functions from Fig. 1 are now well typed. Obviously, minor variations of these functions are still not considered well typed with our simple approach, e.g.

```
f4 = (1:nil, 'a':nil) where nil = id []
```

Here the type of `nil` is not generalized because the expression `id []` is expansive. However, we conjecture that examples like `f4` occur rarely in practice.

$$\text{(LET-M)}\ \frac{A \vdash e_1 :: \tau_1 \quad A[x/\tau_1] \vdash e_2 :: \tau_2}{A \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: \tau_2} \quad \text{if } e_1 \text{ is expansive}$$

$$\text{(LET-P)}\ \frac{A \vdash e_1 :: \tau_1 \quad A[x/\sigma_1] \vdash e_2 :: \tau_2}{A \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: \tau_2} \quad \begin{array}{l}\text{if } e_1 \text{ is non-expansive,}\\ \sigma_1 = \mathrm{GEN}(A, \tau_1)\end{array}$$

Fig. 4. Modified typing rules

4 Related Work

Our approach to identify ground expressions purely syntactically is inspired by the value restriction used in ML-like languages in order to ensure type soundness in the presence of mutable references. Mutable references present a similar problem for type soundness like logical variables in Curry. A mutable reference in Objective Caml has type `'a ref` and supports assignment and dereference operations (denoted by `:=` and prefix `!`, respectively). New references are constructed and initialized by an application of the `ref` constructor. Even though the `ref` type itself is polymorphic, the type of a variable bound to a mutable reference must not be generalized since otherwise it would be possible to define an unsound polymorphic cast function, e.g.,

```
let cast x = let r = ref x in !r
```

In order to prevent this unsoundness, early implementations of ML did not allow generalization of any type variables which appear under a `ref` constructor. However, it became soon clear that this approach was too restrictive as it did preclude the polymorphic use of some useful functions that were using references internally. For instance, in those early ML implementations the type of `cast` would be monomorphic. However, since each application of `cast` uses its own mutable reference and the visibility and lifetime of the reference do not extend beyond the right hand side of `cast`, it is sound to assign type $\forall\alpha.\alpha \to \alpha$ to `cast`.

A first step towards this more general typing was the introduction of imperative type variables in the Tofte discipline [Tof90] which is used in Standard ML 90. Imperative type variables are marked with `*` and must be instantiated to ground types whenever a side effect may occur. Later more refined type systems based either on effect analysis by Talpin and Jouvelot [TJ92] or closure typing by Leroy and Weis [LW91] were introduced. Finally, Wright and Felleisen [WF94] proposed the value restriction, which is now used in ML-like languages and is based on restricting polymorphic generalization to syntactic values. Due to the fact that ML-like languages use an eager evaluation strategy and do not provide logical variables, the class of non-expansive expressions in our proposal is slightly different from syntactic values in ML.

Recently, Garrigue [Gar04] has proposed a relaxed version of the value restriction for ML, where polymorphic generalization is also allowed for all type variables of a type which appear only in covariant positions. As a special rule, the argument position of the `ref` type is always considered contravariant. Obviously, this relaxation does not apply in Curry because otherwise the type of the variable `x` in the expression

```
let x = (let y free in y) in ...
```

would be generalized incorrectly to $\forall\alpha.\alpha$.

The problem of typing let-expressions in functional-logic languages has not been addressed to our knowledge so far. This may, in part, be due to the fact that many functional-logic languages, e.g. BABEL and $\mathcal{TOY}$ do not provide local definitions.

5 Relation to Haskell's Monomorphism Restriction

Haskell also suffers from a monomorphism restriction, which is often criticized. The restriction in Haskell is related to overloading with type classes and is present in order to avoid unexpected losses of efficiency. In particular, Haskell's monomorphism restriction prevents the generalization of all constrained type variables in a definition that is syntactically a value definition, i.e., which is of the form $x = e$. For instance, considering the two superficially functions `sqr1` and `sqr2` in

```
sqr1 x = x * x
sqr2 = λx → x*x
```

the function `sqr1` has type $\forall\alpha.$`Num` $\alpha \Rightarrow \alpha \to \alpha$, whereas the type variable α is not generalized in `sqr2`'s type. Therefore, the expression `(sqr1 (0::Int), sqr1 (0::Double))` is well typed and `(sqr2 (0::Int), sqr2 (0::Double))` is not. It is always possible to overcome this restriction with the help of an explicit type signature. For instance, with a type signature `sqr2 :: Num a ⇒ a → a` the definitions of `sqr1` and `sqr2` become completely interchangeable. The rationale behind Haskell's monomorphism restriction is that the expression bound to a variable is shared and its value evaluated at most once. However, sharing is – necessarily – lost if the value of `sqr2` is supposed to be used at different instances of the `Num` class.[2]

The Haskell monomorphism restriction and the Curry monomorphism restriction are orthogonal. Thus, Haskell's monomorphism restriction would apply in just the same way in an implementation of Curry that is enriched with type classes. On the other hand, we can use our distinction of expansive and non-expansive expression in order to lift Haskell's restriction. Since non-expansive expressions cause no evaluation at runtime at all, the argument that losing sharing can degrade performance does not apply. By applying our modified typing

[2] In a dictionary based implementation of type classes, loss of sharing becomes obvious. Without a type signature, `sqr2` would be using a fixed `Num` dictionary. However, if `sqr2` is supposed to be used polymorphically, it acquires an implicit parameter for the `Num` dictionary and thus becomes a unary function.

rule to overloaded value definitions, we see that the type of `sqr2` can be generalized to `Num a` $\Rightarrow$ `a` $\rightarrow$ `a`.

6 Conclusion

In this paper we have presented a simple scheme that recovers polymorphic generalization for let-bound variables in some cases in the functional-logic language Curry. As a consequence of this generalization, some programs, which previously were rejected by Curry compilers, now become valid Curry programs. This gives the programmer a greater freedom in the use of programming styles and also leads to greater compatibility between Haskell and Curry.

As future work, we plan to formally prove the correctness of our typing scheme. Furthermore, we plan to investigate the possibility of enriching Curry's type system with groundness information so that a larger class of programs will be accepted. Yet, such information should be introduced in such a way that it does not break compatibility with Haskell – at least most of the time.

References

[AHH$^+$02] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. An operational semantics for declarative multi-paradigm languages. *ENTCS*, 70(6), 2002.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. POPL'82*, pages 207–212. MIT Press, 1982.

[Gar04] Jacques Garrigue. Relaxing the value restriction. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Proc. FLOPS 2004*, LNCS 2998, pages 196–213. Springer, 2004.

[Han06] Michael Hanus (ed.). Curry: An integrated functional logic language. (version 0.8.2). `http://www.informatik.uni-kiel.de/~mh/curry/report.html`, 2006.

[Hin69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Proc. POPL'91*, pages 291–302, 1991.

[Pey03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report.* Cambridge University Press, 2003.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *LICS*, pages 162–173, 1992.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Formalization of the Java 5.0 Type System

Martin Plümicke

University of Cooperative Education Stuttgart/Horb
Department of Information Technology
Florianstraße 15
D–72160 Horb
tel. +49-7451-521142
fax. +49-7451-521190
m.pluemicke@ba-horb.de

Abstract. With the introduction of Java 5.0 the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. The term

`Vector<? extends Vector<AbstractList<Integer>>>`

is for example a correct type in Java 5.0.
In this paper we present a formalization of this type system. We define the set of correct Java 5.0 type terms, formally. We give a formal definition of the Java 5.0 subtyping ordering. Finally, we consider the properties of the subtyping ordering, which follow from the introduction of wildcards.

1 Introduction

With the introduction of Java 5.0 [1] the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. For example the term

`Vector<? extends Vector<AbstractList<Integer>>>`

is a correct type in Java 5.0.
Considering all that, it is often rather difficult for a programmer to recognize whether such a complex type is the correct one for a given method or not.
This has caused us to develop a Java 5.0 type inference system which assists the programmer by calculating types automatically [2]. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principle types.
In [1] the Java 5.0 type system is specified. This specification is done in a semi-formal way. Some definitions are rather formal, as the subtyping relation (§4.10) or the capture conversion (§5.1.10). Other definitions are only given informal, as wildcard types. The presentation is sometimes less clearly arranged.
In this paper we present an integrated framework for the Java 5.0 type system. Without loss of generality we restrict the type system to parameterized reference

types with and without wildcards. We do not consider base types (`int`, `boolean`, `float`, ...) and raw types.
The paper is organized as follows. In the second section we give the definition of correct Java 5.0 type terms. In the third section we define the subtyping relation, as an extension of the *extends* relation given by the class declarations. In the fourth section we consider the soundness property of the Java 5.0 type system. Finally, we close with a summary and an outlook.

2 Java 5.0 Simple Types

The base of the types are elements of the set of terms $T_\Theta(\,TV\,)$, which are given as a set of terms over a finite rank alphabet $\Theta = \Theta^{(n)}_{n\in\mathbb{N}}$ of class names and a set of type variables TV. Therefore we denote them as *type terms* instead of types.

Example 1. Let the following Java 5.0 program be given:

```
class A<a> implements I<a> { ...}
class B<a> extends A<a> { ...}
class C<a extends I<b>,b> { ...}
interface I<a> { ...}
interface J<a> { ...}
class D<a extends B<a> & J<b>, b> { ...}
```

The rank alphabet $\Theta = \Theta^{(n)}_{n\in\mathbb{N}}$ is determined by

$$\Theta^{(1)} = \{\,\mathtt{A}, \mathtt{B}, \mathtt{I}, \mathtt{J}\,\} \textit{ and } \Theta^{(2)} = \{\,\mathtt{C}, \mathtt{D}\,\}.$$

For example `A<Integer>`, `A<B<Boolean>>`, and `C<A<Object>,Object>` are type terms.

As the type terms are constructed over the class names, we call the class names in this framework *type constructors*.
If we consider the Java 5.0 program of Example 1 more accurately, we recognize that the bounds of the type parameters `b` in the class `C` and the bounds of the type parameter `a` in the class `D` are not considered. This leads to the problem that type terms like `C<C<a,b>,a>` are in the term set $T_\Theta(\,TV\,)$, although they are not correct in Java 5.0.
The solution of the problem is the extension of the rank alphabet Θ to a type signature, where the arity of the type constructors is indexed by bounded type variables. This leads to a restriction in the type term construction, such that the correct set of type terms is a subset of $T_\Theta(\,TV\,)$. Additionally the set of correct type terms is added by some wildcard constructions. We call the set of correct types *set of simple types* $\mathsf{SType}_{TS}(\,BTV\,)$ (Def. 4).
Unfortunately, the definitions of the type signature (Def. 2), the simple types (Def. 4), and the subtyping ordering (Def. 5) are mutually dependent. This is caused by the fact, that the restriction of the set of simple types is defined

by bounded type parameters, whose bounds are also simple types. This means that, for some definitions, we must assume a given set of simple types, without knowing, how the set of simple types is exactly defined.

Definition 1 (Bounded type variables). *Let* $\mathsf{SType}_{TS}(\,BTV\,)$ *be a set of simple types. Then, the set of* bounded type variables *is an indexed set* $BTV = (BTV^{(ty)})_{ty \in \mathsf{I}(\,\mathsf{SType}_{TS}(\,BTV\,)\,)}$*, where each type variable is assigned to an intersection of simple types.*
$\mathsf{I}(\,\mathsf{SType}_{TS}(\,BTV\,)\,)$ *denotes the set of intersections over simple types (cp. Def. 4). In the following we will write a type variable* $\mathtt{a}$ *bounded by the type ty as* $\mathtt{a}|_{ty}$*. Type variables which are not bounded can be considered as bounded type variables by* `Object`*.*

Example 2. Let the following Java 5.0 class be given.

```
class BoundedTypeVars<a extends Number> {
    <t extends Vector<Integer> & J<a> & I,
     r extends Number> void m ( ...) { ...}
}
```

The set of bounded type variable BTV of the method `m` is given as $BTV^{(\mathtt{Number})} = \{\,\mathtt{a}, \mathtt{r}\,\}$ and $BTV^{(\mathtt{Vector<Integer>}\ \&\ \mathtt{J<a>}\ \&\ \mathtt{I})} = \{\,\mathtt{t}\,\}$.

Definition 2 (Type signature, type constructor). *Let* $\mathsf{SType}_{TS}(\,BTV\,)$ *be a set of simple types. A* type signature TS *is a pair* $(\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *where* BTV *is an indexed set of bounded type variables and* TC *is a* $(BTV)^*$*–indexed set of* type constructors *(class names).*

Example 3. Let the Java 5.0 program from Example 1 be given again. Then, the corresponding indexed set of type constructors is given as $TC^{(\mathtt{a}|_{\mathtt{Object}})} = \{\,\mathtt{A}, \mathtt{B}, \mathtt{I}, \mathtt{J}\,\}$, $TC^{(\mathtt{a}|_{\mathtt{I<b>}}\ \mathtt{b}|_{\mathtt{Object}})} = \{\,\mathtt{C}\,\}$, and $TC^{(\mathtt{a}|_{\mathtt{B<a>\&J<b>}}\ \mathtt{b}|_{\mathtt{Object}})} = \{\,\mathtt{D}\,\}$.

For the following definitions, we need the concept of *capture conversion* ([1] §5.1.10).
In order to define the capture conversion, we have to introduce the *implicit type variables* with lower and upper bounds first. Implicit type variables are used in Java 5.0 during the *capture conversion*, where the wildcards are replaced by implicit type variables. Implicit type variables cannot be used explictly in Java 5.0 programs.
We denote an implicit type variable T with a lower bound ty by ${}_{ty}|T$ and with an upper bound ty' by $T|^{ty'}$.
The *capture conversion* transforms types with wildcard type arguments to equivalent types, where the wildcards are replaced by implicit type variables.

Definition 3 (Capture conversion). *Let* $TS = (\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *be a type signature. Furthermore, let be* $C \in TC^{(a_1|_{u_1}, \ldots, a_n|_{u_n})}$ *and* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \in \mathsf{SType}_{TS}(\,BTV\,)$*. Thus, the* capture conversion $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>}$ *of* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ *is defined as:*

- *if* $\theta_i = ?$ *then* $\overline{\theta}_i = b_i|^{u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n]}$, *where* b_i *is a fresh implicit type variable.*
- *if* $\theta_i = ?$ `extends` $\theta'_{\mathtt{i}}$ *then* $\overline{\theta}_i = b_i|^{\theta'_i \,\&\, u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n]}$, *where* b_i *is a fresh implicit type variable with* upper bound $\theta'_i \,\&\, u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n]$).
- *if* $\theta_i = ?$ `super` $\theta'_{\mathtt{i}}$ *then* $\overline{\theta}_i =_{\theta'_i}|b_i|^{u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n]}$, *where* b_i *is a fresh implicit type variable with* lower bound θ'_i *and* upper bound $u_i[a_j \mapsto \overline{\theta}_j \,|\, 1 \leqslant j \leqslant n]$).
- *otherwise* $\overline{\theta}_i = \theta_i$

The capture conversion of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ *is denoted by* $CC(\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}\,)$.

Example 4. Let the indexed set of type constructors TC from Example 3 be given again. Then the following holds

$$CC(\,\texttt{A<? extends Integer>}\,) = \texttt{A<X}|^{\texttt{Integer\&Object}}\texttt{>}, \text{ as } \texttt{A} \in TC^{(\texttt{a}|_{\texttt{Object}})},$$
$$CC(\,\texttt{C<? extends A<c>, c>}\,) = \texttt{C<Y}|^{\texttt{A<c>\&I<c>}}\texttt{, c>}, \text{ as } \texttt{C} \in TC^{(\texttt{a}|_{\texttt{I<b>}}\ \texttt{b}|_{\texttt{Object}})},$$
$$CC(\,\texttt{B<? super Integer>}\,) = \texttt{B<}_{\texttt{Integer}}|\texttt{Z}|^{\texttt{Object}}\texttt{>}, \text{ as } \texttt{B} \in TC^{(\texttt{a}|_{\texttt{Object}})}.$$

The following definition of the set of simple types is connected to the corresponding definition of parameterized types in [1], §4.5.

Definition 4 (Simple types). *The set of* simple types $\mathsf{SType}_{TS}(\,BTV\,)$ *for a given type signature* $(\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *is defined as the smallest set satisfying the following conditions:*

- *For each intersection type ty:* $\underline{BTV^{(ty)}} \subseteq \mathsf{SType}_{TS}(\,BTV\,)$
- $\underline{TC^{()}} \subseteq \mathsf{SType}_{TS}(\,BTV\,)$
- *For* $ty_i \in \mathsf{SType}_{TS}(\,BTV\,)$
  $\cup \{\,?\,\}$
  $\cup \{\,\texttt{? extends } \tau \mid \tau \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$
  $\cup \{\,\texttt{? super } \tau \mid \tau \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$
  and $C \in TC^{(a_1|_{b_1} \ldots a_n|_{b_n})}$ *holds*

  $$\underline{C\texttt{<}ty_1, \ldots, ty_n\texttt{>}} \in \mathsf{SType}_{TS}(\,BTV\,)$$

  if after $C\texttt{<}ty_1, \ldots, ty_n\texttt{>}$ *subjected to the capture conversion resulting in the type* $C\texttt{<}\overline{ty_1}, \ldots, \overline{ty_n}\texttt{>}$[1], *for each actual type argument* $\overline{ty_i}$ *holds:*

  $$\overline{ty_i} \leq^* b_i[a_j \mapsto \overline{ty_j} \mid 1 \leqslant j \leqslant n],$$

  where $\leq^*$ *is a subtyping ordering (Def. 5).*
- *The* $\underline{\textit{set of implicit type variables with lower or upper bounds}}$ *belongs to* $\mathsf{SType}_{TS}(\,BTV\,)$

The set of intersection types *over a set of* $\mathsf{SType}_{TS}(\,BTV\,)$ *is denoted by:*
$\mathsf{I}(\,\mathsf{SType}_{TS}(\,BTV\,)\,) = \{\,\theta_1 \,\&\, \ldots \,\&\, \theta_n \mid \theta_i \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$

The following example shows the simple type construction, where the arguments of the type constructors are unbounded, respectively, bounded by `Object`.

[1] For non wildcard type arguments the capture conversion $\overline{ty}_i$ equals ty_i

Example 5. Let the Java 5.0 program from Example 1 and the corresponding indexed set of type constructors TC from Example 3 be given again. Let additional `Integer` $\in TC^{()}$.
The terms `A<Integer>` and `A<I<Integer>>` are simple types.
From `Integer` $\in TC^{()}$ follows `Integer` is a simple type. As `A` $\in TC^{(\mathtt{a}|_{\mathtt{Object}})}$ with $ty_1 =$ `Integer` follows, that `A<Integer>` is a simple type. From this follows as `I` $\in TC^{(\mathtt{a}|_{\mathtt{Object}})}$ with $ty_1 =$ `I<Integer>`, that `A<I<Integer>>` is also a simple type. As `A<a>` $\leq^*$ `I<a>` the type term `C<A<Integer>, Integer>` is also a simple type. In contrast `C<Integer, Integer>` is no simple type, as `Integer` $\not\leq^*$ `I<Integer>`.

After the definitions of the subtyping relation, we give another example, where the arguments of the type constructors are bounded and wildcards are used.

The set of bounded type variables BTV is in the following extended by the lower and upper bounded type variables.

3 Subtyping in Java 5.0

The Java 5.0 inheritance hierarchy consists of two different relations: The "extends relation" (in sign $<$) is explicitly defined in Java 5.0 programs by the *extends*, and the *implements* declarations, respectively. The "subtyping relation" (cp. [1], §4.10) is built as the reflexive, transitive, and instantiating closure of the extends relation.
In the following we will use $_?\theta$ as an abbreviation for the type term "`? extends` θ" and $^?\theta$ as an abbreviation for the type term "`? super` θ".

Definition 5 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(BTV)$). *Let $TS = (\mathsf{SType}_{TS}(BTV),\ TC)$ be a type signature of a given Java 5.0 program and $<$ the corresponding extends relation. The* subtyping relation $\leq^*$ *is given as the reflexive and transitive closure of the smallest relation satisfying the following conditions:*

- *if $\theta < \theta'$ then $\theta \leq^* \theta'$.*
- *if $\theta_1 \leq^* \theta_2$ then $\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)$ for all substitutions $\sigma_1,\ \sigma_2 : BTV \to \mathsf{SType}_{TS}(BTV)$, where for each type variable a of θ_2 holds $\sigma_1(a) = \sigma_2(a)$ (soundness condition).*
- *$a \leq^* \theta_i$ for $a \in BTV^{(\theta_1 \& \ldots \& \theta_n)}$ and $1 \leqslant i \leqslant n$*
- *It holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ if for each θ_i and θ'_i, respectively, one of the following conditions is valid:*
  - *$\theta_i = {}_?\overline{\theta}_i$, $\theta'_i = {}_?\overline{\theta}'_i$ and $\overline{\theta}_i \leq^* \overline{\theta}'_i$.*
  - *$\theta_i = {}^?\overline{\theta}_i$, $\theta'_i = {}^?\overline{\theta}'_i$ and $\overline{\theta}'_i \leq^* \overline{\theta}_i$.*
  - *$\theta_i, \theta'_i \in \mathsf{SType}_{TS}(BTV)$ and $\theta_i = \theta'_i$*
  - *$\theta'_i = {}_?\theta_i$*
  - *$\theta'_i = {}^?\theta_i$*

  (cp. [1] §4.5.1.1 type argument containment)
- *Let $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>}$ be the capture conversions of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ and $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ then holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.*

- *For an intersection type* $ty = \theta_1 \,\&\, \ldots \,\&\, \theta_n$ *holds* $ty \leq^* \theta_i$ *for any* $1 \leqslant i \leqslant n$*.*
- $T|^{(\theta_1 \& \ldots \& \theta_n)} \leq^* \theta_i$ *for any* $1 \leqslant i \leqslant n$*.*
- $\theta \leq^* {}_{\theta}|T$

Corollary 1. *The subtyping relation is an ordering.*

The following examples illustrates the subtyping definition.

Example 6. Let the Java 5.0 program from Example 1 be given again. Then the following relationships hold:

- `A<a>` $\leq^*$ `I<a>`, as `A<a>` $<$ `I<a>`
- `A<Integer>` $\leq^*$ `I<Integer>`, where $\sigma_1 = [a \mapsto \texttt{Integer}] = \sigma_2$
- `A<Integer>` $\leq^*$ `I<? extends Object>`, as `Integer` $\leq^*$ `Object`
- `A<Object>` $\leq^*$ `I<? super Integer>`, as `Integer` $\leq^*$ `Object`

The following example shows, how the capture conversions is used.

Example 7. Let the subtyping relationship `Vector<Vector<a>>` $\leq^*$ `Matrix<a>` be given. The the following holds:

$$\texttt{Matrix<Integer>} \leq^* \texttt{Vector<Vector<Integer>>}$$

From this follows the question if it holds

$$\texttt{Matrix<}_?\texttt{Integer>} \overset{!}{\leq^*} \texttt{Vector<Vector<}_?\texttt{Integer>>}$$

or if it holds

$$\texttt{Matrix<}_?\texttt{Integer>} \overset{!}{\leq^*} \texttt{Vector<}_?\texttt{Vector<}_?\texttt{Integer>>}$$

The two Hasse-diagrams presented in Fig. 1 shows that only the second approach is correct. For the supertype construction of `Matrix<`$_?$`Integer>` follows by the definition of the subtyping relation that the capture conversion `Matrix<X|`$^{\texttt{Integer}}$`>` must be built. This means that, `Vector<Vector<X|`$^{\texttt{Integer}}$`>>` is a supertype of `Matrix<`$_?$`Integer>`. As `Vector<Vector<`$_?$`Integer>>` is no supertype of `Vector<Vector<X|`$^{\texttt{Integer}}$`>>`, it is also no supertype of `Matrix<`$_?$`Integer>`. In contrast the simple type `Vector<`$_?$`Vector<`$_?$`Integer>>` is a supertype of `Vector<Vector<X|`$^{\texttt{Integer}}$`>>`, which means that `Vector<`$_?$`Vector<`$_?$`Integer>>` is also supertype of `Matrix<`$_?$`Integer>`.

4 Soundness of the Java 5.0 type system

Let us consider again the definition of the subtyping relation (Def. 5). It is surprising that the condition for σ_1 and σ_2 in the second item is not $\sigma_1(\,a\,) \leq^* \sigma_2(\,a\,)$, but $\sigma_1(\,a\,) = \sigma_2(\,a\,)$. This is necessary to get a sound type system. This property is the reason for the introduction of wildcards in Java 5.0 (cp. [1], §5.1.10).
Let the following Java 5.0 classes be given.

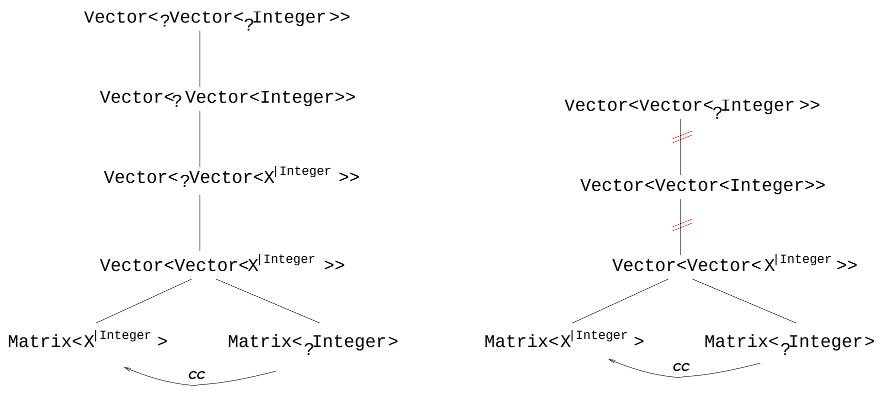


Fig. 1. Subtyping relation with capture conversion

```
class Super { ...}
class Sub extends Super { ...}

class Application {
  public static void main(String[] args) {
    Vector<Super> v = new Vector<Sub> ();  //not really correct
    v.addElement(new Super()); }
}
```

An element of the type `Vector<Sub>` is assigned to the variable `v` of the type `Vector<Super>`. This is no problem, as all elements which have the type `Sub` have also the type `Super`. Then a new element of the type `Super` is added to the vector which is assigned to the variable `v`. Now we have the problem, that elements of this vector have the type `Sub` and `Super` is no subtype of `Sub`. If this would be type correct, the type system would be unsound.
As in expression assignments, like `Vector<Super> v = new Vector<Sub>();` the type of the right hand side must be a subtype of the left hand side's type, the subtyping restriction of Def. 5 is introduced. The restriction demands that the declaration must be `Vector<Super> v = new Vector<Super>> ();`. But, sometimes assignments like

```
Vector<Super> v = new Vector<Sub> ();
```

would although be desirable. Therefore wildcards are introduced.
For example it is allowed:

```
Vector<? extends Super> v = new Vector<Sub> ();
```

Now `Vector<Sub>` is a subtype of `Vector<? extends Super>`, which means the assignment is type correct. In this case `v.addElement(new Super());` is prohibited as `Super` is no subtype of "`? extends Super`". This means that the unsoundness problem is also solved.

On the other hand, if an element of a subclass should be added to a vector of its superclass, the parameter of the vector must have a lower bound:

```
Vector<? super Super> v2 = new Vector<Super> ();
v2.addElement(new Sub());
```

In this case only vectors with a parameter of a supertype of `Super` can be assigned to `v2`. This means that no unsoundness arises.
We have used wildcard types like "`? extends Super`", although there are no simple types. Therefore we have to extend the set of simple type.

Definition 6 (Extended simple types). *Let* $\mathsf{SType}_{TS}(\, BTV \,)$ *be a set of simple types. The corresponding set of* extended simple types *is given as*

$$\begin{aligned}\mathsf{ExtSType}_{TS}(\, BTV \,) = {} & \mathsf{SType}_{TS}(\, BTV \,) \\ & \cup \{\, ? \,\} \\ & \cup \{\, ?\,\texttt{extends}\ \theta \mid \theta \in \mathsf{SType}_{TS}(\, BTV \,) \,\} \\ & \cup \{\, ?\,\texttt{super}\ \theta \mid \theta \in \mathsf{SType}_{TS}(\, BTV \,) \,\}\end{aligned} \quad .$$

Wildcard types cannot be used explicitly in Java 5.0 programs. But they are allowed as instances of type variables, which means that types like this occur implicitly during the type check of Java 5.0 programs (cp. example 8).
Additionally, we have to extend the subtyping relation to wildcard types.

Definition 7 (Subtyping relation $\leq^*$ on $\mathsf{ExtSType}_{TS}(\, BTV \,)$). *Let $\leq^*$ be a subtyping relation on a given set of simple types* $\mathsf{SType}_{TS}(\, BTV \,)$*. Then $\leq^*$ is continued on the corresponding set of extended simple types* $\mathsf{ExtSType}_{TS}(\, BTV \,)$ *by: For $\theta \leq^* \theta'$:*

- $\theta \leq^* {}^{?}\theta'$,
- ${}_{?}\theta \leq^* \theta'$, *and*
- ${}_{?}\theta \leq^* {}^{?}\theta'$.

In the following we give two examples, which shows some properties of wildcard types in Java 5.0.

Example 8. Let us consider again the class `Vector` with its methods `addElement` and `elementAt` and the classes `Sub` and `Super`.
Let the following declaration be given:

```
Vector<? extends Super> v = new Vector<Sub> ();
```

As said before the methodcall `v.addElement(new Super());` is not correct as `Super` $\not\leq^*$ ${}_{?}$`Super`. But as it holds ${}_{?}$`Super` $\leq^*$ `Super` the assignment

```
Super sup = v.elementAt(0);
```

is correct.
Vice versa for

```
Vector<? super Super> v2 = new Vector<Super> ();
```

the methodcall `v2.addElement(new Sub());` is correct as `Sub` $\leq^*$ ${}^{?}$`Super`. But now

```
  Super sup = v2.elementAt(0);   //not really correct
```
is not correct, as ${}^{?}$Super $\not\leq^*$ Super.
Furthermore, the methodcall
```
  v2.addElement(v.elementAt(0));
```
is correct, as it holds ${}_{?}$Super $\leq^*$ ${}^{?}$Super.

Example 9. Let the following Java 5.0 program be given:

```
    class B<a> { ... }
    class C<a> extends B<a> { ... }
    class Matrix<a> extends Vector<Vector<a>> { ... }
    class ExtMatrix<a> extends Matrix<a> { ... }
    class Super { ... }
    class Sub extends Super { ... }
```

Now we will give some applications, which show properties of the subtyping ordering and explain that the type system is sound.
The first property is obvious: For all $\theta \leq^* \theta'$ holds B<θ> $\leq^*$ B<${}_{?}\theta'$>.
This leads to the question, if for any class Y holds also Y<B<θ>> $\leq^*$ Y<B<${}_{?}\theta'$>>?
This question can be answered considering the fourth condition of definition 5. As B<θ> $\neq$ B<${}_{?}\theta'$>, the argument type of Y would have to be a wildcard argument. But B<${}_{?}\theta'$> is no wildcard argument. This means that Y<B<θ>> $\not\leq^*$ Y<B<${}_{?}\theta'$>>.
If this would be correct, the following Java 5.0 fragment would also be correct

```
  Vector<B<? extends Super>> v = new Vector<B<Sub>> (); //is not really
                                                        //correct
  v.addElement(new B<Super>());
```

If this would be correct, the type system would be unsound, as an element of the type B<Super> is assigned to a vector of elements of the type B<Sub>.
But for any Y holds obviously Y<B<θ>> $\leq^*$ Y<${}_{?}B$<${}_{?}\theta'$>>.
The next question is, if it holds Matrix<θ> $\leq^*$ Vector<Vector<${}_{?}\theta'$>> for $\theta \leq^* \theta'$?
As Matrix<a> $\leq^*$ Vector<Vector<a>> for a type variable a $\in BTV$ holds Matrix<θ> $\leq^*$ Vector<Vector<θ>>. But

$$\texttt{Matrix<}\theta\texttt{>} \leq^* \texttt{Vector<Vector<}\theta\texttt{>>} \not\leq^* \texttt{Vector<Vector<}{}_{?}\theta'\texttt{>>}$$

(cp. Example 7). This means that Matrix<θ> $\not\leq^*$ Vector<Vector<${}_{?}\theta'$>>.
We will also consider, what would happen if it would be correct:

```
  Vector<Vector<? extends Super>> v = new Matrix<Sub>(); //is not really
                                                         //correct
  v.addElement(new Vector<Super>());
```

In this case the type system would also be not sound, as an element of the type Vector<Super> is assigned to a matrix of elements of the type Sub.
A further question arises if we consider again that for $\theta \in \mathsf{SType}_{TS}(\,BTV\,)$ holds Matrix<θ> $\leq^*$ Vector<Vector<θ>> . The question is, if it holds also Matrix<${}_{?}\theta$> $\leq^*$ Vector<Vector<${}_{?}\theta$>>? As ${}_{?}\theta \notin \mathsf{SType}_{TS}(\,BTV\,)$ for $a \in BTV$ from Matrix<a> $\leq^*$ Vector<Vector<a>> does not follow Matrix<${}_{?}\theta$> $\leq^*$ Vector<Vector<${}_{?}\theta$>>.
For this we consider again an application:

```
Vector<Vector<? extends Super>> v;
Matrix<? extends Super> w = new Matrix<Sub>();
v = w; //is not really correct
v.addElement(new Vector<Super>());
```

This application shows again that the type system would not be sound. Therefore Matrix<$_?\theta$> $\not\leq^*$ Vector<Vector<$_?\theta$>> and the assignment `v = w` is not type correct.
But Matrix<$_?\theta$> $\leq^*$ Vector<$_?$Vector<$_?\theta$>> holds. As the capture conversions of Matrix<$_?\theta$> is Matrix<T$|^\theta$> and $T|^\theta \in \mathsf{SType}_{TS}(\,BTV\,)$ follows, that Matrix<T$|^\theta$> $\leq^*$ Vector<Vector<T$|^\theta$>>. With Vector<Vector<T$|^\theta$>> $\leq^*$ Vector<$_?$Vector<T$|^\theta$>> $\leq^*$ Vector<$_?$Vector<$_?$T$|^\theta$>> $\leq^*$ Vector<$_?$Vector<$_?\theta$>> follows Matrix<$_?\theta$> $\leq^*$ Vector<$_?$Vector<$_?\theta$>>.

Often the properties covariance respectively contravariance of type constructors are considered in object-oriented languages. Java 5.0 type constructors are neither covariant nor contravariant. The following corollary shows corresponding properties in Java 5.0.

Corollary 2 (Subtyping properties). *For two simple types $\theta \lneq^* \theta'$ and a type constructor* `Cl` *(class name) holds*

– Cl<θ> $\not\leq^*$ Cl<θ'>
– Cl<θ'> $\not\leq^*$ Cl<θ>
– ? extends $\theta \leq^* \theta'$, *but* Cl<θ> $\leq^*$ Cl<? extends θ'>.
– (? extends) $\theta \leq^*$? super θ' *but* CL<(? super) θ'> $\leq^*$ CL<? super θ>.

5 Conclusion and Outlook

In this paper we presented a formalization of the Java 5.0 type system. We defined the set of Java 5.0 simple types as type terms, which are explicitly allowed in Java 5.0 programs. We extended this set by wildcard types, which appear implicitly during the type checking. We defined a subtyping ordering at first on the set of Java 5.0 simple types and extended it to wildcard types. Additionally, we considered the soundness of the Java 5.0 type system. We showed, how the Java 5.0 type system becomes quite flexible by introducing the wildcards without loosing the soundness.
The Java 5.0 type system is the base for the definition of a type inference algorithm. We will give a type inference algorithm for Java 5.0 type terms with wildcards. Furthermore, we will implement this system.

References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification. 3rd edn. The Java series. Addison-Wesley (2005)
2. Plümicke, M., Bäuerle, J.: Typeless Programming in Java 5.0. In Gitzel, R., Aleksey, M., Schader, M., Krintz, C., eds.: 4th International Conference on Principles and Practices of Programming in Java. ACM International Conference Proceeding Series, Mannheim University Press (August 2006) 175–181

Source-to-Source Transformations for WCET Analysis: The CoSTA Approach

Adrian Prantl[*]

TU Vienna, Austria
adrian@complang.tuwien.ac.at

Abstract. *Worst-case execution time* (WCET) analysis is concerned with computing upper bounds of the maximum computation time of a program. This is indispensable for the development of safety-critical real-time systems, where missing a deadline can have disastrous consequences, including the loss of lives. Tools for WCET analysis typically analyze the object-code of a program since this is the code which is actually executed. Simultaneously, they usually rely on user-provided *annotations* such as loop-bounds or execution frequencies of program statements in order to be effective. From the perspective of a programmer, it is often more adequate to provide such information on the source code level than on the object code level. This, however, introduces a gap between the WCET annotation and the WCET analysis level. Within the CoSTA project (*Compiler Support for Timing Analysis*) we are aiming at bridging this gap. Fundamental to this is to provide appropriate new compiler support allowing to transform source code annotations into equivalent object code annotations.

In this paper we outline the approach taken in the CoSTA project to achieve this. In this project, which has recently been started, the compilation process is decomposed into a high-level machine-independent and a low-level machine-dependent two-stage process. Here, we will focus on the first stage of this process, the *high-level source-to-source compiler* and the *annotation framework*.

1 Background and Motivation

For safety-critical real-time systems the timing behavior is as important as the correctness of the calculations, since the consequences of missing a deadline can be equally catastrophic as an incorrect calculation, causing even the loss of lives. Before deploying such a system it is thus indispensable to ensure that the system meets in addition to its functional constraints also its timing constraints. Determining the *worst-case execution time* (WCET) of a program as precisely as possible is essential for this.

Intuitively, the determination of the WCET of a program is equivalent to the search for the most time-consuming path in the control flow graph of the

[*] This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P18925-N13.

program. An early approach for WCET analysis is called *timing schema* [10]. In this approach, the execution time of each basic block is assumed to be a constant and the number of iterations of each loop construct to be bounded by an upper limit, while branches are replaced by the $max()$-function.

A more sophisticated approach for supplying path information to the WCET calculation tool is based on *linear flow constraints* [11]. In this approach the program flow information - often called *flow facts* - is expressed as a system of inequalities that forms the input of an *integer linear programming* (*ILP*) problem that can be solved efficiently by a variety of tools [8]. This method is also called *implicit path enumeration technique* (*IPET*). It is implemented by commercially available tools like AiT [2] and Bound-T [4].

While it is often possible to automatically extract flow facts from the program code, it is usually necessary to require the programmer to (additionally) *manually* annotate the program with appropriate flow facts. On the one hand, this is necessary because the overall problem is undecidable (such as the determination of loop bounds). On the other hand, the programmer might have additional knowledge about the input data.

State-of-the-art tools perform the WCET-calculation on the *object code level*, which is as close as possible to the code that will eventually run on the target hardware. These tools expect that any user annotations are provided in the object code. This, however, is very demanding for a programmer and error-prone. Moreover, it implies to reassure the correctness of the annotations after each compiler run during the development phase.

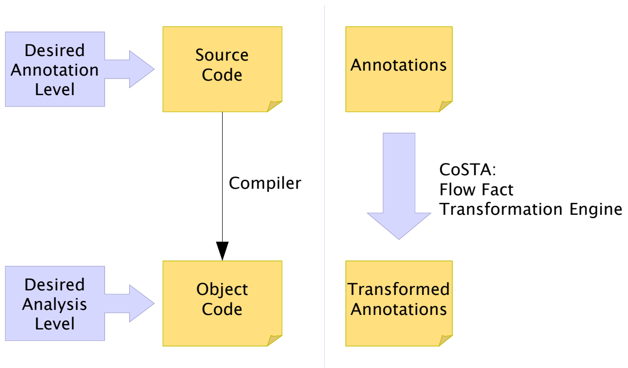


Fig. 1. Bridging the gap between annotation and analysis level

The CoSTA project aims at improving this situation. By developing and providing suitable compiler support it aims at allowing the user to add the annotations to the source code of a program which are then – together with the source code – transformed to the object code level when compiling the program.

In this paper we present and discuss the overall architecture of the system we develop in the CoSTA project, highlight the essential benefits envisioned, and discuss important features of the current state of the prototype implementation. In particular, we highlight the important role of optimizing source-to-source transformations for the overall approach. They are crucial for generating high-performance object code and for ensuring portability especially of the first stage of our approach.

2 The CoSTA-Architecture for Source-based Annotations

As indicated in the previous section, the CoSTA project strives for bridging the gap between source code annotations and object code WCET calculation. More specifically, this shall be achieved and demonstrated by developing and implementing a safe transformation framework for flow facts [5], where we target a subset of the C++ language as the programming language. The final framework shall seamlessly interact with existing IPET-based calculation tools.

As discussed before, we expect that such a source-based WCET-annotation framework makes WCET analysis more easily amenable to a programmer and overall more effective. In particular, we perceive the following benefits to be of particular value.

Validation. Automatically computed annotations on the source code level can easily be verified by the user. The increased trust in the reliability of such an analysis tool should help to reduce the amount of annotations a user makes manually.

Refinement. The user can introduce his domain-specific knowledge to provide additional information which is beyond the scope of the automatic analysis, and which can then be integrated into a cyclic work flow of perpetual refinement.

Visualization. With applying source-to-source transformations, the user can conveniently follow the steps of the compiler and thus fine-tune optimization options according to their impact on the WCET.

Figure 2 illustrates the architecture of the CoSTA approach to achieve these goals. Fundamental is the decomposition of the system into a *high-level source-to-source transformation framework* (first stage) and a *low-level WCET-aware code generation back end* (second stage).

This decomposition is motivated by the fact that many optimizations can be performed at a very high abstraction level; in our case the *abstract syntax tree* (*AST*). This way, the optimization step is independent from the target machine tool chain, but may still be parameterized to reflect specific machine characteristics. Moreover, if the WCET-critical optimizations can be moved to the source code level, it suffices to employ a relatively simple back-end compiler to finally transform the (optimized) source code into assembly language. We define *WCET-critical optimizations* as optimizations that change the control flow, thus invalidating any annotations (which are in turn assertions about the control flow graph (CFG)). An example of a CFG-modifying optimization is *loop unrolling*, which directly modifies the iteration count of a loop.

First stage. The prototype implementation of the first stage of our system uses the SATIrE[1] framework which is being developed by Markus Schordan at TU

[1] `http://www.complang.tuwien.ac.at/markus/satire/`

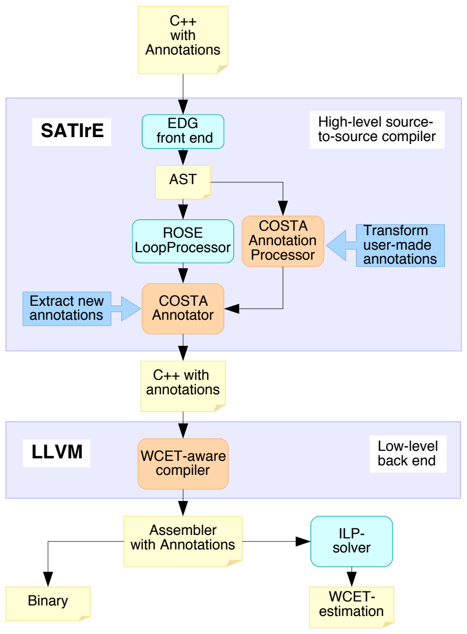


Fig. 2. A schematic overview of the CoSTA architecture

Vienna [13]. Intuitively, SATIrE is a tool environment that integrates the high-level source-to-source transformation and program analysis framework LLNL-ROSE[2] [14] with other program analysis tools, such as the *Program Analysis Generator* (*PAG*) from AbsInt [9]. For the purpose of the CoSTA project it is particularly important that SATIrE provides an external representation of the abstract syntax tree (AST) of a C++ program which can be both written to and read from. Moreover, this representation uses a syntax which can be interpreted as terms of the Prolog language. This allows us to specify program transformations directly in the Prolog language, using predicates to implement a term rewriting system. In fact, this approach was chosen to implement the first stage of our prototype.

The LLNL-ROSE framework contains a sophisticated loop optimization tool which has its roots in the Fortran D compiler. The tool can handle generic C++ programs and outputs C++ code that is very close to the original input; even templates are preserved. In the current CoSTA implementation, we use this tool to gain access to high-level optimization functions.

Second stage. The second stage of our system, the code generation back end is currently being implemented on the basis of LLVM[3], a relatively new compiler infrastructure based on a low-level virtual machine and SSA graphs that

[2] `http://www.llnl.gov/CASC/rose/`
[3] `http//www.llvm.org/`

Original user-annotated program

```
  int f(int a[]) {
    for(int i=0; i<N; i+=1) {
      if (a[i] < 0) {
        // domain-spec. knowledge
        Restriction M1 <= 24
        Marker M1;
        ...
} } }
```

After loop unrolling with factor 2

```
  int f(int a[]) {
    for(int i=0; i<N; i+=2) {
      if (a[i] < 0) {
        Restriction M1 <= 24/2
        Marker M1;
        ...
      }
      if (a[i] < 0) {
        Restriction M2 <= 24/2
        Marker M2;
        ...
} } }
```

Fig. 3. Transformation of user-specified annotations

is implemented in C++ [7]. We plan to implement a WCET-aware instruction selection mechanism for complex hardware architectures on this basis. The back end will only use optimizations that are not WCET-critical. This means, they will not further modify the control flow graph of the program.

In the course of implementing a safe flow facts transformation framework, the key component is the *CoSTA Annotation Processor* which is currently under implementation. The Annotation Processor takes a user-annotated program and a sequence of optimizations as input and transforms the annotations according to a set of rules. It then inserts the updated annotations into the optimized program source. Figure 3 shows an example of such a transformation.

In the following section we highlight the key components of the first stage of our system, the CoSTA Annotator and the CoSTA Annotation Processor.

3 The CoSTA Annotator and the CoSTA Annotation Processor: Extracting and Transforming Flow Facts

The *CoSTA Annotator* automatically extracts flow facts information of a program. The CoSTA Annotator thus offers an alternate route to obtain annotated source code. In particular, it avoids bothering a user to manually annotate control flow information which can be automatically extracted from the source code of the program. In fact, in many cases flow facts like loop bounds can be automatically found by a *static analysis* of the program. On the other hand, flow facts might depend on domain-specific knowledge about input-data which is usually beyond the scope of static analyses. It is thus worth noting that the CoSTA Annotator is orthogonal to the *CoSTA Annotation Processor*. The latter transforms annotations alongside optimizing transformations it applies to a program. Exemplary, we will now discuss the automatic bounding of loops:

The automatic finding of upper bounds of loop constructs is one of the tasks of the CoSTA Annotator. Currently, the algorithm operates on counter-based *for*-loops. Since C and C++ do not have a strict *for*-statement in the sense of Fortran or Pascal, loops have to satisfy a few extra conditions to be analyzable. These conditions are verified by the Annotator in advance: Each *for*-statement consists of initializer, condition, increment and body. The loop has to be induction variable based, i.e., initializer, condition and increment have to modify and test the same variable. The loop body may not contain a write access to the induction variable. The initializer may be empty. The loop may not contain early exits, such as a *break* or *return* statement. To give the programmer a little more flexibility, we provide a preprocessor that transforms *while*-loops into for loops in case they satisfy these very conditions. Using the SATIrE-framework, we were able to implement this preprocessor in very few lines of Prolog.

The implemented loop-bounding algorithm uses two strategies of varying precision. The first approach uses *symbolic evaluation* of terms in the source code to solve the equation $Bound = (End - Start)/Step$ for the induction variable i. In order to solve the equation, the terms are transformed according to a set of *rules* that exploit algebraic properties like commutativity to reduce the equation term to a single value (Figure 5). In order to reach a fixpoint and thus to guarantee termination, the rules have to satisfy a *monotonicity* property. Here, this means that the term has to shrink with each application of a rule. It should be noted that this rule-based equation solver is a good example illustrating the benefits resulting from using Prolog as implementation language. To extract the necessary information about the possible values of the program variables, every node in the AST is decorated with a pre- and a postcondition which hold the possible values of all integer variables. The loop bounds that are shown in the listing were derived by the symbolic analysis. Often the value of a variable is

```
 1  // {empty}
 2  for (int i = 0; i < 42; i += 8) {
 3    // {i_range ∈ [0..41]}
 4    // LoopBound = 6
 5    for (int j = i; j < min(42, i+8); j += 1) {
 6      // {j_range ∈ [0..41], j_symbolic ∈ [i..i + 8]}
 7      // LoopBound = 8
 8    }
 9    // {j = 42}
10  }
11  // {i = 48}
```

Fig. 4. Analysis information for loop bounds

known to be within a certain *range*, as it is the case with the induction variable in the body of a loop. This range information is important for the second strategy used by the analysis, which is intended as a fallback in case the first one fails to find a precise result. Using the range information, it is often still possible to provide a conservative estimate for many loops. As can be seen in Figure 4, the analysis information gathered by the two strategies is of varying precision. While the *range* information tends to be more pessimistic, it can unfold its whole potential when it is combined with the equation solver. The major advantage of the symbolic approach is its ability to cancel out subterms of the equation such as i in line 6 of the example in Figure 4.

A More Complex Example

We conclude this section with discussing a more complex example, which is displayed in Figure 6. The original source code of the program consists of three nested loops that perform a multiplication of two two-dimensional arrays and

```
% (a+b)-b = a
transformation(sg_subtract_op(sg_add_op(E1, E2, _, _), E3, _, _),
               E1) :-
  term_identical(E2, E3).
% (v+i1)-i2 = v+i'
transformation(sg_subtract_op(sg_add_op(E1, E2, _, _), E3, _, _),
               sg_add_op(E1, E4, _, _)) :-
  isIntVal(E2, X), isIntVal(E3, Y), Z is X-Y, isIntVal(E4, Z).
...
```

Fig. 5. Excerpt from the rule base

accumulate the result into a third array. This source code is now processed by the ROSE loop optimizer (Figure 7). First *Loop Blocking* is performed, using a block size of 8, which should improve the locality of the memory accesses, then, the innermost loop is also being unrolled by a factor of 2. This necessitates an extra loop to be created to take care of the last element in case the total number of iterations is odd. Note that ROSE instantiates the uses of the macro N, which is important for the following step: Compared with the original loop, the resulting loop conditions are quite complex, but due to their constant bounds, they are fully analyzable. The Annotator can now traverse the AST top-down and use the existing information to solve the bounding equations for each for-loop (Figure 8). Consider the induction variables of the newly generated outer loops; for the loop bodies, only a range can be found by the analysis. However, this does not affect the precision of the analysis, since their occurrences in initializer and condition of the inner loops cancel each other out. For the newly added remainder part of the innermost loop, the initializer is missing. In this case the analysis has to use the postcondition of the previous for-statement to know about possible start values for k.

```
#define N 50
int i,j,k;
double a[N][N], b[N][N], c[N][N];
...
for (i = 0; i <= N-1; i+=1) {
  for (j = 0; j <= N-1; j+=1) {
    for (k = 0; k <= N-1; k+=1) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
} } }
```

Fig. 6. Phase 1: Original Code

4 Additional Benefits and Outlook

In this section we highlight some additional benefits of our overall approach and provide an outlook to further extensions.

It is worth noting that the framework presented in the previous sections offers two alternatives to create correctly annotated optimized code. The first one is to manually annotate the program, and then to optimize the annotated program. In this case the CoSTA Annotation Processor will update the annotations alongside the program optimization transformations. The second one is to first optimize the program and then to use the CoSTA Annotator to automatically annotate the optimized program. Figure 9 illustrates both alternatives. This flexibility

```
int _var_2;  int _var_1;  int _var_0;
int i;  int j;  int k;
double a[50][50];  double b[50][50];  double c[50][50];
...
for (_var_1 = 0; _var_1 <= 49; _var_1 += 8) {
  for (_var_2 = 0; _var_2 <= 49; _var_2 += 8) {
    for (_var_0 = 0; _var_0 <= 49; _var_0 += 8) {
      for (i = _var_2; i <= min2(49,_var_2 + 7); i += 1) {
        for (j = _var_1; j <= min2(49,_var_1 + 7); j += 1) {
          for (k = _var_0; k <= -1 + min2(49,7 + _var_0); k += 2) {
            (c[i])[j] = (((c[i])[j]) + (((a[i])[k]) * ((b[k])[j])));
            (c[i])[j] = (((c[i])[j]) + (((a[i])[1 + k]) * ((b[1 + k])[j])));
          }
          for (; k <= min2(49,7 + _var_0); k += 1) {
            (c[i])[j] = (((c[i])[j]) + (((a[i])[k]) * ((b[k])[j])));
} } } } } }
```

Fig. 7. Phase 2: Cache optimized code (Loop Blocking + Unrolling)

```
...
for (_var_1 = 0; _var_1 <= 49; _var_1 += 8) {
  #pragma WCET_LOOP_BOUND 7
  for (_var_2 = 0; _var_2 <= 49; _var_2 += 8) {
    #pragma WCET_LOOP_BOUND 7
    for (_var_0 = 0; _var_0 <= 49; _var_0 += 8) {
      #pragma WCET_LOOP_BOUND 7
      for (i = _var_2; i <= min2(49,_var_2 + 7); i += 1) {
        #pragma WCET_LOOP_BOUND 8
        for (j = _var_1; j <= min2(49,_var_1 + 7); j += 1) {
          #pragma WCET_LOOP_BOUND 8
          for (k = _var_0; k <= (-1) + min2(49,7 + _var_0); k += 2) {
            #pragma WCET_LOOP_BOUND 5
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
            c[i][j] = c[i][j] + a[i][1 + k] * b[1 + k][j];
          }
          for (; k <= min2(49,7 + _var_0); k += 1) {
            #pragma WCET_LOOP_BOUND 2
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
} } } } } }
```

Fig. 8. Phase 3: Automatically annotated optimized code

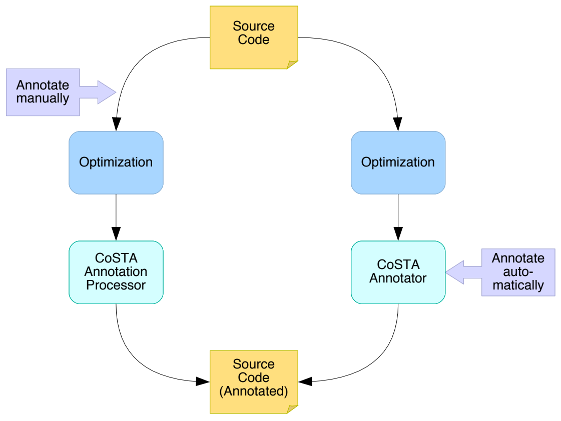


Fig. 9. Two routes to optimized annotated source code are provided in CoSTA

and dualism is only possible by using a high-level optimization approach as in our framework. Note, however, that the high-level approach requires a compiler back-end that guarantees to preserve the control flow in a way that does not alter the worst-case timing behavior of the program. Constructing such a back end is work-in-progress as part of the CoSTA project.

On modern processors, hardware resource allocation conflicts can trigger *timing anomalies*, where a locally faster execution increases the total execution time [16, 12]. Thus, another focus of our work on the compiler back end is to research scheduling and instruction selection algorithms for increased predictability.

In complex hardware architectures using features such as pipelines or instruction and data caches, the timing of an instruction depends highly on the execution history. Currently, the majority of annotation languages do not allow to specify explicit execution paths [6]. If these features shall be considered by a WCET calculation, it will be necessary to undertake a deeper look at annotation languages. While there are already approaches to integrate execution context information into the IPET calculation method [1], it will be necessary to create adequate annotation methods for context sensitive path descriptions as well.

5 Conclusions

The CoSTA project aims at making WCET analysis more effective and more amenable, especially by lifting the annotation level from the object code level to the source code level. Experiences with the current prototype implementation indicate that the chosen system architecture is well-suited to meet these goals. In particular, our experiences with optimizing source-to-source transformations indicate that these benefits can be achieved without sacrificing the performance of the object code of the application programs. Currently, we work on integrating more refined algorithms for automatic flow facts extraction and further source-to-source transformations. Simultaneously, we work on connecting the high-level first stage of our system with its low-level second stage, which will enable us to link our system to existing WCET analysis tools. As another strand of research in the CoSTA project we consider the development of advanced annotation languages which are even more suitable to reach the goals of the CoSTA project. In fact, investigating the adequacy of the commonly used annotation languages for this purpose, it turned out that all these languages have their own strengths and limitations motivating us to rise the *annotation language challenge* [6] – a challenge being closely related to and complementing the previously launched WCET tool challenge [3, 15].

Acknowledgements. The author would like to thank the members of the CoSTA-project team for discussions related to the subject of this paper. In particular, he would like to thank Jens Knoop for helpful comments on earlier versions of this paper.

References

1. J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.
2. C. Ferdinand. Worst case execution time prediction by static program analysis. *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 03:125a, 2004.
3. J. Gustafson. The WCET tool challenge 2006. In *Preliminary Proceedings 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.
4. N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proceedings Euromicro Worst-Case Execution Time Workshop 2002 (WCET 2002)*, 2003.
5. R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.
6. R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. WCET Analysis: The Annotation Language Challenge. In *Proceedings 7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007. To appear.
7. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
8. Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.
9. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
10. C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *Computer*, 24(5):48–57, May 1991.
11. P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.
12. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In F. Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
13. M. Schordan. Combining tools and languages for static analysis and optimization of high-level abstractions. In *Proceedings 24. Workshop of "GI-Fachgruppe Programmiersprachen und Rechenkonzepte"*, 2007.
14. M. Schordan and D. Quinlan. Specifying transformation sequences as computation on program fragments with an abstract attribute grammar. In *Proceedings Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 97–106. IEEE Computer Society Press, 2005.
15. L. Tan and K. Echtle. The WCET tool challenge 2006: External evaluation – draft report. In *Handout at the 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Paphos, Cyprus, November 2006. 13 pages.
16. I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings 5th International Conference on Quality Software*, Sep. 2005.

Slicing zur Modellreduktion von symbolischen Kellersystemen

Dirk Richter
richterd@informatik.uni-halle.de

Wolf Zimmermann
zimmer@informatik.uni-halle.de

Abstract: Sowohl für die Software-Modell-Prüfung als auch für das modellbasierte Testen sind Größe und Komplexität von Modellen entscheidende Einflussfaktoren. Wir haben Slicing auf Programmpunktebene als Programmanalyse für die Anwendung auf symbolische Kellersysteme (Remopla Modelle) übertragen, um kleinere und weniger komplexe Modelle zu erhalten. Die Messergebnisse zeigen eine teils erhebliche Laufzeitreduktion für den eingesetzten Modellprüfer Moped.

Schlüsselworte: Software-Modell-Prüfung (Model checking), Remopla, Moped, Slicing, Metrik (metric), McCabe

1 Einleitung

Während der Modellprüfung werden vorgegebene Eigenschaften überprüft (z.B. dass vor der Landung eines Flugzeugs stets das Fahrwerk ausgefahren ist). Ist die Modellprüfung erfolgreich, so erfüllt das Modell die vorgegebenen Eigenschaften **sicher** (in jedem Fall) im Gegensatz zum Testen, wo lediglich die Existenz von Fehlern festgestellt werden kann. Bei der Software-Modell-Prüfung werden Modelle aus Softwareprogrammen gewonnen. Modellprüfung als auch das modellbasierte Testen bzw. die Testfallgenerierung sind jedoch sehr aufwändig. Die Verwendung von Kellersystemen bei der Software-Modell-Prüfung ermöglicht dabei unter anderem die Berücksichtigung von rekursiven Methoden bzw. Funktionen. Modellprüfung von Software zur Steigerung der Softwarequalität ist bereits im Einsatz, wie die Projekte JavaPathFinder (NASA) [KT00], SMV [McM93] oder Moped [Jav01] zeigen. Allerdings sind die derzeit verfügbaren Werkzeuge (Tools) nicht in der Lage mittlere oder große Softwareprojekte zu überprüfen. Vielfach werden nur Variablen mit kleinem Zustandsraum unterstützt (logische/boolean Variable in Bebop [Tom00]) oder sogar vollständig auf **endliches** Modellprüfen beschränkt, da sonst der Suchraum für den Modellprüfer „explodiert". Wir untersuchen Ansätze, die solchen Einschränkungen möglichst nicht unterliegen.

Aufgrund der Abstraktion kann es im Modell zu Zustandsfolgen kommen, die im ursprünglichen Programm nicht auftreten können (sog. **False Negatives**). Ist aber einmal ein Modell hinreichend genau, um bestimmte Eigenchaften nachzuweisen, so erzeugen wir durch Slicing [M. 84] auch keine weiteren False Negatives.

Wir haben Erreichbarkeitsprobleme in Java-Programmen (Version 6.0) untersucht. Dabei transformieren wir Java-Programme mittels JMoped und `javac` nach Remopla. Remopla ist eine Beschreibungssprache für symbolische Kellersysteme und sieht einer imperativen Programmiersprache schon ähnlich. Diese Remopla-Modelle dienen dann dem Modellprüfer Moped als Eingabe, um Erreichbarkeitsprobleme in Kellersystemen zu lösen. Wir haben Werkzeuge zur Automatisierung (**JmBatch**) und zur Modellreduktion (**Optimizer**) konstruiert. Im Folgendem werden kurz grundlegende Begriffe, die Transformation von Java-Programmen in symbolische kellersysteme und verwendete Metriken erläutert. Abschließend werden Slicing auf diesen symbolischen Kellersystemen erklärt und unsere Ergebnisse präsentiert.

2 Begriffe

$M = (S, \rightarrow, L)$ heißt **Kripkestruktur**, falls S und A (nicht notwendigerweise endliche) Mengen sind, $\rightarrow \subseteq S \times S$ und $L : S \rightarrow 2^A$. In Abbildung 1 ist ein Beispiel einer Kipke-

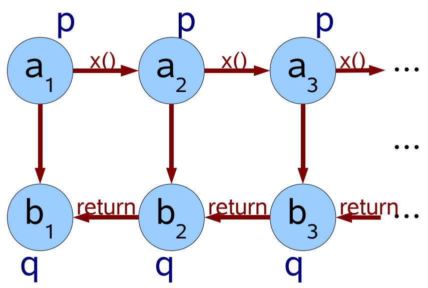


Abbildung 1: Beispiel einer Kipkestruktur

struktur zu sehen. $S = \{a_1, b_1, a_2, b_2 \ldots\}$ ist darin die Menge der Zustände, $A = \{p, q\}$ die Menge der Atome. Bei gegebener Kripkestruktur M ist das **Erreichbarkeitsproblem** die Frage, ob es in M von einem Zustand $s \in S$ einen Pfad zu einem anderen Zustand $z \in S$ gibt ($s \rightarrow^* z$?). Im Falle von $s, z \subset S$ (anstatt $s, z \in S$) sprechen wir vom **verallgemeinerten Erreichbarkeitsproblem**. Zur Beschreibung von (unendlich) großen Kripkestrukturen (wie diejenige aus Abbildung 1) verwenden wir Kellersysteme (Pushdown Systems). $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ heißt **Kellersystem**, falls P eine Menge von Zuständen, Γ eine Menge (das Kelleralphabet) und $\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine Menge von Transitionen ist. Informal ist ein Kellersystem ein Kellerautomat ohne Eingabe. (p, w) heißt **Konfiguration**, falls $p \in P, w \in \Gamma^*$. Auf Konfigurationen wird die Transitionsrelation $\hookrightarrow$ erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$. Statt des Erreichbarkeitsproblems kann auch das LTL-Modellprüfungsproblem für Kellersysteme gelöst werden. Dabei ist eine LTL-Formel für die durch das Kellsersystem beschriebene

Kripkestruktur gegeben und gefragt wird nach der Gültigkeit dieser Formel für alle Pfade beginnend bei einem Anfangszustand bzw. bei einer Menge von Anfangszuständen der Kripkestruktur. Büchiautomaten werden dabei aus LTL-Formeln konstruiert und können entsprechend minimiert werden [Tau03, FW05, FW02, Wol04, Jam04]. Obwohl das Modellprüfungsproblem für LTL und Linearzeit μ-Kalkül auf Kellersystemen i.A. *DEXPTIME*-vollständig ist [A. 97], bleibt der Aufwand für symbolisches Modellprüfen auf Kellersystemen mit LTL polynomiell in der Größe des konstruierten Büchiautomaten und des Kellersystems (bei fester LTL-Formel) [Jav00]. Natürlich ist daher die Größe des Büchiautomaten exponentiell in der Länge der LTL-Formel [Mat06].

3 Modellierung von Programmen als Kellersystem

Um die Semantik eines Programms in einem Kellersystem zu modellieren, wird jeder Programmzustand einschließlich potenziell unendlich vieler (rekursiver) Methodenaufrufe (die sog. **Aufrufhierarchie**) als Konfiguration des Kellersystems kodiert. In unserem Fall wird der Kontrollfluss eines Java-Programms direkt in Transitionen des Kellersystems überführt. Für andere Programmiersprachen funktioniert dies völlig analog.

Lokale Variablen von Methoden werden als Bitvektoren über das Kelleralphabet zusammen mit der Aufrufhierarchie im Keller repräsentiert. Globale Variablen (in Java Klassenvariablen), die Halde (engl. Heap) sowie Ausnahmen (engl. Exceptions) werden mit Hilfe der Zustände des Kellersystems beschrieben. Genaue Details zur Transformation kann man [Jan01] entnehmen. Voraussetzung zur Verarbeitung solcher symbolischer Kellersysteme ist ein symbolischer Modellprüfer wie Moped [Jav01]. Eine Beschreibung der Eingabesprache für Moped (Remopla) ist in [Ste06] zu finden.

Ein großer Zustandsraum des Programms (z.B. viele Variablen mit vielen Variablenwerten) ist ein generelles Problem, welches in der Literatur als **Problem der Zustandsexplosion** (engl. **State-Explosion-Problem**) bezeichnet wird [Edm01]. Daher sind Reduktionstechniken zur „Verkleinerung“ des Modells wichtig. Bei **endlicher** Modellprüfung werden solche Techniken bereits (wenn auch teils nur als Einzeluntersuchung von Hand) eingesetzt: Beispiele solcher Reduktionstechniken sind **Heap-Symmetry-Reduction** [Rad01], **Slicing/Influence Analysis** [M. 84], **Stotteräquivalenz** [E. 98], **Partial-Order-Reduction** [St04], **Shape Analysis** [Rei00], **Abstract Interpretation** [Pat96] oder sogar eine vollständige **Beschränkung auf Boolsche Programme** [Tom00]. Uns sind bisher keine Veröffentlichungen bekannt, in denen diese Techniken auf symbolische Kellersysteme übertragen wurden.

4 Metriken

Um einen Zusammenhang zwischen Eigenschaften von Remopla-Modellen und der Dauer des Modellprüfens zu untersuchen, wurden verschiedene Metriken für Remopla implementiert. Schließlich läßt sich die Zustandsraumgröße und Modellkomplexität von Remoplamodellen nur ungeeignet durch die Metrik **LOC** (lines of code) ausdrücken und schon gar nicht von einander unterscheiden. Dabei ist es wichtig zu wissen, welche Eigenschaften eines Modells die Modellprüfung besonders beeinflussen, um gezielt diesen „Flaschenhals“ beheben zu können. Zur Schätzung der Modellkomplexität bzw. **Kontrollflusskomplexität** verwenden wir die aus der Literatur bekannte Komplexitätsmetrik von **McCabe**. Diese Metrik gibt die Anzahl der konditionellen Zweige des Kontrollflussdiagramms an. Wir haben diese auch zyklomatische Zahl von McCabe genannte Metrik imperativer Programme für Remopla-Modelle adaptiert und verwenden sie als Maß für die Komplexität des Kontrollflusses in Remopla-Modellen.

Wie unsere Untersuchungen zeigten, sind sowohl die LOC-Metrik als auch die McCabe-Metrik ungeeignet zur Schätzung der Größe des Zustandsraums. In der Literatur ist uns bisher keine Metrik bekannt, welche die Größe des Zustandsraums geeignet schätzt. Dies ist auch nicht verwunderlich, da der Zustandsraum für Kellersysteme i.A. unbeschränkt ist (man denke z.B. an Rekursion).

Um dennoch den Zustandsraum zu schätzen, haben wir uns auf die benötigte Größe der sog. Köpfe von Konfigurationen eingeschränkt. Köpfe sind die linken Seiten von Transitionen des Kellersystems und bestehen aus einem Zustand und oberstem Kellersymbol des Kellersystems. Sei R ein Remopla-Modell, welches g Bits zur Repräsentation globaler Variablen benötigt (einschließlich Exceptions und Halde des ursprünglichen Java-Programms), l_i Bits für alle lokalen Variablen des Moduls/der Prozedur i und p Knoten im **interprozeduralen** Kontrollflussdiagramm hat. Dann haben wir die **ZR-Metrik** zur Schätzung des Zustandsraums wie folgt definiert:

$$ZR(R) := g + \sum_i l_i + \log_2 p. \tag{1}$$

Für die später in Abschnitt 5 verwendeten Beispiele ist in Abbildung 2 die Laufzeit des Modellprüfers Moped in Abhängigkeit der ZR-Metrik zu sehen. Dabei wurden für jedes Java-Programm 5 Modellprüfungen mit unterschiedlichen Parametern für die Bitbreite eines Integers (von 1 Bit bis 5 Bit) durchgeführt. Nicht dargestellte Punkte liegen rechts oberhalb außerhalb der Grafik oder wurden wegen eines Zeitüberlaufs (engl. Timeout) abgebrochen. Es ist ein klarer exponentieller Trend in der geschätzten Zustandsraumgröße erkennbar. Ein solcher Trend ist bei Verwendung der LOC- oder McCabe-Metrik nicht feststellbar. Durchgeführte Reduktionen der Remoplamodelle lassen sich damit nicht nur über die Dauer des Modellprüfens, sondern auch anhand dieser Metriken quantifizieren. Schließlich unterliegen Laufzeitmessungen naturgemäß diversen Schwankungen, nicht zuletzt, da Moped intern OBDDs (Ordered Binary Decison Diagramms) einsetzt, welche in einer ungünstig gewählten Variablenordnung zu unerwartet langen Laufzeiten führen können.

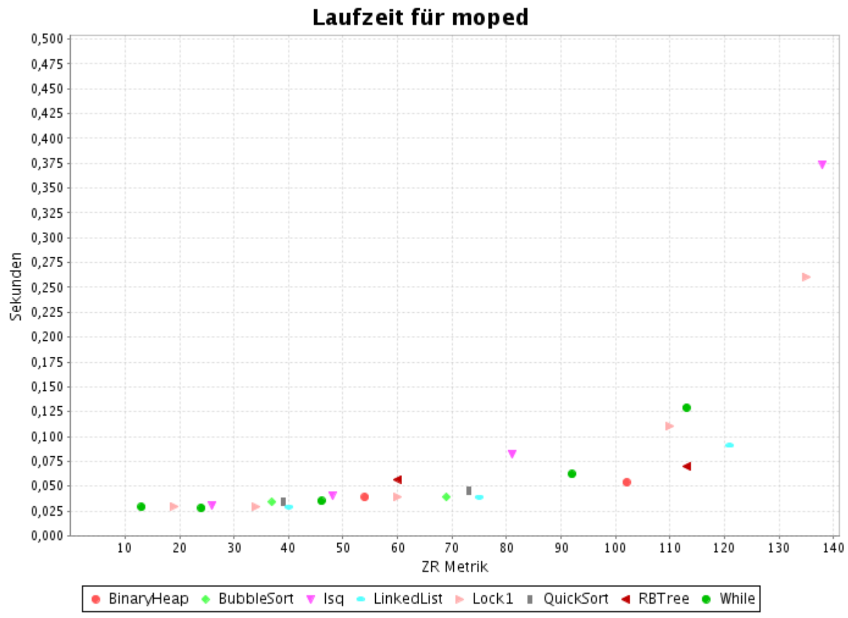


Abbildung 2: Laufzeit des Modellprüfers Moped für einige Beispielprogramme

5 Slicing und Ergebnisse

Mit Slicing- und Slicing ähnlichen Analysen lassen sich nicht ausgeführte Anweisungen eines Programms finden [Mat99, H. 01, E.04, E.04, Fle05, Ped06, M. 03]. Dies ermöglicht daher eine Reduktion der Modellgröße. Und verringert damit die Zeit und Komplexität für Testfallgenerierungen als auch für Modellprüfungen.

Wir haben interprozedurales Slicing auf Remopla-Modelle übertragen. In unserer Umsetzung wird ein Vorwärtsslice beginnend bei den Startkonfigurationen berechnet, welcher alle symbolischen Remopla-Konfigurationen des Modells enthält, welche aus den Startkonfigurationen über den Kontrollfluss erreichbar sind und reduzieren den Kontrollfluss entsprechend. Beginnend bei den Fehlerkonfigurationen berechnen wir anschließend einen Rückwärtsslice, um symbolische Remopla-Konfigurationen zu identifizieren, welche zu Fehlern im Modell über den Kontrollfluss führen können. Alle nicht im Rückwärtsslice auf dem reduzierten Kontrollflussgraph enthaltenen Konfigurationen können aus dem Modell entsprechend entfernt werden. Im Gegensatz zu anderen Reduktionstechniken bzw. Abstraktionstechniken entstehen durch Slicing keine neuen **False Negatives**.

Durch diese Technik konnten wir die in Abbildung 3 zu sehenden Modellprüfzeiten wie folgt verbessern. Als Testgrundlage dienten die 7 bereits in JMoped enthaltenen Java-Beispiele sowie 24 Java-Beispiele aus eigenen Untersuchungen (davon 11 durch Anpassung aus dem Coverage Eclipse-PlugIn [Fel06] gewonnen).

- In 2 der 31 Java-Beispiele wurde durch Slicing das Erreichbarkeitsproblem bereits negativ beantwortet, was die Anwendung eines Modellprüfers erübrigt [1].
- In 5 der 155 untersuchten Modellprüfungen konnte durch Slicing ein *Timeout* von 15 Minuten vermieden werden. D.h. diese 5 Modellprüfungen konnten früher nicht, jetzt aber schon abgeschlossen werden.
- In allen weiteren 30 Fällen, in denen das Modellprüfen mehr als 2 Sekunden gedauert hat, konnte durch Slicing die Modellprüfdauer (einschließlich der Slicing-Zeit) verringert werden.

Ist der Zustandsraum des Modells bereits hinreichend klein[2], so benötigt das Slicing teilweise auch mehr Zeit als die Modellprüfung an sich.

6 Zusammenfassung

Durch Einsatz von Slicing im Kontrollflussgraph auf Konfigurationenebene haben wir symbolische Kellersysteme (genauer: Remopla-Modelle) derartig transformiert, dass nur noch diejenigen Remopla-Modell-Transitionen im Modell enthalten sind, welche sowohl von den Startkonfigurationen erreichbar sind als auch zu angegebenen Fehlerkonfigurationen führen können. Dies reduziert die Zeit für den Modellprüfer Moped für Modelle mit hinreichend großem Konfigurationenraum. Analog wird damit aber auch der für Softwaretesten zu überdeckende Modellraum kleiner, wodurch weniger Softwaretests nötig werden.

Es sind weitere Programmanalysen zur Integration geplant, um durch Ausnutzung weiterer Eigenschaften der Remopla-Modelle noch kleinere Modelle (bezüglich ZR-Metrik) zu erzeugen. Geeignete ähnliche Ansätze finden sich in der Literatur als **Cone of Influence** (COI) [H. 01, E.04], **Localization** [E.04], **Live Variables Analysis** (LVA) [Fle05], **Influence Analysis** (IA) [Ped06] und **Lebendigkeits-Analyse** (engl. live range analysis) [M. 03]. Unser vorrangiges Ziel ist bisher entsprechend unserer Ergebnisse eine weitgehende Abstraktion von Daten.

Literatur

[E.04] E. M. Clarke, Fujita, P. Rajan, Reps S. Shankar. *Program Slicing of Hardware Description Languages*. L. Pierre and T. Kropf (Eds.): CHARME'99, LNCS 1703, In Conference on Correct Hardware Design and Verification Methods, Seiten 298–313, Springer-Verlag Berlin Heidelberg, url:citeseer.ist.psu.edu/article/clarke99program.html, 2004.

[1] Im Gegensatz zur Modellprüfung ist unser Slicing unabhängig von der verwendeten Bitbreite für Integer.

[2] Hier ließe sich mit der ZR-Metrik ein entsprechender Schwellwert bestimmen.

[St04] Stefan Edelkamp, StefanLeue, Alberto Lluch-Lafuente. *Partial-Order Reduction and Trail Improvement in Directed Model Checking*. International Journal on Software Tools for Technology Transfer, Band 6(4), Seiten 277–301, 2004.

[A. 97] A. Bouajjani, J. Esparza, O. Maler. *Reachability Analysis of Pushdown Automata: Application to Model-Checking*. In Proceedings of CONCUR 1997, Lecture Notes in Computer Science (LNCS) 1243, Springer-Verlag Berlin Heidelberg, 1997.

[E. 98] E. M. Clarke, O. Grumberg, M. Minea, D. Peled. *State Space Reduction using Partial Order Techniques*. Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213–3891, USA, Dept. of Computer Science, The Technion, Haifa 32000, Israel, Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974–2070, USA, 1998.

[Edm01] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. *Progress on the State Explosion Problem in Model Checking*. Lecture Notes in Computer Science (LNCS) 2000/2001, Seite 176, url:citeseer.ist.psu.edu/clarke00progress.html, ISSN 1611-3349, 2001.

[Fel06] Felix Berger. *A test and verifiation environment for Java programs*. Diplomarbeit Nr. 2470 der Abteilung Sichere und Zuverlässige Softwaresysteme, Institut für Formale Methoden der Informatik, Universität Stuttgart, 2006.

[Fle05] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of program analysis*. Korrigierte 2. Auslage, Springer-Verlag Berlin Heidelberg New York, 2005.

[FW02] Carsten Fritz und Thomas Wilke. State Space Reductions for Alternating Büchi Automata: Quotienting by Simulation Equivalences. In Manindra Agrawal und Anil Seth, Hrsg., *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science: 22nd Conference*, Jgg. 2556 of *Lecture Notes in Computer Science*, Seiten 157–168, Kanpur, India, 2002.

[FW05] Carsten Fritz und Thomas Wilke. Simulation Relations for Alternating Büchi Automata. *Theoretical Computer Science*, 338(1–3):275–314, 2005. Available online at url:http://authors.elsevier.com/sd/article/S0304397505000563.

[H. 01] H. Peng, Y. Mokhtari, S. Tahar. *Syntactic Model Reduction*. url:citeseer.ist.psu.edu/peng01syntactic.html, 2001.

[Jam04] James Ezick. *An Optimizing Compiler for Batches of Temporal Logic Formulas*. ISSTA'04, Boston, Massachusetts, USA, ACM 1-58113-820-2/04/0007, 2004.

[Jan01] Jan Obdrzálek. *Model Checking Java Using Pushdown Systems*. LFCS, Division of Informatics, The University of Edinburgh, 2001.

[Jav00] Javier Esparza, David Hansel, Peter Rossmanith, Stefan Schwoon. *Efficient Algorithm for Model Checking Pushdown Systems*. TUM-INFO-01-10002-0/1.-FI, Technische Universität München, Institut für Informatik, Sonderforschungsbereich (SFB) 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen, 2000.

[Jav01] Javier Esparza, Stefan Schwoon. *A BDD-based model checker for recursive programs*. Lecture Notes in Computer Science, Band 2102, Seiten 324–336, Springer, 2001.

[KT00] Klaus Havelund und Thomas Pressburger. *Model Checking Java Programs Using Java PathFinder*. International Journal of Software Tools for Technology Transfer (STTT), Band 2(4), Seiten 366–381, url:citeseer.ist.psu.edu/havelund98model.html, 2000.

[M. 84] M. Weiser, Peter Henderson, Jim Lyle, Glenn Pearson, Joan Shertz, Randall H. Trigg. *Program slicing*. IEEE Transactions on Software Engineering, SE-10(4), San Diego, California, United States, Seiten 352-357, 1984.

[M. 03] M. Christodorescu, S. Jha. *Static Analysis of Executables to Detect Malicious Patterns*. In Proceedings of the 12th USENIX Security Symposium, Seiten 169–186, url:citeseer.ist.psu.edu/christodorescu03static.html, 2003.

[Mat99] Matthew B. Dwyer, John Hatcliff. *Slicing Software for Model Construction*. In Partial Evaluation and Semantic-Based Program Manipulation, Seiten 105–118, Kansas State University, 1999.

[Mat06] Matthew Hague. *The use of Alternating Automata for the Efficient Model Checking of Linear-Time Logics*. TODO(unbekannt), 2006.

[McM93] K. McMillan. *Symbolic Moldel Checking*. Kluwer Academic Publishers, 1993.

[Pat96] Patrick Cousot. *Program analysis: the abstract interpretation perspective*. In ACM Workshop on Strategic Directions in Computing Research, MIT Lab. for Comput. Sci., Boston, Electronically available abstract in ACM Comput. Surv. 28A, 4, url: http://citeseer.ist.psu.edu/cousot96program.html, 1996.

[Ped06] Pedro de la Cámara, María del Mar Gallardo, Pedro Merino. *Abstract Matching for Software Model Checking*. in A. Valmari (Ed.): SPIN 2006, LNCS 3925, Seiten 182–200, Springer-Verlag Berlin Heidelberg, 2006.

[Rad01] Radu Iosif, Marius Bozga, Claudio DeMartini, Claudio Demartini, Matthew B. Dwyer, John Hatcliff, Yassine Laknech, Riccardo Sisto. *Exploiting Heap Symmetries in Explicit-State Model Checking of Software*. Automated Software Engineering, Proceedings of the 16th IEEE international conference on Automated software engineering Computer Society Washington, DC, USA, Seite 254, ISSN 1527-1366, 2001.

[Rei00] Reinhard Wilhelm, Shmuel Sagiv, Thomas W. Reps. *Shape Analysis*. Compiler Construction, 9th International Conference, CC 2000, European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, 25. März - 2. April, Proceedings, Lecture Notes in Computer Science (LNCS), Band 1781, 2000.

[Ste06] Dejvuth Suwimonteerabuth Stefan Kiefer, Stefan Schwoon. *Introduction to Remopla*. Institute of Formal Methods in Computer Science, University of Stuttgart, 2006.

[Tau03] H. Tauriainen. *On translating linear temporal logic into alternating and nondeterministic automata*. Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, Research Report A83, 2003.

[Tom00] Tom Ball, Sriram Rajamani. *A symbolic model checker for Boolean programs*. In SPIN Workshop 2000, Lecture Notes in Computer Science (LNCS) 1885, Seiten 113–130, Springer-Verlag, 2000.

[Wol04] Wolfgang Thomas, Christof Löding. *Model-Checking*. Vorlesung WS03/04, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Lehrstuhl für Informatik 7, http://www-i7.informatik.rwth-aachen.de/teaching/ws0304/modelchk/, 2004.

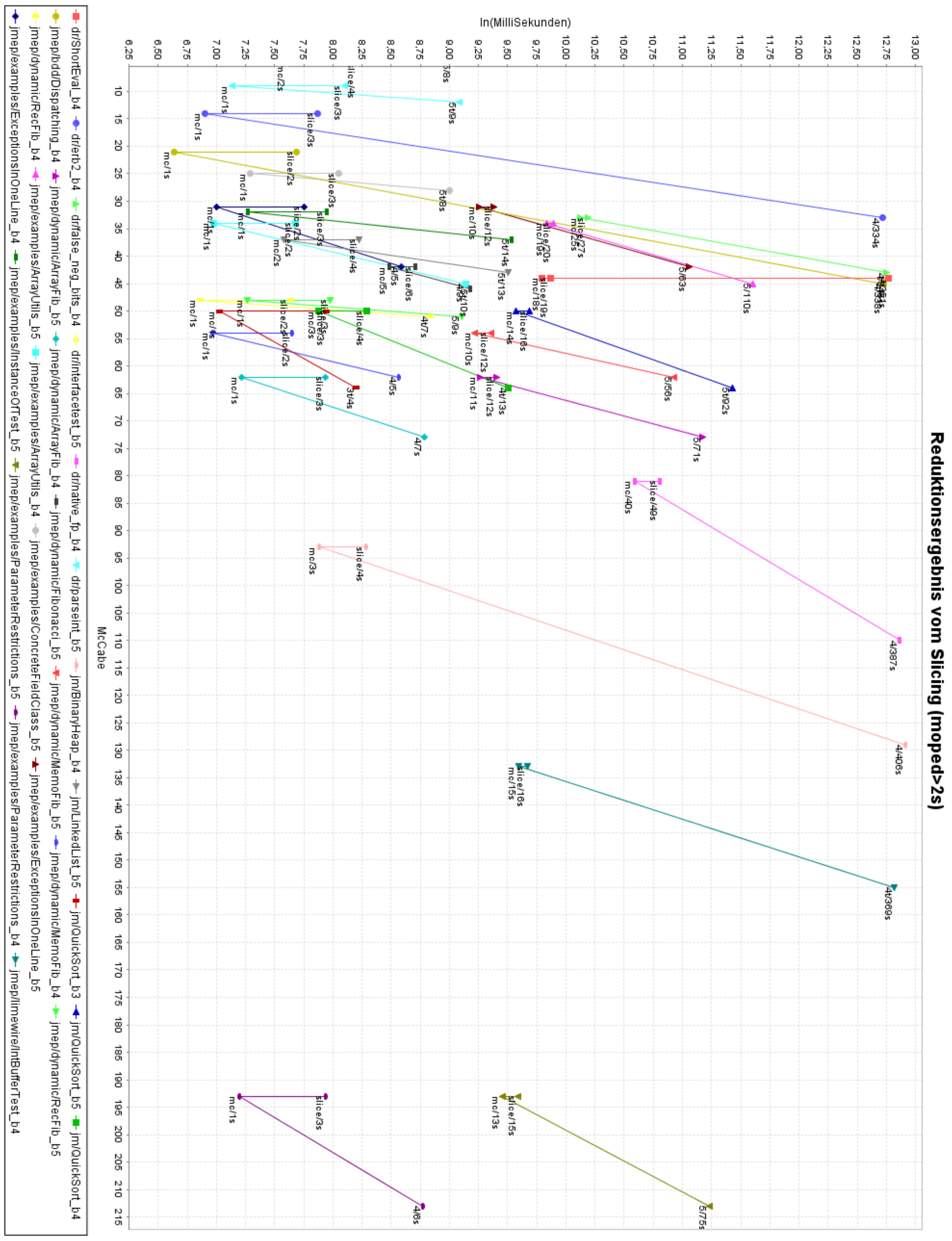


Abbildung 3: Reduktionsergebnis des Slicings in denen Moped mehr als 2 Sekunden benötigte.

Verifying Concurrent List–Manipulating Programs by LTL Model Checking

Joost–Pieter Katoen, Thomas Noll, and Stefan Rieger

RWTH Aachen University
Software Modeling and Verification Group
52056 Aachen, Germany
{katoen,noll,rieger}@cs.rwth-aachen.de

Abstract:

Techniques for the verification of elementary properties of concurrent pointer programs are indispensable. Programming with pointers is error–prone with potential pitfalls such as dereferencing null pointers and the creation of memory leaks. Pointer programming becomes even more vulnerable in a concurrent setting where data structures such as linked lists and trees are manipulated and inspected by several threads.

We present a model–checking approach to the verification of concurrent programs that manipulate singly–linked lists.

Our approach is illustrated by considering a simple concurrent programming language that besides the usual control structures offers primitives for pointer manipulation, cell creation and destruction, and (guarded) atomic regions that allow concurrency control constructs such as test–and–set primitives and monitors. An operational semantics is provided in terms of labeled transition systems in which states are equipped with a graph structure representing the current list configuration. List abstraction exploits a variant of summary nodes [→ Sagiv et al.] that represent more than M chained list cells where constant M is directly obtained from the formula to be checked. Each configuration is shown to have a canonical representation (up to isomorphism). The abstract semantics of any concurrent program in our language is finite, obtained in a fully mechanized manner, and keeps the minimal "distance" between program variables and summary nodes invariant. Over–approximation occurs in a very controlled manner; only assignments may yield nondeterminism as variables may get "too close" to summary nodes.

Properties are expressed in a first–order linear–time temporal logic (LTL) that is enriched with assertions on singly–linked lists such as reachability of cells, aliasing, and freshness of cells. Our logic is similar in spirit to NTL [→ Distefano et al.] and ETL [→ Sagiv et al.]. Opposed to NTL, we avoid the use of temporal operators inside quantification. In this way, involved mechanisms to keep track of the identities of individual cells are not needed. As a result, standard LTL model checking algorithms can be employed. The differences with ETL are more of a technical nature. ETL has a three–valued interpretation, whereas our logical interpretation is a standard binary one. Moreover, ETL–formulas are translated in first–order logic with transitive closure for the evaluation on a trace, whereas in

our case traces are generated by labeled transition systems and used in standard LTL model checking.

Combining Tools and Languages for Static Analysis and Optimization of High-Level Abstractions

Markus Schordan

Vienna University of Technology, Austria
markus@complang.tuwien.ac.at

Abstract. We present an approach for combining different analysis and transformation tools that enables their application to popular programming languages without extending existing compilers. Analysis results are made available as annotations of a common high-level intermediate representation and as generated source code annotations. We also support an external file format. The presented Static Analysis Integration Engine allows the selection of an arbitrary tool chain from the pool of integrated tools, most suitable for a certain program analysis or manipulation task. The architecture is evaluated with an implementation targeting full C++, considering templates, object-oriented features, as well as low-level features. The integrated tools are the LLNL-ROSE source-to-source infrastructure, the Program Analyzer Generator from AbsInt, and the language Prolog for manipulating terms representing C/C++ programs.

1 Motivation

For instrumentation tools, source-to-source optimizers, slicing tools, refactoring tools, and tools for enabling code comprehension, it is important to keep the source-code structure available for presenting the results of source code manipulating operations to the user. It is important that the results can be easily put into relation to the original program. This aids the user of such a tool, but complicates the internal handling of the source program during analysis and transformation because the results must be mapped back to the original program. Compilers usually translate the input programs to a lower-level representation for reducing the number of different language constructs, allowing to keep a program analysis more compact. The presented approach aims at utilizing compiler technology

but without losing syntactic or semantic information about the original input program. Therefore all tools are integrated to operate on, or map forth and back to a high-level intermediate representation. The goal is to permit building arbitrary tool chains from the pool of integrated tools.
In Section 2 we present the architecture of our Static Analysis Tool Integration Engine (SATIrE) allowing a seamless integration of powerful tools. The concrete implementation is presented in Section 3, also describing each tool and how it is integrated in SATIrE. In Section 4 we discuss related tool-based infrastructures and in in Section 5 we provide a short overview of the perspectives that we anticipate for the extensibility of our approach.

2 Architecture

The architecture of the Static Analysis Tool Integration Engine (SATIrE) is shown in Fig. 1. An essential aspect is that information gathered about an input program can be generated as annotation in the output program, and that the output program can again serve as input program. This allows to make analysis results persistent as generated source-code annotations. Utilizing such annotations, it allows to perform whole program optimization.
The architecture shown in Fig. 1 consists of the following kinds of components

Front End. The input language, L, is translated to a high-level intermediate representation (HL-IR).

Annotation Mapper. The annotations in L are translated to annotations of the HL-IR.

Tool IR Builder. Each tool may require its own IR. The Tool-IR Builder creates the required Tool-IR by translating the HL-IR to the Tool-IR.

Tool. A tool analyzes or transforms its respective Tool-IR.

Tool IR Mapper. The Tool-IR mapper either maps the Tool's IR back to High-Level IR or maps the computed information or results back to locations in the HL-IR.

Program Annotator. The HL-IR annotations are translated to a representation in the source code. This can be comments, pragmas, or some specific language extension.

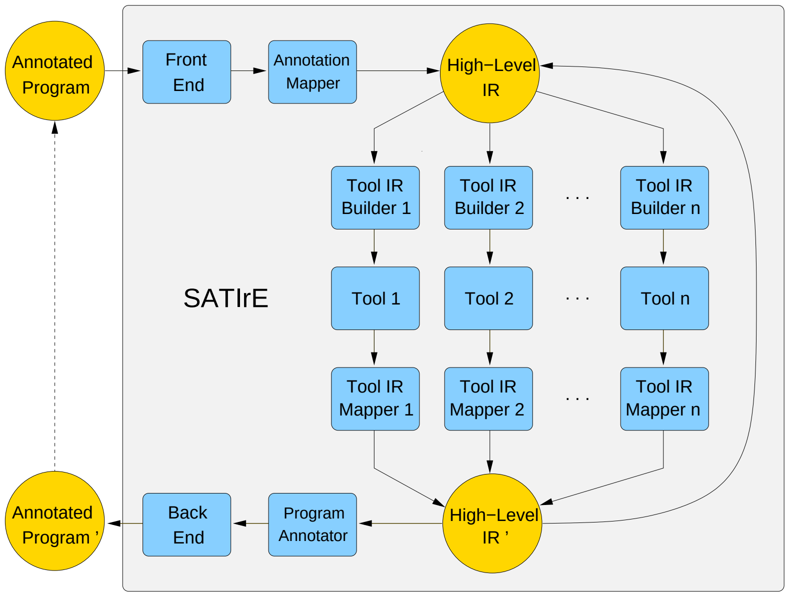


Fig. 1. Static Analysis Tool Integration Engine Architecture

Back End. From the HL-IR a program in language L is generated (including annotations).

To allow a seamless integration of the tools, the Annotation Mapper, Program Annotator, the Tool-IR Builders and Tool-IR Mappers are offered by SATIrE. In Fig. 1 the solid back-edge represents an iterative application of the tools within SATIrE.
For example, library source codes can be analyzed and the library's interface source code can be annotated with analysis results. When the library is used by an application, the library annotations can then be utilized by the application optimizer. We have demonstrated the optimization of the use of a parallel C++ array abstraction and achieved similar performance as with an equivalent Fortran implementation [9].

3 Integrated Tools and Languages

To date we have integrated the Program Analyzer Generator PAG [7], which generates analyzers from high-level specifications, the LLNL-ROSE infrastructure for source-to-source transformation of C++ programs [12], and a term representation of programs suitable for a Prolog interpreter, into SATIrE. In the following sections we describe each integrated tool and give a short overview of its integrated components.

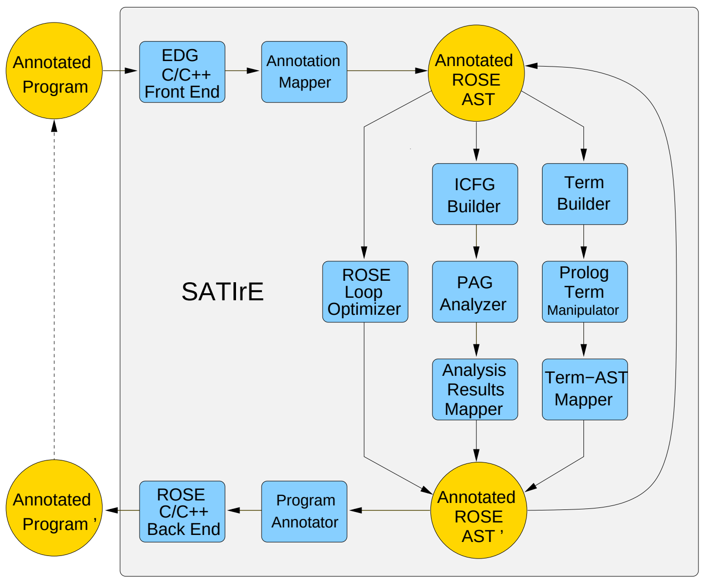


Fig. 2. Static Analysis Tool Integration Engine Implementation

3.1 LLNL-ROSE Integration

The LLNL-ROSE infrastructure offers several components to build a source-to-source translator. The ROSE components integrated into SATIrE are

C/C++ Front End. ROSE uses the Edison Design Group C++ Front End (EDG) [3] to parse C++ programs. The EDG Front End generates an abstract syntax tree (AST) and performs a full type evaluation of the C++ program. The AST is represented as a C data structure. ROSE translates this data structure into a decorated object-oriented AST (ROSE-AST).

Abstract Syntax Tree (ROSE-AST). The ROSE-AST represents the structure of the input program. It holds additional information such as the type information for every expression, exact line and column information, instantiated templates, the class hierarchy (as it can be computed from the input files), an interface that permits querying the AST, an an attribute mechanism for attaching user-defined information to AST nodes.

C/C++ Back End. The Back End unparses the AST and generates C++ source code. It can be specified to unparse all included (header) files or the source file(s) specified on the command line with include-directives. This feature is important when transforming user-defined data types.

Loop Optimizer. The loop optimizer was ported by Qing Yi from the Fortran-D compiler to directly operate on the ROSE-AST. It supports a wide range of loop transformations such as loop fusion, loop fission, loop skewing, loop interchange and blocking that can be applied to a given ROSE-AST.

3.2 Program Analyzer Generator Integration

The Program Analyzer Generator (PAG) from AbsInt, takes as input a specification of a program analysis and generates an analyzer that implements the analysis. The analyzer operates on an inter-procedural control flow graph (ICFG) and provides the computed analysis results as C data structure as well as a visualization of the ICFG and the analysis results. The components necessary for a seamless integration of PAG into SATIrE are

ICFG Builder. Creates the inter-procedural control flow graph (ICFG) for a given ROSE-AST.

PAG Analyzer. Generated by the Program Analyzer Generator (PAG) from a user-defined analysis specification using the OPTLA language.

Analysis Results Mapper. Maps the analysis results back to locations in the ROSE-AST and makes them accessible as ROSE-AST annotations.

Various types of ICFG attributes (for example numeric labels for statements) and support functions are provided to the analyzer by appropriate functions. Thus, the high-level analysis specification can access any information the ROSE-AST provides, such as types of expressions, the class hierarchy, etc.

3.3 Example

A short example output of an automatically annotated program is shown for the post-processed results of a shape analysis [10] in Fig. 3. The shape analysis is specified using PAG, the input program is a C++ program implementing a list reversal (and other list operations). After translating the C++ program to the corresponding ROSE-AST, SATIrE's ICFG builder creates the ICFG. Then the PAG analyzer performs the shape analysis and the Analysis Results Mapper maps the results back to the ROSE-AST. A post-processing of the computed shapes generates may and must alias information. The aliasing results are attached to the ROSE-AST nodes as must/-may alias annotations. The Program Annotator generates from the AST the annotations as source code comments, and the ROSE Back End generates the annotated C++ code.
The actual parameter in the call to the function `reverseList` is `l`, and is therefore aliased with the formal parameter `x`. When post analysis information (after a statement) and pre analysis information (before a statement) is the same, it is shown in the same line and preceded with `post,pre`.

3.4 Prolog Integration

The integration of Prolog allows to specify a manipulation of the AST as term manipulation. The SATIrE components necessary for integration are

Term builder. Creates a term representation for a given AST. The term representation is complete, meaning that it contains all information available in the AST. The term representation is stored in an external file.

Prolog term manipulator. The term manipulation is specified as Prolog rules.

Term-AST Mapper. The transformed term is read in and translated to a ROSE-AST.

```
class List* reverseList(class List* x)
{
  // pre must_aliases : {(l,x)}
  // pre may_aliases : {(l,x)}
  class List* y;
  // pre,post must_aliases : {(l,x)}
  // pre,post may_aliases : {(l,x)}
  class List* t;
  // post,pre must_aliases : {(l,x)}
  // post,pre may_aliases : {(l,x)}
  y = ((0));
  // post must_aliases : {(l,x)}
  // post may_aliases : {(l,x)}
  // pre must_aliases : {}
  // pre may_aliases : {(l,t),(l,x),(l,y),(l,y->next),(t,y->next)}
  while(x != ((0))) {
    // pre must_aliases : {}
    // pre may_aliases : {(l,t),(l,x),(l,y),(l,y->next),(t,y->next)}
    t = y;
    // post,pre must_aliases : {(t,y)}
    // post,pre may_aliases : {(l,t),(l,x),(l,y),(l,y->next),(t,y)}
    y = x;
    // post,pre must_aliases : {(x,y)}
    // post,pre may_aliases : {(l,t),(l,x),(l,y),(x,y)}
    x = (x -> next);
    // post,pre must_aliases : {(x,y -> next)}
    // post,pre may_aliases : {(l,t),(l,y),(x,y->next)}
    y -> next = t;
    // post must_aliases : {}
    // post may_aliases : {(l,t),(l,y),(l,y->next),(t,y->next)}
  }
  // post,pre must_aliases : {}
  // post,pre may_aliases : {(l,t),(l,x),(l,y),(l,y->next),(t,y->next)}
  t = ((0));
  // post,pre must_aliases :  {}
  // post,pre may_aliases : {(l,x),(l,y),(l,y->next)}
  return y;
  // post must_aliases : {}
  // post may_aliases : {(l,x),(l,y),(l,y->next)}
}
```

Fig. 3. Example of a C++ program, annotated automatically with must/may aliasing information which is computed by a post-processing phase from the results of a shape analysis [10]. We extended the shape analysis to an inter-procedural analysis. The analysis is specified by using PAG's specification language.

This approach has been successfully adopted within the COSTA project for performing Worst-Case Execution Time Analysis for a given C program. A detailed description can be found in [8].

4 Related Work

Glynn et al. show that support for program understanding in development and maintenance tasks can be facilitated by program analysis techniques [4]. They outline the addition of generic program analysis support to a generic, language-based software development environment.

Harrold and Rothermel present a technique for separate analysis of modules [5]. The work focuses on one particular analysis, inter-procedural may alias analysis, but the design of the analyzer is general and similar to our setting. For inter-procedural analysis an ICFG is created. The separation in control flow and intermediate representation of statements and expressions is the same as in our approach. The analysis is a modular analysis, meaning that a module is a set of interacting procedures or a single procedure that has a single entry point. The approach allows to reuse the analysis results after analyzing a module and thus, is applicable to large scale software and real world applications. In our approach we can add analysis results as annotations to source-code, allowing to reuse analysis results in a subsequent analysis step. This can either be done on the IR-level or the annotated source code is read in again.

For optimizing compilers the automatic generation of data flow analyses and optimizations out of concise specifications has been a trend for several years. The systems of [1,2] concentrate on "classical" inter-procedural optimizations, whereas the system of [13] is particularly well suited for local transformations based on data dependency information. We integrated PAG because it is a tool that allows to generate analyzers from specifications for similar analysis problems. In our infrastructure the transformation of the program is performed by utilizing the AST rewrite capabilities of ROSE and by using Prolog for term manipulation.

In [6] a technique is presented for automatically proving compiler optimizations *sound*, meaning that their transformations are always semantics-preserving. The domain specific-language Cobalt allows to specify optimizations to operate on a C-like intermediate representation. The implemented correctness checker interfaces with the automatic theorem prover Simplify. A similar setting could be added to our infrastructure by integrating also tools for checking and automatic proving into our current PAG-ROSE environment. Addressing the additional needs of such tools and leveraging its benefits is a driving force in the development of SATIrE.

5 Conclusions and Perspectives

We have presented SATIrE that allows to combine tools for analysis and transformation. The Front End translates the possibly annotated input program to a high-level representation (HL-IR). This HL-IR is translated to an appropriate Tool-IR for each integrated tool. The results computed by the respective tool are always mapped back to the common HL-IR. The HL-IR can be unparsed to annotated source code.
The applicability of our approach was demonstrated by integrating into SATIrE the program analyzer generator PAG, the LLNL-ROSE source-to-source translator, and by generating an external representation of the ROSE-AST as Prolog term. We are using SATIrE [11] in a lecture on optimizing compilers at TU Vienna since 2006. Currently we focus on specifying different kinds of pointer analyses for evaluation with respect to scalability, WCET analyses, and on design pattern detection and extraction. Other tools that we are presently integrating are Stratego and iburg. Tools of interest to be integrated in future are model checking tools and automatic theorem provers.
We aim at providing a platform of integrated tools for program analysis research of multi-million line applications. We hope that the use of high-level specification languages permits a qualitative comparison of analyses and that the analysis and transformation of real-world application codes permits a quantitative evaluation of program analyses at a broad range in future.

Acknowledgements. This work has been funded in part by the ARTIST2 Network of Excellence (http://www.artist-embedded.org). I wish to thank Dan Quinlan for the cooperation and fruitful joint work on LLNL-ROSE, Florian Martin for the support in integrating

PAG, Adrian Prantl for his work on maintaining the Prolog term representation, Jens Knoop for his support in integrating SATIrE in various research projects, and all students who have contributed in several SATIrE projects: Gergo Barany, Viktor Pavlu, Christoph Bonitz.

References

1. U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proceedings of the 6th International Conference on Compiler Construction* (*CC'96*) (*Linköping, Sweden*), Lecture Notes in Computer Science, vol. 1060, pages 121 – 135. Springer-Verlag, Heidelberg, Germany, 1996.
2. U. Aßmann. On edge addition rewrite systems and their relevance to program analysis. In *Proceedings of the 5th International Workshop on Graph Grammars and Their Application to Computer Science* (*GGTA'94*) (*Williamsburg*), Lecture Notes in Computer Science, vol. 1073, pages 321 – 335. Springer-Verlag, Heidelberg, Germany, 1996.
3. Edison Design Group. http://www.edg.com.
4. E. Glynn, I. Hayes, and A. MacDonald. Integration of generic program analysis tools into a software development environment. In *ACSC '05: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 249–257, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
5. M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Softw. Eng.*, 22(7):442–460, 1996.
6. S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231, New York, NY, USA, 2003. ACM Press.
7. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
8. A. Prantl. Source-to-source transformations for WCET analysis: The COSTA approach. In *24. Workshop der GI-Frachgruppe Programmiersprachen und Rechenkonzepte*, 2007.
9. D. Quinlan, M. Schordan, B. Miller, and M. Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 16, Issue 2-3:293–302, 2004.
10. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.
11. SATIrE. http://www.complang.tuwien.ac.at/markus/satire. Static Analysis Tool Integration Engine.
12. M. Schordan and D. Quinlan. Specifying transformation sequences as computation on program fragments with an abstract attribute grammar. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 97–106. IEEE Computer Society Press, 2005.
13. D. Whitfield and M. L. Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053 – 1084, 1997.

Design-by-Contract für funktionale Sprachen mit verzögerter Auswertung

Stefan Wehr [1]

[1] (in Zusammenarbeit mit Markus Degen und Peter Thiemann)
Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 079,
79110 Freiburg, Germany
{degen,thiemann,wehr}@informatik.uni-freiburg.de

1 Hintergrund

Design-by-Contract [2] ist eine Methodologie, um das Erstellen von korrekten Programmen zu erleichtern. Ein *Contract* ist dabei ein Prädikat, welches bestimmte Eigenschaften des Programms kodiert. Typischerweise wird die Gültigkeit von Contracts durch *Contract Monitoring* zur Laufzeit überprüft. Dabei sollen zwei Eigenschaft gelten: (1) Falls ein Programm keinen Contract verletzt, soll Contract Monitoring die Bedeutung des Programms nicht ändern. (2) Contracts sind idempotent, d.h. es ist egal ob ein Contract einmal oder mehrmals angewandt wird.

2 Problemstellung

Die ursprüngliche Umsetzungen von Design-by-Contract für funktionale Sprachen [1] ist im Umfeld von Scheme anzusiedeln, einer Sprache mit strikter Auswertung. Überträgt man nun diesen Ansatz auf eine Sprache mit verzögerter Auswertung, ergibt sich durch Contract Monitoring das Problem, dass Contracts möglicherweise auf noch unausgewertete Teile des Programms zugreifen. Um nun die beiden oben aufgeführten Eigenschaften nachzuweisen, genügt es daher für solche Sprachen nicht, lediglich gewisse Seiteneffekte wie etwa Nichtterminierung aus den Prädikaten der Contracts zu verbannen (was für strikte Sprachen ausreichend ist), sondern es müssen zusätzlich auch die Seiteneffekte der Argument eines Contracts (d.h. der Ausdrücke, die mit einem Contract versehen sind) eingeschränkt werden. Damit wird das Programmieren mit Contracts starkt eingeschränkt.

3 Lösung

Ausgehend von einem Typ- und Effektsystem für eine funktionale Sprache mit verzögerter Auswertung und Contracts wird die ursprüngliche, problematische Form des Contract Monitorings formalisiert und die angesprochenen Eigenschaften 1 und 2 nachgewiesen. Anhand des Beweises kann man nachvollziehen, welche Effekteinschränkungen nötig sind. Auf dieser Erfahrung aufbauend wird eine neue Form des Contract Monitorungs für Sprachen mit verzögerter Auswertung entwickelt, welche sich besser mit der Auswertungsstrategie verträgt, für praktische Zwecke gut geeignet ist und für die die genannten Eigenschaften gelten. Eine Haskell Implementierung für die neue Form des Contract Monitorings liegt vor.

Literatur

1. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Simon Peyton-Jones, editor, *Proc. Intl. Conf. Functional Programming 2002*, pages 48–59, Pittsburgh, PA, USA, October 2002. ACM Press, New York.
2. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

Tabular Expressions and Total Functional Programming

Baltasar Trancón y Widemann and David Lorge Parnas

Software Quality Research Laboratory (SQRL)
University of Limerick, Ireland
`http://www.sqrl.ie`

Abstract. Tabular expressions are a multidimensional structured notation for complex mathematical definitions of relations or functions. In order to create tools to check and evaluate tabular expressions, we have investigated functional programming as an implementation paradigm that reflects mathematical semantics faithfully. We explain why and how the restriction to total functions improves the semantic correspondence substantially, and describe the basic design and capabilities of our total functional programming tools for tabular expressions.

1 Introduction

1.1 Context

Our research group is developing methods of producing practical reference documentation for software products and components. Our document contents are defined by a relational model in which each document is required to be a representation of a specified relation. In effect, we are using mathematical descriptions of relations to provide specifications and descriptions of programs written in conventional programming languages.

Key to making these documents readable is a multidimensional form of expressions, which we call *tabular expressions* or just *tables*. These parse complex expressions into arrays of simpler expressions allowing readers to "look up" the information that they seek without understanding the whole expression.

Tools that check and evaluate these expressions would be very useful when these methods are applied and we are looking for effective and efficient implementations of such tools.

1.2 This Work

This paper reports on our experiences with applying the functional programming paradigm to the construction of tools for tabular expressions. Functional programming is a natural choice because

1. The tasks of checking and evaluating tabular expressions are typical examples of side-effect-free processing and interpretation of structured data.

2. The formal semantic model of tabular expressions, as presented to some degree in the earlier work [1] and more generically in the forthcoming [2], is given largely in terms of functions.
3. The intended application of these expressions is software documentation using a relational model [3] but the relations are described by their characteristic predicate and those are always functional.

Our intent is to give a reference implementation of the formal model that is not only executable, but also mirrors the intended semantics and the model's theoretical properties as faithfully as possible. We shall argue that our goals can almost, but not quite, be achieved by using a universal functional language, and describe an alternative.

1.3 Related Work

This is not the first time that the relation between tabular expressions and functional programming has been noticed or exploited. In [4], Kahl presents an inductive approach to tables of certain regular types that is compositional in table content and semantics at the same time. He provides an implementation of table constructors and inductive interpretation in Haskell, and corresponding formal proofs in the proof system Isabelle. Because of the restricted set of constructors, the resulting theory is compact and elegant.

In contrast, our current work is intended to implement the more generic table model of [2], that allows *all* constructs of a mathematical base language to be used freely in content and semantics of tables. This paper discusses the requirements of such a generic view, and presents preliminary results from the approach we have taken.

2 Example Tabular Expression

We shall use a simple tabular expression taken from [5] as the running example for explaining the basic usage of tables and the services we expect from an evaluation tool.

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

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

Table 1. Power Conditioning (Specification)

The tabular expression depicted as table 1 is a small, but real example.[1] It specifies a family of control functions of a nuclear reactor shutdown system. As some of the status monitoring logic is only applicable when the reactor is operating near its maximum output power level, they need to be "conditioned in" (activated) above a certain power level, and "conditioned out" (deactivated) below. To avoid *jitter* (many changes separated by very short intervals), hysteresis is simulated by setting the threshold for conditioning in (K_{in}) slightly higher than that for conditioning out (K_{out}). In between, the previous state (*Prev*) is maintained. A graph illustrating some change of power over time is depicted in figure 1. The relevant state transitions and their effects are marked.

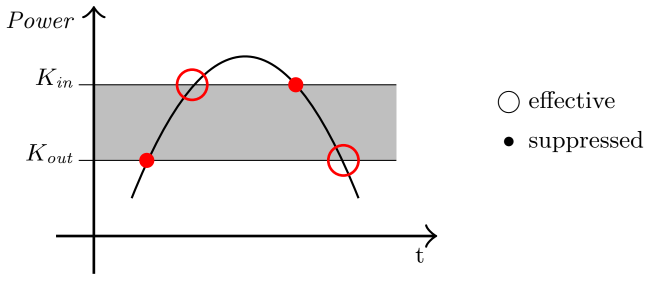


Fig. 1. Power Conditioning (Example Graph)

2.1 Meaning of a Tabular Expression

The concrete syntax for this table is deceivingly straightforward; for multidimensional or irregular tables, there may not be such an obvious graphical representation. Hence the mathematical table model only represents the abstract syntax of the *content* of the table as an indexed set (aka family or map) of *grids*. Each grid is in turn an indexed set of *cells*, each of which contains a (conventional or nested tabular) expression. A table *type* complements the content to make the tabular expression semantically self-contained. The table type, that may be shared by many similar tables, comprises

1. an *evaluation term*, i.e., an algorithm for evaluating the table's content, depending on a valuation of free variables,
2. a *restriction predicate*, i.e., a well-formedness condition that a table's content must satisfy for the evaluation algorithm to be applicable.

The table type is an integral part of the table expression. One can consider it as an instance of dynamic typing, or as semantically rigorous meta-data.

[1] Although we have chosen the simplest possible real example to illustrate these expressions, many much more complex tables were used in the inspection of the Darlington Nuclear Power Generation Station described in [5].

The given example table is an instance of the one-dimensional *normal function table* type:

1. It contains two grids of three cells each.
   (a) The upper grid is a *header* grid that contains predicate expressions.
   (b) The lower grid is a *main* grid that contains value expression (also of type *bool* in this case).
2. To evaluate the table, choose an index to the header grid, such that
   (a) the selected predicate expression evaluates to *true*,
   (b) then evaluate only the corresponding cell of the main grid.

   (For more than one dimension, one index for each header grid would be chosen independently.)
3. The table is well-formed, if
   (a) the main grid has the same indexes as the header grid (for higher dimensions, the index set of the main grid must be the Cartesian product of the index sets of the header grids), and
   (b) each header grid partitions the set of possible variable valuations, i.e., exactly one is found to be true in any case.

See [1, 2] for more exact definitions of the normal function table type and other types of tabular expressions.

2.2 Tool Requirements

We expect an evaluation tool to enable us to

1. evaluate a tabular expression for a given variable valuation, by applying the evaluation term specified by the table's type to its content,
2. check the restriction predicate, distinguishing two parts for practical reasons:
   (a) clauses that do not depend on variable values (called the *static* restriction), to be checked universally for the table's content,
   (b) clauses that do depend on variable values (called the *dynamic* restriction), to be checked specifically for the table's content and a given variable valuation,

all with reasonable efficiency.

We do not expect an evaluation tool to support checking dynamic restrictions universally for all possible variable valuations. This is a task for a theorem proving system, and may involve much more complex computations. In [5], based on earlier work [6], the authors show how a flaw in the specified table has been discovered by the automatic theorem prover PVS: the header cells are only a partition of the valuation space, if the (intuitive, but unstated) assertion $K_{out} < K_{in}$ holds. Otherwise, the first and third columns overlap, and the table does not specify a function.

3 The Logic Behind Tabular Expressions

If tabular expressions are to be used for describing real problems, they must be able to deal with partial functions. Partial functions can lead to undefined expressions, and there are many ways to handle undefinedness in logic, e.g., by having three or more truth values.

The meaning of partial functions in tabular expressions here is the one defined in [7]. It was chosen to give the shortest possible expressions in the table cells. It can be summarized as follows:

1. There is a special value $(*)$, distinct from all proper values of interest. This value is assigned to the application of a partial function to arguments outside its domain.
2. The domain of partial functions never contains $(*)$. This implies that the result of a partial function is $(*)$ whenever one of its arguments is $(*)$, i.e., functions are strict. A partial function is being treated as if it were a *total* function whose range includes $(*)$.
3. Predicates are treated differently from functions. A predicate is simply *false* if any of its arguments is $(*)$. Consequently, the truth value of a formula is always *true* or *false*, but never $(*)$. I.e., predicates are non-strict.

Note that $(=)$ is also a predicate, so (by the third rule) the seemingly trivially true predicate expression $f(x) = f(x)$ is *not* true if x is outside the domain of f. On the other hand, the equation $f(x) = y$ is logically equivalent to $F(x, y)$ where F is the characteristic predicate for f. It has been argued in [7] and later work that this interpretation of partial functions is particularly concise and useful for writing software descriptions and specifications in the tabular notation.

This semantic decision has consequences for the construction of an effective universal evaluation algorithm for tabular expressions. The intuitively appealing representation of $(*)$ by the element $(\perp)$ of standard domain-theoretic semantics does not work as intended: Since $(\perp)$ is also assigned to expressions that cannot be evaluated effectively, e.g., a nonterminating recursive function application, writing a program that would evaluate any predicate of the formalism becomes as hard as solving the halting problem, i.e., impossible without restrictions.

1. The *pragmatic* approach is to use a universal language to implement the model, accepting some semantic deviations. It is impossible to preclude undetermined predicate expressions in this case; so the responsibility is placed on the programmer to find the appropriate termination arguments.
2. The *rigorous* approach is to use a restricted language with the "right" semantics to implement the model. If we have to decide whether an expression evaluates to $(*)$, it has to be an proper value in a calculus of total functions. The advantage of this approach is that properties of the implementation are closely related to (and not much more complex to prove than) properties of the formal model. The price is that one has to obey the restrictions of the implementation language.

4 Total Functional Programming

In [8], Turner expresses similar, albeit more fundamental concerns regarding the relation of universal functional programming calculi and mathematical functions:

> *The driving idea of functional programming is to make programming more closely related to mathematics. [...] The existing model of functional programming [...] is compromised to a greater extent than is commonly recognized by the presence of partial functions.*

He strives for a language that abolishes partial functions, but retains as much as possible of the notational ease of Miranda or Haskell.

A quite different approach to total functions is taken by total function calculi in the style of Martin-Löf's type theory or Coquand's *calculus of constructions*. These are closely connected to higher-order logic (via the Curry-Howard isomorphism), a fact that is exploited in constructive proof systems like Coq.

We have chosen a "middle road", employing a rigorous explicit type system like the latter, but focusing on computation (rather than logic) like the former. The result is FCN[2], the design and implementation of a practical programming language for pure total functions. Like other total languages, it is characterized by the absence of general recursion: The syntax forbids recursive definitions, and the type system forbids fixpoint operators.

5 Functional Programming Techniques Applied

Limited space prohibits the detailed description of the FCN language. The following subsections can provide only brief examples of how our requirements have been mapped successfully onto features of the functional paradigm.

5.1 Partial Functions

The logical rules concerning partial functions and predicates can be implemented in a completely explicit way using a simple *error monad* [9].

1. For each type A, a *dubious* type $A?$ is defined to contain one additional element:
$$\texttt{type}\ A? = A + *$$
Readers familiar with Haskell will easily recognize this as the *Maybe* functor.
2. A partial function $f : A \nrightarrow B$ is represented as a total function $f' : A \to B?$. Consider another partial function $g : B \nrightarrow C$, totalized as $g' : B \to C?$. The composition $g' \circ f'$ is not type-correct, so a canonical transformation
$$\mathit{bind} : \forall\, B, C.\, (B \to C?) \to (B? \to C?)$$

[2] Functional Core Notation

is inserted. It satisfies the strictness law

$$bind(g')(x) = \begin{cases} * & x = * \\ g'(x) & x \neq * \end{cases}$$

such that the composition of total functions $bind(g') \circ f'$ correctly implements the composition of partial functions $g \circ f$. The operation *bind* can be extended to the functorial operation of (?) to deal with total functions:

$$lift : \forall\, B, C.\, (B \to C) \to (B? \to C?)$$

3. For predicates, a different canonical transformation

$$prim : \forall\, A.\, (A \to bool) \to (A? \to bool)$$

is used to compose them with partial functions. It satisfies the non-strictness law

$$prim(p)(x) = \begin{cases} false & x = * \\ p(x) & x \neq * \end{cases}$$

Apart from reflecting the intended semantics precisely, this approach has several additional benefits:

1. Algebraic simplification laws that do not hold for the original implicit notation are restored. These include the aforementioned $f(x) = f(x) \iff true$, as well as general β-reduction.
2. A single boolean-valued function can be re-used to define both a partial function and a total predicate, by exchanging *lift* and *prim*.
3. There is no ambiguity which symbols are primitive predicates and thus subject to the non-strictness rule: they are explicitly qualified with *prim*.

5.2 Cells and Variables

Tabular expressions are used to define functions and relations, hence they are likely to contain (free) variables. In a language with first-order functions, the grid structure of a table and the functionality of individual cells can be separated cleanly by *closure conversion*, aka *lambda lifting*: The open expression in each cell is turned into a function of the table's variables, which can then be stored in a data structure. In the given example table, the effect is that the phrase

$$\lambda\, Prev : bool; Power, K_{in}, K_{out} : real. \cdots$$

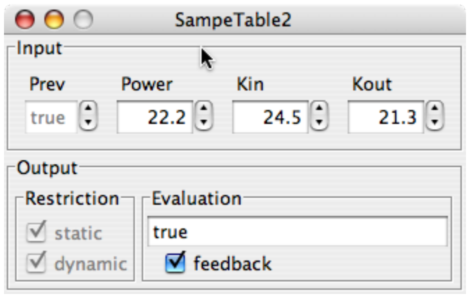
is prepended to each cell expression. A variable valuation then takes the form of an argument vector that is applied uniformly to all cells of the table.

5.3 Table Interpretation

Table content is structured as grids and cells, organized in list-like collections. All typical access operations required to define a table type, such as the normal function table type described above, are easily defined in terms of primitive recursion, applying a recursion operator (aka *fold*) to a non-recursive step function. So far, we have not encountered any problem in this specific domain that would have required recursion support from the language.

6 Tool Support

Programming in FCN is supported by tools, most notably a parser, type checker and compiler. All of these are written in Java. The compiler produces Java code that runs on the JVM, together with a small runtime library. Figure 2 shows a compiled version of the running example table. It is controlled by a GUI that is derived directly from the function signature, and will be generated automatically by a future version of the tools.



Prev: true · Power: 22.2 · Kin: 24.5 · Kout: 21.3

Restriction: [✓ static] [✓ dynamic]

Evaluation: true [✓ feedback]

Fig. 2. Power Conditioning (Simulation Screenshot)

A library of about 1000 lines of FCN code defines the ubiquitous basic types and operations: booleans, natural numbers, tuples, lists monads, etc. The powerful type system of the calculus of constructions allows the definition of all of these in terms of the λ operator only; no additional primitive constructs are needed or used.

A second level of library code, about 500 lines of FCN, defines the table model in terms of standard functional data structures and operations, as well as some common table types, including the multidimensional normal function table type used in the example. This library will be extended in the future to support other table types.

A tabular expression is simply data structured according to the model, containing functions at the cell level. Evaluation and restriction checking are completely generic operations, because all semantic information is explicit in the "type" part of the table data.

Tabular expressions that describe software behaviour can be "animated" with compiled FCN code to produce simulations, test oracles or prototypes.

6.1 Total Functions and Theorem Proving

There is ongoing work [10] to represent the formal model of tabular expressions as a theory in the proving system PVS. This would complement the services of the evaluation tools by allowing to prove properties of tables universally for a class of variable valuations.

Because of the close similarity between the calculi of total functions and the higher-order logic of PVS, and because of the explicit treatment of partiality issues in the FCN implementation, large parts of the implementation's design carry over to PVS directly. The FCN type checker has proved a valuable tool for quick consistency checking in the design process, helping to keep consistency proof obligations in the PVS theory tractable.

7 Conclusion

The work described in this paper is an experimental use of functional programming in the creation of software engineering tools. The approach has provided a formulation of the mathematical model of tabular expressions that can reflect semantics precisely, but is also directly and effectively executable. The strict type system has proven a valuable consistency check. The features of functional programming that are supposed to support abstraction and reuse in functional programming, namely parametric polymorphism and higher-order functions, have found essential use, e.g., as primitive recursion operators and monadic liftings.

We have also found that the notion of total functional programming, that is looked upon with some scepticism by most of the community, is quite feasible for this specific application. The absence of general recursion does not impede the construction or interpretation of tables unduly. The pervasive use of recursion operators even encourages a point-free programming style.[3]

The absence of infinite evaluation branches greatly simplifies the choice of, and encourages experiments with, evaluation strategies. This applies both at runtime, where no semantic difference between eager and lazy evaluation exists, and also at compile-time to program specialization by partial evaluation. Both cases are investigated in ongoing work.

Finally, we have found that a calculus of total functions greatly reduces the impedance mismatch between the implementation of a formalism and its formalization in a proof system, making it attractive for projects that involve both evaluation and verification.

[3] An obvious benefit from the viewpoint of the functional programmer, but of questionable merit for the software engineer.

Acknowledgements

Thanks to Dennis Peters, Mark Lawford and other SQRL members for helpful discussions.

References

1. Parnas, D.L.: Tabular representation of relations. CRL Report 260, McMaster University (1992)
2. Balaban, A., Bane, D., Jin, Y., Parnas, D.L.: Mathematical model of tabular expressions. SQRL draft (2007) available for review.
3. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Trans. Softw. Eng. **20**(12) (1994) 948–976
4. Kahl, W.: Compositional syntax and semantics of tables. SQRL Report 15, McMaster University (2003)
5. Lawford, M., Froebel, P., Moum, G.: Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Submitted to Formal Methods in System Design (2000)
6. Jing, M.: A table checking tool. SERG Report 384, McMaster University (2000)
7. Parnas, D.L.: Predicate logic for software engineering. IEEE Trans. Softw. Eng. **19**(9) (1993) 856–862
8. Turner, D.A.: Total functional programming. Universal Computer Science **10**(7) (2004) 751–768
9. Spivey, M.: A functional theory of exceptions. Sci. Comput. Program. **14**(1) (1990) 25–42
10. Peters, D.K., Lawford, M., Trancón y Widemann, B.: An IDE for software development using tabular expressions. In: CASCON 2007. (2007) to appear.

Monadic, Lazy Assertions in Haskell[⋆]

Frank Huch
CAU Kiel, Germany
fhu@informatik.uni-kiel.de

Olaf Chitil
University of Kent, UK
oc@kent.ac.uk

Abstract. Assertions test expected properties of run-time values without disrupting the normal computation of a program. We present a library for enriching Haskell programs with assertions. Expected properties can be specified in a parser-combinator like language. The assertions are lazy: they do not force evaluation but only examine what is evaluated by the program. They are also prompt: assertion failure is reported as early as possible. The implementation is based on lazy observations and continuation-based coroutines.

1 Introduction

Assertions are parts of a program that, instead of contributing to the functionality of the program, express properties of run-time values the programmer expects to hold. It has long been recognised that augmenting programs with assertions improves software quality. An assertion both documents an expected property (e.g. a pre-condition, a post-condition, an invariant) and tests this property at run-time. For example, an assertion may express that the argument of a square root function has to be positive or zero and likewise the result is positive or zero. Assertions can be an attractive alternative to unit tests. Assertions simplify the task of locating the cause of a program fault: in a computation faulty values may be propagated for a long time until they cause an observable error, but assertions can detect such faulty values much earlier.

We can easily define a combinator for attaching assertions to a Haskell expression:

```
assert :: Bool -> a -> a
assert b x = if b then x else error "Assertion failed."
```

The assertion is an identity function when the expected property holds, but raises an exception otherwise[1].

We can define normal Haskell functions for our expected properties, e.g.

```
ordered :: Ord a => [a] -> Bool
ordered []       = True
ordered [_]      = True
ordered (x:y:ys) = x<y && ordered (y:ys)
```

⋆ This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-2 and by the United Kingdom under EPSRC grant EP/C516605/1.

[1] The Glasgow Haskell Compiler provides a variant that produces a more informative error message that includes the source location of the failed assert call

and use them to assert for example a pre-condition:

```
checkedInsert :: Ord a => a -> [a] -> [a]
checkedInsert x xs = assert (ordered xs) (insert x xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x < y then x:y:ys else y : insert x ys
```

In many applications this works fine

```
> checkedInsert 4 [1,3,2,5]
Assertion failed.
```

but sometimes it does not, as the non-terminating expression shows:

```
> take 4 (checkedInsert 4 [1,2..])
```

In our example the function `ordered`, which expresses our expected property, is fully strict and thus forces evaluation of the whole infinite list. Programming with assertions as above results in strict programs and thus a loss of the expressive power of laziness, for example, the use of infinite data structures and cyclic definitions. As long as an assertion does not fail, a program augmented with assertions should have exactly the same input/output behaviour as the one without assertions. Hence assertions for a lazy language should be lazy, that is, a property should only be checked for the part of a data structure that is evaluated during the computation anyway.

Our example above also demonstrates that using Boolean functions for specifying properties is rather limiting in expressiveness. We want to say that any list containing two neighbouring elements in the wrong order should raise an assertion failure, also when most of the rest of the list has not been evaluated. However, `ordered` only decides on totally evaluated finite lists. We present a parser-combinator like monadic language for expressive lazy assertions. Parser combinators are a well-known tool for describing a set of token sequences. Similarly our assertion combinators describe a set of possibly partial expected values.

Whenever a part of a value is evaluated that violates an asserted property, the assertion immediately fails. We say our assertions are *prompt*. Promptness ensures that the reported unexpected value is as unevaluated as possible and thus smaller to read. Furthermore, a program fault usually violates many assertions, but promptness ensures that the assertion that is closest to the fault with respect to data flow is reported. In summary, our assertions have the following properties:

– Lazy: They do not modify the lazy behaviour of a program.
– Prompt: The violation of an assertion is reported as early as possible, before a faulty value is used by the surrounding computation.
– Expressive: Complex properties can be expressed using full Haskell.
– Portable: Assertions are implemented as a library and do not need any compiler or run-time modifications; the only extension to Haskell 98 used for the implementation are `unsafePerformIO` and `IORef`s.

This papers sketches some first ideas, how such assertions can be implemented as a Haskell library. For lack of space, the paper concentrates on the basic ideas and does not explain how promptness can be added to the presented assertions.

2 Using the Assertion Monad

Expected properties are specified in an assertion monad `Try a` that combines pattern matching and non-deterministic computations. The combinators are used very similarly to standard monadic parser combinators [8].

Here is the specification of the ordered property discussed in the Introduction:

```
ordered :: Ord a => Lazy [a] -> Try ()
ordered xs = pNil xs
        ||| (do (_,ys) <- pCons xs; pNil ys)
        ||| (do (x,ys) <- pCons xs; (y,_) <- pCons ys;
                ((do rx <- pVal x; ry <- pVal y; guard (rx < ry))
                 &&& ordered ys))
```

The tested argument is wrapped within a new type constructor `Lazy` and the result type has to be `Try ()`. Together these two types enable prompt and lazy evaluation of assertions. To specify the three different cases for lists of length zero, one, and longer lists, the assertion monad `Try a` provides the non-deterministic choice operator `(|||) :: Try a -> Try a -> Try a`. It provides fair evaluation, that is, there is no fixed order in which the different cases are evaluated. Similarly, we provide a fair, parallel conjunction operator `(&&&) :: Try () -> Try () -> Try ()`, which allows the test of the ordered property in every position within the list.

For pattern matching we provide the following pattern combinators within the assertion monad:

```
pNil :: Lazy [a] -> Try ()
pCons :: Lazy [a] -> Try (a,[a])
pVal :: Lazy a -> Try a
```

For each data constructor we provide a pattern combinator that matches only the constructor and that yields the sub-structure as a tuple within the `Try` monad. For example, for the empty list it returns the empty tuple and for `(:)` it returns a pair consisting of the element and the remaining list. The combinator `pVal` matches every value and directly corresponds to a variable in a Haskell pattern. Finally, the function `guard` is the standard Haskell function that integrates a Boolean test into a `MonadPlus`.

To attach an assertion to an expression we provide the function `assert :: String -> (Lazy a -> Try ()) -> a -> a`. The first parameter is a label naming the assertion. When an assertion fails, the computation aborts with an appropriate message that includes the assertion's label. As further parameters `assert` takes the property and the value on which it behaves as a partial identity.

For expected values an assertion is an identity function. For partial values that are smaller than expected values (in the ordering where unevaluated/undefined is less than any value) the assertion cannot be decided and hence it is also the identity function. For any unexpected value the assertion raises an exception.

To prevent an assertion from evaluating too much, the property has to be defined as a predicate on the tested data structure. The implementation of `assert` ensures that only the context in which the application of `assert` appears determines how far the tested data structure is evaluated and only that part is passed to the predicate.

```
insertWithPre :: (Ord a,Observe a) => a -> [a] -> [a]
insertWithPre x xs = insert x (assert "insert input ordered" ordered xs)
```

The assertion is evaluated in a prompt, lazy manner, as the following call shows:

```
> take 4 (insertWithPre 4 ([3,4] ++ [1,2..]))
[3,4,*** Assertion (insert input list ordered) failed: 3 :4:1: _
```

Beside reporting the failed assertion, we also present the wrong value to the user and highlight that parts of the data structure that contribute to the failure. Here these are, beside the unordered values, all `(:)` constructors above the unordered values, because the assertion would not have failed if any of them was `[]`.

Similar to this precondition, we can add a postcondition specifying that the result of `insert` is ordered. However, this is not exactly what one would like to specify as a property of `insert`. In case `insert` is called with an unordered list, this fault should not be blamed on `insert`, but on the function applying `insert` to an unordered list. A better specification for `insert` would be: if the argument list is ordered, then the result is ordered as well. In contrast to the first assertion, this property is defined for a function. It specifies properties for an argument and the result. Functional assertions can be expressed by means of function `fun`n [2] for functions of arity n:

```
insertChecked :: (Ord a, Observe a) => a -> [a] -> [a]
insertChecked = assert "insert preserves ordered property"
                       (fun2 (\ _ ys zs -> ordered ys ==> ordered zs))
                       insert
```

To express the dependence between the two ordered properties, we can use an implication which is simply defined by negation `notAssert` and disjunction `(|||)`. Executing `insertChecked` yields the following behaviour:

```
> insertChecked 3 [5,3,4]
[5,3,3,4]
> insertChecked 3 [2,3,4]
[2,3,*** Assertion (insert preserves ordered property) failed:
3 -> ( 2:3:4:[] -> 2 :3:3: 4:_)
```

In the second case highlighting shows that for the ordered input list [2,3,4] the duplicate occurrence of `3` in the result list does not meet the specification. To correct the program, we could omit duplicated elements.

[2] `fun2 :: (Lazy a -> Lazy b -> Lazy c -> Assert) -> Lazy (a -> b -> c)`
`        -> Assert`

3 The Idea of Respecting Laziness

In this section we outline how the two types `Try a` and `Lazy a` enable Haskell computations to respect how far their argument values are evaluated. To represent which parts of a data structure are evaluated already we introduce the data type:

```
data EvalTree = Eval [EvalTree] | Uneval
```

An `EvalTree` represents how far a corresponding data structure is evaluated. It has the same tree structure as the data structure itself except that parts may be cut of by means of the constructor `Uneval`; that is, if the data structure contains an n-ary evaluated constructor, then the corresponding `EvalTree` contains an `Eval` node with n `EvalTree`s in the argument list. For instance, the evaluation of list `[1,2,3]` in the call of `[1,2,3]!!1` is represented by the `EvalTree`: `Eval [Uneval,Eval [Eval [],Uneval]]`. Note that in later sections we will refine the definition of `EvalTree` further.

Now we can introduce the type synonym

```
type Lazy a = (EvalTree,a)
```

in which values are paired with their corresponding evaluation information. This type enables us to define an assertion that respects the evaluation state of the tested value, for example a function `checkOrdered` that checks whether a given list is ordered with respect to its evaluated parts:

```
checkOrdered :: Lazy [Int] -> Maybe Bool
checkOrdered (Eval [], [])           = Just True
checkOrdered (Eval [_,Eval []], [_]) = Just True
checkOrdered (Eval [eX,eYXs@(Eval [eY,eXs])], (x:yxs@(y:xs))) =
                         leq (eX,x) (eY,y) &|& checkOrdered (eYXs,yxs)
checkOrdered _                       = Nothing

leq :: Lazy Int -> Lazy Int -> Maybe Bool
leq (Eval [],x) (Eval [],y) = Just (x <= y)
leq _           _           = Nothing

(&|&) :: Maybe Bool -> Maybe Bool -> Maybe Bool
(Just True)  &|& (Just True)  = Just True
(Just False) &|& _            = Just False
_            &|& (Just False) = Just False
_            &|& _            = Nothing
```

The result type of `checkOrdered` reflects that beside being ordered or not, there is a third alternative (`Nothing`), namely that at this stage of evaluation it is not possible to decide whether the list is ordered or not. For comparing two elements of the list we use a variation of `(<=)`, which also respects the `EvalTree`. Finally, the results of each comparison of two elements is done by a modified version of `(&&)`. Besides using the extended type `Maybe Bool` this function also implements a parallel version of `(&&)` by means of its third rule. Independent of the other argument, `(&|&)` propagates `Just False` as a result.

How can this approach be generalised to arbitrary computations on lazy values? Although, assertions have to return Boolean values as result, subcomputations may return other result types. Here we can also use the `Maybe` type to express that we either obtain a result or have a suspension. But how can the parallel evaluation in the `(&|&)` function be generalised? The key idea is to introduce non-determinism to our framework. In principle, the function `checkOrdered` can be seen as a non-deterministic predicate, which compares arbitrary successive elements within the list. Then, used as an assertion, the function only yields `Just True`, if all non-deterministic checks yield `Just True`; it yields `Just False` if one of the non-deterministic checks yields `Just False` and it *suspends* if some checks cannot be performed, that is, yield `Nothing`, and all other checks yield `Just True`. Hence, we define the generalised result type for computation on lazy values as follows:

```
newtype Try a = Try [Maybe a]
failT = Try []
suspT = Try [Nothing]
```

The non-determinism is encoded by a list of possible results from which each single result may not be computable because of insufficient evaluation. The type constructor `Try` forms a monad in which functions are applied to all list elements.

```
instance Monad Try where
  (Try as) >>= f = Try $ concatMap (applyRes (fromTry . f)) as
    where fromTry (Try x) = x
          applyRes :: (a -> [Maybe b]) -> Maybe a -> [Maybe b]
          applyRes f (Just x) = f x
          applyRes f Nothing = [Nothing]

  return x = Try [Just x]
```

For non-deterministic branching we define a parallel disjunction operator, which collects all possible results[3]:

```
(|||) :: Try a -> Try a -> Try a
(Try xs) ||| (Try ys)  = Try (xs++ys)
```

Within the `Try` monad we can now define pattern combinators for matching lazy values, e.g.:

```
pCons :: Lazy [a] -> Try (Lazy a,Lazy [a])
pCons (Eval [eX,eY],v) = case v of (x:xs) -> return ((eX,x),(eY,xs))
                                   _      -> failT
pCons (Eval _,_) = failT
pCons (Uneval,_) = suspT
```

[3] In fact, `Try` can also be made an instance of `MonadPlus` with `mplus = (|||)` and `mzero = failT`.

```
pNil :: Lazy [a] -> Try ()
pNil (Eval _,v) = if null v then return () else failT
pNil (Uneval,_) = suspT
```

These pattern combinators respect the evaluation of a given argument. If the argument is not evaluated at all, then the result is a suspension `Nothing`. If the constructor is evaluated and it is the wrong constructor, then matching fails. Finally, if the constructor matches, then we succeed and return the sub-terms together with their evaluation information. Similarly we define a pattern combinator that strictly matches any value.

```
pVal :: Lazy a -> Try a
pVal (et,v) = condEval et (return v)

condEval :: EvalTree -> a -> a
condEval (Eval ets) tv = foldr condEval tv ets
condEval Uneval _ = suspT
```

It is mostly used for flat data types such as `Int` or `Char`.

Next we define the parallel `(&&)` function within our framework. We start with a more general function, which applies arbitrary result functions to `Try` results:

```
(***) :: Try (a -> b) -> Try a -> Try b
(***) (Try fs) (Try xs) = Try [res | fRes <- fs, xRes <- xs,
                                     let res = do f <- fRes
                                                  x <- xRes
                                                  return (f x)]
type Assert = Try ()
(&&&) :: Assert -> Assert -> Assert
t1 &&& t2 = (return (\x1 x2 -> ()) *** t1) *** t2
```

Whereas our old `(&|&)` on type `Maybe Bool` could produce only one of three values, the new `(&&&)` may produce a value representing many successful and suspended computations. Now it is possible to define the `ordered` assertion from Section 2. For a first implementation it remains to show how the `EvalTree` can be successively constructed during the computation.

4 Generating EvalTrees

To generate evaluation information for data structures we use the idea of *observations*, first introduced by Hood [6]. All values for which an assertion is specified are *observed*, which constructs a corresponding `EvalTree` representing how far the data structure has been evaluated. The key idea is that the context of a computation demands head normal forms (*hnf*). Whenever such an hnf is computed we extend its `EvalTree` by means of a side effect. This means an `Uneval` leaf is replaced by `Eval [Uneval,...,Uneval]` where the number of `Uneval`s within the list is equal to the arity of the constructor of the hnf.

Because we construct and use `EvalTree`s in program parts that are not linked by data-flow and for efficiency reasons we use mutable references (`IORef`s) in our new EvalTree representation.

```
data EvalTree = EvalR [EvalTreeRef] | UnevalR
type EvalTreeRef = IORef EvalTree
```

With this representation it is not necessary to descend the whole data structure, when extending it in a leaf position. Instead, we can directly update the leaf.

Observable data types are represented by the following class:

```
class Observe a where
  obs :: a -> EvalTreeRef -> a
```

We demonstrate how an instance of this class can be defined by means of the list data type:

```
instance Observe a => Observe [a] where
  obs (x:xs) r = unsafePerformIO $ do [aRef,bRef] <- mkEvalTreeCons r 2
                                      return (obs x aRef : obs xs bRef)
  obs []       = unsafePerformIO $ do mkEvalTreeCons r 0
                                      return []
```

Whenever the context demands the evaluation of an observed value the corresponding node in the `EvalTree` is extended by means of the function

```
mkEvalTreeCons :: EvalTreeRef -> Int -> IO [EvalTreeRef]
mkEvalTreeCons r n = do refs <- sequence (replicate n emptyUnevalRef)
                        writeIORef r (EvalR refs)
                        return refs

emptyUnevalRef :: IO EvalTreeRef
emptyUnevalRef = newIORef UnevalR
```

Furthermore observers are added to the (not yet evaluated) arguments of the resulting constructor. These observers on demand extend the `IORef`s returned by `mkEvalTree` (`aRef` and `bRef`), which are also added to the new `EvalR` node within the `EvalTree`. The initial observer can be added with the function

```
observe :: Observe a => a -> IO (EvalTreeRef,a)
observe x = do r <- emptyUnevalRef
               return (r,obs x r)
```

This function is called, whenever an assertion is added to a data structure, as discussed in the next section.

On top of these functions, it is possible to define a late (in contrast to prompt) implementation of our lazy assertions. It stores all assertions of the program within a global state. At the end of the execution, all checks within this state are executed. Failed assertions are reported to the user.

5 Promptness

So far, our assertions fulfill two major goals. They respect the laziness of the program and they provide non-determinism by means of the operators `(|||)`, `(***)`, and `(&&&)`. It remains to make them prompts. The implementation shall suspend checks on unevaluated parts of data structures and directly awake them if these parts are evaluated to hnf. For this purpose we extended the idea of the prompt, lazy assertion logic from [1] to our monadic frame work. The key idea is the evaluation of assertions with coroutines stored within the `Uneval` leaves of the `EvalTree`. If the main computation replaces an `Uneval` leaf by some `Eval` node, then suspended coroutines within the `Uneval` leaf are restarted and evaluated. This may on the one hand result in new suspended coroutines for suspended assertions and checks on already evaluated parts of the `EvalTree`. On the other hand it can also result in having succeeded to check an assertion or showing the violation of an assertion. To highlight the relevant parts for a violation of an assertion, we collect position informations about the `EvalTree` while checking an assertion. Unfortunately, for lack of space, we cannot go into further details.

6 Related Work

The first systematic approach of adding assertions to a functional language targets the strict language Scheme [5]. It provides convenient constructs for expressing properties of functions, including higher-order functions, and augmenting function definitions with assertions. Laziness is irrelevant and promptness trivial for strict functional languages. Instead a major concern of this work is which program part to blame when an assertion fails. The approach to blaming cannot directly be transferred to a lazy language, because there the run-time stack does not reflect the call structure. Instead a cooperation with the Haskell tracer Hat [9] may provide a solution in the future. The Scheme approach has been transferred to Haskell [7], but without taking its lazy semantics into account.

The first paper on *lazy* assertions for the lazy language Haskell [2] uses normal functions with Boolean result for expressing properties and hence the assertions are not prompt. The paper gives several examples of where the lack of promptness renders the assertions useless. Furthermore, expressibility of properties of functions is limited and the implementation requires concurrency language extensions as provided only by GHC.

In the first paper on *lazy and prompt* assertions for Haskell [1] properties are expressed in a pattern logic. The logic provides quantifiers and context patterns that allow referring to substructures of the tested value. However, most Haskell users find this logic hard to understand and many simple properties, such as that two lists have the same lengths, require complex descriptions. The implementation of the pattern logic is only sketched.

QuickCheck is a library for testing Haskell functions with random data [3]. Normal Boolean functions express expected properties, for example

```
prop :: Int -> [Int] -> Property
prop x xs = ordered xs ==> ordered (insert x xs)
```

where `ordered :: [Int] -> Bool` states that the function `insert` preserves order. Normal Boolean functions can be used, because only total, finite data structures are tested. An extension for (finite) partial values [4] has fundamental limit whereas our assertions fully support laziness. It can be very hard to generate random test data, for example input strings for a parser that are likely to be parseable. QuickCheck can only test top-level functions whereas an assertion can be attached to any local definition or subexpression. So testing with random data and testing with real data as our assertions do are two different methods which complement each other.

7 Conclusion

We have presented a new approach for augmenting lazy functional programs such as Haskell with assertions. The implementation is a portable library that requires only two common extensions of Haskell 98, `unsafePerformIO` and `IORef`s, that are supported by all Haskell compilers. The assertions are lazy and prompt. Most importantly, the combinator language for expressing asserted properties is easy to use, because it is similar to familiar parser combinator libraries. It combines pattern matching and non-deterministic computations. Furthermore, it is very expressive, allowing the formulation of any imaginable computable property. Assertions for functional values are easy write and syntax highlighting simplifies the identification of parts of a value that are relevant for a failure.

References

1. Olaf Chitil and Frank Huch. A pattern logic for prompt lazy assertions in Haskell. In Andrew Butterfield Zoltan Horvath, editor, *Implementation and Application of Functional Languages: 18th International Workshop, IFL 2006*, volume 4449 of *LNCS*. Springer, 2007.
2. Olaf Chitil, Dan McNeill, and Colin Runciman. Lazy assertions. In Phil Trinder, Greg Michaelson, and Ricardo Pena, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003*, LNCS 3145, pages 1–19. Springer, November 2004.
3. K. Claessen and R. J. M. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. 5th Intl. ACM Conference on Functional Programming*, pages 268–279. ACM Press, 2000.
4. Nils Anders Danielsson and Patrik Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, LNCS 3125, pages 85–109. Springer-Verlag, July 2004.
5. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59. ACM Press, 2002.
6. A. Gill. Debugging Haskell by observing intermediate datastructures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. (Proc. 2000 ACM SIGPLAN Haskell Workshop).
7. Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS 2006*, LNCS 3945, pages 208–225, 2006.
8. Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998.
9. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *ACM Workshop on Haskell*, 2001.

Towards A New Denotational Semantics For Curry and The Algebra of Curry[⋆]

Bernd Braßel and Jan Christiansen

Institute of Computer Science
University of Kiel, 24098 Kiel, Germany
{bbr,jac}@informatik.uni-kiel.de

Abstract. It has often been observed that a *point-free style* of programming provides a more abstract view on programs. We aim to use the gain in abstraction to obtain a denotational semantics for *functional logic languages* in a straightforward way. We propose a set of basic operations based on which arbitrary functional logic programs can be transformed to point-free programs. Surprisingly, the additional features of functional logic languages do require *less* basic operations to obtain point-free programs than known approaches for functional languages. This effect is mostly due to employing so called *function patterns*.
We interpret the basic operations in *relation algebra* to obtain a denotational semantics for the whole point-free subset of functional logic languages. As this subset is connected to the whole language by the proposed transformation this enables a purely algebraic view on the whole language.
A final example illustrates the additional possibilities gained by this approach.

1 Introduction

This work aims at combining the results of several well researched fields. Most notably, these are the fields of *declarative programming* and *relation algebra*. Moreover, the importance of a *point-free* view on *programming* has been emphasized particularly in the applications of category theory to semantics of programming languages. We expect the combination of these fields to be very fruitful. Many results and techniques from the field of relation algebra could be used for the analysis of functional logic programs and the knowledge about the implementation of functional logic languages could be employed to concisely solve relation-algebraic problems. As a concrete example motivating this approach in Section 3 we use algebraic calculations to optimise an operator definition. Due to space reasons the semantics proposed in Section 3 is strict, whereas the transformation developed in Section 2 takes laziness into account. In order to cover laziness one needs to extent the domain of values and to redefine the transposition of a relation as we want to show in future work.

[⋆] This work has been partially supported by DFG grant Ha 2457/5-1.

1.1 Functional Logic Languages

We consider a functional logic program as a constructor-based rewriting system, allowing extra variables on the right hand side and so called *function patterns.* This section establishes some of the involved notation, referring to [13] for functional logic programming and [3] for function patterns. For our examples we adopt the syntax of Curry [15].

For a program P, Σ_P is a signature partitioned into two sets, the set of *constructors* $\mathcal{C}_P$ and the set of defined *operations* $\mathcal{O}_P$. We denote n-ary constructor (operation) symbols by c^n (f^n, g^n) omitting the arity where it is apparent. For a set of (sorted) variables $\mathcal{X}$, the sets of (well-sorted) *terms* and *constructor terms* are denoted by $\mathcal{T}(\Sigma_P, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}_P, \mathcal{X})$, respectively. The function $var(t)$ yields the set of variables occurring in term t. A term is *linear* if every variable occurs at most once. Sorts and constructors are introduced by a `data` declaration, as shown in Example (1). The "`a`" in the third declaration denotes that `[a]` is a polymorphic type. We use syntactic sugar for list terms, e.g., `[True,False]` instead of `(True : (False : []))`. Operations are defined by *rewrite rules* of the form "$f\ p_1\ \ldots\ p_n = e$" where $f^n \in \mathcal{O}_P$ and $p_1, \ldots, p_n$ are called *patterns.* The standard way to define an operation in Curry is that each pattern of the rewrite rules is a *constructor term* and each variable occurs not more than once in the whole pattern. In other words, the term $(p1, \ldots, p_n)$ must be a *linear constructor term.* Such standard Curry programs are evaluated by *weakly needed narrowing* [14]. A *narrowing step* is a rewrite step combined with substitutions for extra variables needed to match the left-hand side of an applicable rule. E.g., a narrowing step for Example (2) is `app x [True]` $\rightarrow_{\{\texttt{x} \mapsto \texttt{[]}\}}$ `[True]`.

```
data Success = Success
data Bool = True | False   (1)
data [a] = [] | a : [a]
```

```
app []     ys = ys
app (x:xs) ys = x : (app xs ys)   (2)
```

In addition to defining rules, *type signatures* are used to declare the sorts an operation is defined for. As an example, `app:: [a] -> [a] -> [a]` declares that `app` is an operation which maps two (polymorphic) lists to a list. These lists have elements of the same type.

As in Example (2) there might be more than one possible narrowing step. Functional logic languages provide *non-deterministic search* to obtain values in this situation. Non-determinism does not only stem from narrowing but also from operator definitions with overlapping left hand sides.

```
coin :: Bool
coin = True    (3)
coin = False
```

E.g., there are two derivations `coin` $\rightarrow$ `True` and `coin` $\rightarrow$ `False` or, for short, `coin` $\rightarrow$ `True | False`. The operation `coin` is very popular because it can be used to exemplify an important feature of functional logic languages: *call-time choice*, cf. [12]. With call-time choice non-deterministic choices for arguments are done before application, at least conceptually. In combination with laziness call-time choice is affine to the concept of referential transparency, as illustrated in the next example.

```
or :: Bool -> Bool -> Bool
or x y = if x then x else y
```
(4)

For example, employing call-time choice the expression e := `or coin True` is evaluated to `True` only, because both occurrences of `x` are substituted with the same value. That is, employing call-time choice there are the derivations $e \rightarrow$ `if True then True else True` $\rightarrow$ `True` and $e \rightarrow$ `if False then False else True` $\rightarrow$ `True`. In the dual conception, run-time choice, `e` is reduced to `if True then coin else True` $\rightarrow$ `coin` $\rightarrow$ `True | False` and `if False then coin else True` $\rightarrow$ `True`. That is, the evaluation of e yields non-deterministically `True` or `False`. We have to make sure to capture call-time choice accordingly in Section 3.

An important operation in functional logic languages is the *strict equality* `(=:=) :: a -> a -> Success`. The intended meaning is that the equation e_1`=:=`e_2 is satisfied iff e_1 and e_2 can be reduced to the same constructor term, see [11] for a detailed discussion. In Curry, satisfying a predicate like the above equation is modelled by a reduction to the special type `Success`, cf. Program (1).

Strict equality can be employed to allow a certain type of non-standard operator definitions. A non-linear left hand side of a rewrite rule $l = e$ with x occurring n times in l can be taken as syntactic sugar for a rule where x is replaced by different variables `x1` ...`x`n in l and e is extended by constraints `(x1=:=x2)`, ..., `(x1=:=x`n`)`. See [2, 4.1] for a discussion of this transformation.

Finally, *function patterns* [3] allow operator definitions with arbitrary first order patterns.[1] The intended meaning of a function pattern is that only the pattern is evaluated to a constructor term. The argument is evaluated until a unification is possible. Unlike extra variables unified with strict equality `(=:=)` this unification may bind a pattern variable to an unevaluated term. Non-linear patterns are still treated with `(=:=)` as described above.

```
last (app xs [x]) = x
```
(5)

The operation `last` yields the last element of a given list. We apply `last` (5) to the list `[True,False]`, that is, we evaluate the term e :=`last [True,False]`. We get the following reduction:

`app xs [x]` $\rightarrow_{\{\texttt{xs} \mapsto \texttt{y : ys}\}}$ `y : (app ys [x]))` $\rightarrow_{\{\texttt{ys}\mapsto\texttt{[]}\}}$ `[y,x]` (6)

`[y,x]` can be unified with `[True,False]` yielding $e \rightarrow_{\{\texttt{xs}\mapsto\texttt{[True]},\texttt{x}\mapsto\texttt{False}\}}$ `False`.

1.2 Point-free Style

The term *point-free* originates from topology where you have points in a space and functions that operate on these points. In functional programming spaces are types, functions are functions and points are the arguments of a function. In *point-free* style you do not explicitly access the points, that is, the arguments of a function. The idea of the *point-free* programming paradigm is to build functions by combining simpler ones. It was introduced by John Backus in his Turing

[1] We will see in the Section 2.2 that the restriction imposed on function patterns in [3] is not necessary to obtain a reasonable semantics for such patterns. Therefore, we will omit it for simplicity.

Award Lecture in 1977 [4]. The counterpart of *point-free* is *point-wise*, that is, functions that explicitly access their arguments. Here, *point-free* programs are based on a couple of *point-wise* primitives.

1.3 Relation Algebra

In this section we present the relation-algebraic basics for Section 3. For a detailed introduction to relation algebra see [18]. Relation algebras can also be defined as a special kind of categories, first in [5].

In this paper we make use of the so called *concrete* relation algebra, in which relations are represented as sets of Cartesian products. We denote the set of all relations with domain X and range Y by $[X \leftrightarrow Y]$. We write $R :: X \leftrightarrow Y$ instead of $R \in [X \leftrightarrow Y]$ and xRy instead of $(x, y) \in R$ and call $X \leftrightarrow Y$ the type of R.

Lattice For each set $[X \leftrightarrow Y]$ the operations $\cup$ (intersection, meet), $\cap$ (union, join), and $\bar{\cdot}$ (complement, negation) together with a greatest element L (universal relation) and a smallest element O (empty relation) form a boolean lattice. A lattice provides an induced order denoted by $\subseteq$.

Union, intersection and complement of the concrete relation algebra are the standard union, intersection and complement operations on sets. The empty relation is the empty set and the universal relation $\mathsf{L} :: X \leftrightarrow Y$ is $X \times Y$. The lattice order is the subset relation. The relation algebra enriches the lattice with two operations $\circ$ (multiplication) and $\cdot^{\top}$ (inversion, transposition) and a constant I (identity relation).

The Multiplication of relation algebra is a binary associative operation for which the identity relation I is the neutral element. For each set X, $\mathsf{I} :: X \leftrightarrow X$ is defined by $x\mathsf{I}x$ for all $x \in X$. The multiplication of two relations $R :: X \leftrightarrow Y$ and $S :: Y \leftrightarrow Z$ is defined by $x(R \circ S)z$ iff there exists $y \in Y$ such that xRy and ySz. It is $R \circ S :: X \leftrightarrow Z$.

The Inversion of a relation $R :: X \leftrightarrow Y$ is defined by $yR^{\top}x \Leftrightarrow xRy$. It is $R^{\top} :: Y \leftrightarrow X$. In Section 3 we use some properties of inversion: $(R^{\top})^{\top} = R$, $(R \circ S)^{\top} = S^{\top} \circ R^{\top}$ and $(R \cup S)^{\top} = R^{\top} \cup S^{\top}$.

Direct Products Given a product $X \times Y$ there are two projections which map a pair (u_1, u_2) to its first component u_1 and second component u_2, respectively. We consider the corresponding projection relations $\pi_1 :: X \times Y \leftrightarrow X$ and $\pi_2 :: X \times Y \leftrightarrow Y$ such that $u\pi_1 x \Leftrightarrow x = u_1$ and $u\pi_2 y \Leftrightarrow y = u_2$ for $u = (u_1, u_2)$.

The tupling $[\cdot, \cdot]$ of two relations $R :: X \leftrightarrow Y$ and $S :: X \leftrightarrow Z$ is defined by $x\,[R, S]\,(y, z) \Leftrightarrow xRy \wedge xSz$ and its type is $X \leftrightarrow (Y, Z)$. The parallel composition $\cdot \mid\mid \cdot$ of two relations $R :: X \leftrightarrow Z$ and $S :: Y \leftrightarrow W$ is defined by $(x, y)(R \mid\mid S)(z, w) \Leftrightarrow xRz \wedge ySw$ and its type is $(X, Y) \leftrightarrow (Z, W)$. Tupling and the projections are connected by the following properties:

$$S \text{ total} \Rightarrow [R, S] \circ \pi_1 = R \qquad\qquad R \text{ total} \Rightarrow [R, S] \circ \pi_2 = S$$

Direct Sums are introduced as a dual concept to direct products. Sums are constructed by the *injections* ι_1 and ι_2 which can be used to define the semantics of constructors.

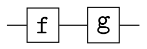
2 Transformation to Point-free Style

2.1 The Set of "Primitives"

In this section we define a small set of point-wise operations which allow the definition of arbitrary functional logic operations in a point-free style.

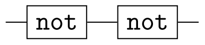
Composition of Operations The first such "primitive" is *sequential composition*, occasionally simply referred to as "composition".

```
(*) :: (a -> b) -> (b -> c) -> a -> c
(f * g) x = g (f x)
```
(7)



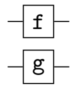
The primitive `(*)` is a flipped version of `(.)`. Whereas `(f . g)` reads as "f after g", `(f * g)` is more like "f before g". This is more convenient with regard to our aim of a relation-algebraic treatment of programming semantics. Furthermore, the left-to-right reading provides a very descriptive graphical representation. The composition is visualised by connecting two operations with a line, indicating that the output of one is the input of the other. Such visualisations were also used in connecting functional programs [16] and allegory theory with hardware design [8] and to describe physical structures in general [19]. Simple definitions can be made point-free by using sequential composition, cf. Example (8).

```
involution x = not (not x)
involution = not * not
```
(8)



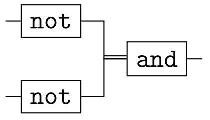
Operations with several arguments are composed by *parallel composition*.

```
(/) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f / g) (x,y) = (f x,g y)
```
(9)



Example (10) illustrates the use of parallel composition. The Operation `(*)` has a higher precedence than `(/)` making the parenthesis necessary.

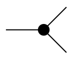
```
nor :: Bool -> Bool -> Bool
nor x y = not x && not y

nor :: (Bool,Bool) -> Bool
nor = (not / not) * and
```
(10)



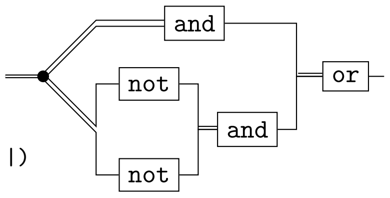
Discretely, we have changed the type of `nor` to a so called "uncurried" version. We use curried operations only when higher order is employed, as discussed in Paragraph "Higher Order".

Interface Adaption So far, we can express only right linear rules. Sharing arguments is the first of the primitives dealing with what we call "interface adaption". Interface adaption means that the connectives of two operations have to be copied, joined or reordered in some way. An uncurried and point-free version of the boolean operator "if and only if" (12) can be formulated using `(/)` and `fork`.

```
fork :: a -> (a,a)
fork x = (x,x)
```
(11)



```
(<=>) :: Bool -> Bool -> Bool
x <=> y = x && y || not x && not y

(<=>) :: (Bool,Bool) -> Bool
(<=>) = fork * ((&&)/(not/not)*(&&)) * (||)
```
(12)



There are two more primitives for interface adaption. The operator `unit` to "discard a value" and the identity `id` to "pass a value on". Both are exemplified in the following sections.

```
unit :: _ -> ()
unit _ = ()
```
(13)

```
id :: a -> a
id x = x
```
(14)

Data Structures, Inversion and Pattern Matching We do not wish to abstract from concrete domains at this point. Later in Section 3, we treat data structures in the standard way of sums and products. Here we define different operations to construct data. Each constructor of the original program will be assigned one operation.

```
nil :: () -> [a]                    true, false :: () -> Bool
nil () = []                         true  () = True
cons :: (a,[a]) -> [a]              false () = False
cons (x,xs) = x : xs
```
(15)

Note that these operations are again uncurried and that the constants `True`, `False`, and `[]` are extended with an argument. The reason for the latter extension will become apparent soon.

What we have seen so far is a more or less standard treatment of expressing functional programs in a point-free style. To concisely express pattern matching and to combine several rules we employ two additional features of functional *logic* programming, i.e., non-determinism and function patterns.

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```
(16)

```
coin :: () -> Bool
coin = true ? false
```
(17)

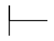
As stated in the introduction, overlapping rules in functional logic languages lead to non-deterministic search, cf. [14]. In principal, all non-determinism can be introduced by permitting only a single operation with overlapping rules `(?)` (16), cf. [2]. We use `(?)` to combine the rules of a function, cf. Example (17). Note that the introduction of the argument `()` for constant constructors extends to all definitions of constants.

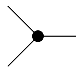
```
invert :: (a -> b) -> (b -> a)
invert f = f' where f' (f x) = x
```
(18)

Function patterns can be used to invert arbitrary operations. This yields the primitive `invert` defined in (18). The semantics of function patterns are described in [3] in terms of an possibly infinite set of rewrite rules. We aim at giving a denotational semantics for function patterns for the first time.

The expressive power of function patterns can be estimated by considering that all other logic features can be obtained by using function patterns.

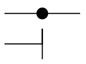
E.g., `invert unit :: () -> a` yields a logic variable when applied to `()` and `invert fork :: (a,a) -> a` performs unification, cf. Section 1.1. Therefore we can define the following useful abbreviations for interface adaption.

```
unknown :: () -> a
unknown = invert unit
```
(19)


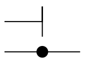
```
join :: (a,a) -> a
join = invert fork
```
(20)


The operations `up` and `down` demand the discarded argument to be an empty tuple which is important for the definition of pattern matching.

```
up :: (a,()) -> a
up = (id / unknown)*join
```
(21)

```
fst :: (a,_) -> a
fst = (id / unit)*up
```
(22)


```
down :: ((),a) -> a
down = (unknown / id)*join
```
(23)

```
snd :: (_,a) -> a
snd = (unit / id)*down
```
(24)


The expressive power gained by function patterns is paid with computational overhead, cf. [3]. It is thus desirable to replace a function pattern by an equivalent operation without such patterns. We think that our semantics gives a base on which optimizing procedures can be developed. As a first outlook in this direction we give a derivation of a more efficient version of `last` defined in (5) in Section 3.

```
head :: [a] -> a                  tail :: [a] -> [a]
head = invert cons * fst          tail = invert cons * snd
```
(25)

In addition to `(?)` to combine rules, the primitive `invert` can be used to express arbitrary pattern matching including function patterns. A constructor pattern is a linear constructor term, cf. Section 1.1. In order to match such a pattern we only have to invert the according constructors and then adapt the result like shown in (25). From this point of view it becomes apparent why constant constructors are extended with an argument: to make them invertible.

There is one last feature concerning pattern matching in connection with laziness. If a value is discarded, e.g., by using `unit`, it is not evaluated. The semantics of pattern matching demands that matching is ensured regardless of whether the resulting variable bindings are used or not. The operations `head` and `tail` defined in (25) use one of the variables bound by the matching and therefore the pattern matching is indeed performed. In general we have to combine several of the primitives introduced so far to achieve the desired evaluation.

```
null :: [a] -> Bool
null []    = True
null (_:_) = False
```
(26)

Example (26) shows a case in which the bindings of the matching are discarded. The point-free version has to make sure that a) the empty tuple of `(invert nil)` and b) the pair resulting from `(invert cons)` are demanded, and not more. The following definition provides these properties.

```
null = invert nil * true ? invert cons * (unit/unit) * join * false
```
(27)

The astute reader might wonder why we introduce non-determinism for a perfectly deterministic operation like the pattern matching of `null`. The reason for this is twofold. 1) From a semantic point of view the non-deterministic branching does not matter. If the matching was indeed deterministic, for a given deterministic value all but one branch will finitely (even immediately) fail. 2) In a functional logic language patterns are *not* always deterministic nor treated in a sequential way (like in Haskell). Overlapping patterns induce non-determinism which is easily captured by our approach.

```
member :: [a] -> a
member (x:_)  = x                (28)
member (_:xs) = member xs
```

For example, the operation `member` defined in (28) non-deterministically relates a list with each of its elements. Without further additions this behaviour is captured by the transformation. The following definition shows a point-free version of `member`.

```
member = invert cons * fst ? invert cons * snd * member  (29)
```

Example (28) also illustrates that recursive functions simply stay recursive. There is no need for changes, e.g., a special recursion operator. Complex patterns are treated like complex expressions, i.e., they are composed with `(*)` and `(/)` before inverting the whole expressions. We treat function patterns in the very same way. For example, the function `last` (5) is translated to:

```
last = invert ((id / (id/nil) * cons)  * app) * snd * up  (30)
```

Higher Order In order to introduce higher-order operations we need to adapt the well known pair `apply` and `curry` to our setting. A first point to consider is

```
apply :: (a -> b,a) -> b
apply (f,x) = f x              (31)
```

that values of type `a` correspond to operations of type `() -> a`. Because higher-order operations should be first class objects we need to translate them in the same way. An operation of type `(a -> b)` must become an object of type `() -> (a -> b)` when used as an argument of an operation. If we assume this kind of translation we can define `apply` and `curry` straightforward.

```
curry :: (() -> (a,b) -> c) -> () -> (a -> b -> c)
curry f = \ () x y -> f () (x,y)                     (32)
```

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []                (33)
map f (x:xs) = f x : map f xs
```

The step to obtain the first curried version of a given function cannot be formulated in an equally general way because of call-time choice. This is illustrated by a standard example of a higher-order operation in Example (33). We can already translate `map` with the primitives introduced so far, adding `apply`.

```
map :: (a -> b,[a]) -> [b]
map = invert (id / nil) * nil                             (34)
    ? invert (id / cons) * adapt * ((apply / map) * cons)
```

We assume `adapt` to map the tuple structure `(f,(x,xs))` to `((f,x),(f,xs))`. We omit its concrete definition by means of `id, unit, invert` and `fork`. We want to map the operation `not` on the list `[False,True]`.

```
not = invert true * false ? invert false * true
listFalseTrue = fork * (false / fork * (true / nil) * cons) * cons  (35)
mapNot = fork * (curryNot / listFalseTrue) * map
```

What should `curryNot :: () -> (Bool -> Bool)` be defined as? A first version might be `curryNot = const not`. But evaluating `mapNot ()` yields no solution. The reason is call-time choice. Because `f` is a variable the choice whether `f` is the operation "`invert true * false`" or "`invert false * true`" is made consistently for all applications of `f`. But this decision has to be made anew for each application of `f`. This can be achieved by η-expansion.

```
curryNot () x = not x
```
(36)

Using definition (36) `(mapNot ())` evaluates to `[True,False]` as intended. The example shows that a second version of each operation which will be applied higher order is needed. We have now illustrated all the point-wise primitives necessary to translate arbitrary Curry programs: `(*)` (7), `(/)` (9), `fork` (11), `unit` (13), `id` (14), `(?)` (16), `invert`(18), `apply` (31) and `curry` (32). The next section presents a formal definition of the transformation.

2.2 Obtaining Point-free Style in General

Modern functional logic languages provide syntactic sugar to formulate very concise and readable code, e.g., the `if` · `then` · `else` used in Example (4). In the following, we will consider the core language defined in Figure 1. We distinguishing *partial* and *full* applications in order to model higher order. For example, the Curry expression `map not []` is represented as `(map (PC not) [])`. For our programs, we assume sort correctness. Furthermore, we assume that the source program contains no unary operations nor unary constructors in order to avoid the concept of an "unary tuple". Figure 2 shows how expressions are trans-

$$
\begin{array}{llll}
P & ::= R^* & & \{\text{program}\} \\
R & ::= f^n\ p_1\ \dots\ p_n \texttt{ = } e & f \in \mathcal{O}_P & \{\text{rule}\} \\
p & ::= x & x \in \mathcal{X} & \{\text{(pattern) variable}\} \\
 & \mid\ (s^n\ p_1\ \dots\ p_n) & s \in \Sigma_P & \{\text{complex pattern}\} \\
e & ::= x & x \in \mathcal{X} & \{\text{variable}\} \\
 & \mid\ (s^n\ e_1\ \dots\ e_n) & s \in \Sigma_P & \{\text{full application}\} \\
 & \mid\ (\texttt{PC}\ s^n) & s \in \Sigma_P & \{\text{partial application}\} \\
 & \mid\ (\texttt{AP}\ e_1\ e_2) & & \{\text{higher order application}\}
\end{array}
$$

Fig. 1. The Core Language

$$
\begin{array}{ll}
exp(x) & = (\texttt{PC id}) \\
exp((s^0)) & = (\texttt{PC}\ s^1) \\
exp((s^n\ e_1\ \dots\ e_n)) & = (exp(e_1)\texttt{/}\dots\texttt{/}exp(e_n))\texttt{*}(\texttt{PC}\ s^n)\ n > 1 \\
exp((\texttt{PC}\ s^n)) & = \underbrace{\texttt{curry(}\dots\texttt{(curry}}_{n-1}\ cs)\dots) \\
exp((\texttt{AP}\ e_1\ e_2)) & = (exp(e_1)\texttt{/}exp(e_2))\texttt{*(PC apply)}
\end{array}
$$

Fig. 2. Transforming Expressions and Patterns

lated. Note that constants are replaced by unary symbols of the same name, cf. the discussion of Examples (15) and (25) in the previous section. Furthermore, the partial application of s is replaced by an application of a new symbol cs as motivated in paragraph "Higher Order" above. The operation cs is defined in

Figure 7. Because patterns are effectively a subset of expressions, the rules of Figure 2 are also used to translate patterns.

$$\begin{array}{ll} \mathit{int}(x) & = x \\ \mathit{int}((s\ e_1\ e_2\ \ldots\ e_n)) & = (\mathit{int}(e_1), (\mathit{int}(e_2), (\ldots, \mathit{int}(e_n))\ldots)) \\ \mathit{int}((\texttt{PC}\ s^n)) & = () \\ \mathit{int}((\texttt{AP}\ e_1\ e_2)) & = (\mathit{int}(e_1), \mathit{int}(e_2)) \end{array}$$

Fig. 3. Obtaining Interfaces

Next we present the general approach to "Interface Adaption", cf. the paragraph of the same name above. An *interface* is an abstraction from the actual structure of an expression. That is, it is a tree with the same branching structure as the original expression which contains variables that occur in the expression. The simple mapping from expressions to interfaces is depicted in Figure 3. We denote complex interfaces, i.e, those not in $\mathcal{X} \cup \{()\}$ by $i, i_1, i_2 \ldots$ and by $\mathit{var}(i)$ the set of all variables occurring in i.

$$\begin{array}{ll} \mathit{sel}(x,()) & = (\texttt{PC id}) \\ \mathit{sel}(x,y) & = \begin{cases} (\texttt{PC id}) & \text{, if } x = y \\ (\texttt{PC unit}) & \text{, otherwise} \end{cases} \\ \mathit{sel}(x,(i,i')) & = (\mathit{sel}(x,i)\texttt{/}\mathit{sel}(x,i'))\texttt{*} \left\{ \begin{array}{ll} ((\texttt{PC}\ \mathit{id})\texttt{/}u)\texttt{*} & \text{, if } x \in \mathit{var}(i) \wedge x \notin \mathit{var}(i') \\ (u\texttt{/}(\texttt{PC}\ \mathit{id}))\texttt{*} & \text{, if } x \notin \mathit{var}(i) \wedge x \in \mathit{var}(i') \\ & \text{, otherwise} \end{array} \right\} j. \end{array}$$

$$u = \texttt{(invert (PC unit))} \qquad j = \texttt{(invert (PC fork))}$$

$$\begin{array}{ll} \mathit{adapt}(i,()) & = \mathit{sel}(y,i) \text{ where } y \notin \mathit{var}(i) \\ \mathit{adapt}(i,x) & = \mathit{sel}(x,i) \\ \mathit{adapt}(i,(i_1,i_2)) & = (\texttt{PC fork}) * (\mathit{adapt}(i,i_1)\texttt{/}\mathit{adapt}(i,i_2)) \end{array}$$

Fig. 4. Variable Selection and Interface Adaption

Selecting variables from a given interface and adapting two interfaces is defined in Figure 4. Each occurrence of the selected variable is passed on by `id`, while all other variables are discarded by `unit`. The remaining operations introduced by $\mathit{sel}(\cdot,\cdot)$ ensure that the whole interface structure is matched. This is important in order to demand the whole pattern matching as discussed in connection with Example (26). The effect of mapping $\mathit{adapt}(\cdot,\cdot)$ is twofold. First, an application of the mapping $\mathit{sel}(\cdot,\cdot)$ is introduced for all leaves of the interface adapted to. Second, the incoming argument is copied as often as needed by employing the primitive `fork`. The examples given in the previous section often contain a more simple interface adaption. In the appendix, which will not be part of the final version of this paper but will be made available online as a technical report, we provide a set of simplification rules along with detailed examples. As these rules do not provide any additional insights we omit them here.

Extra variables on the right-hand side are added by the mapping $\mathit{addfree}(\cdot,\cdot)$, defined in Figure 5. For each variable a call to `unknown` alias `invert unit` is introduced as explained in Section 2.1. The rules of a defined operation are trans-

$$addfree(i_1, i_2) = (\underbrace{free \times \ldots \times (free}_{n} \times \texttt{id}) \ldots)\texttt{*}adapt(i_1', i_2)$$

$$\begin{array}{lll} where & free & = (\texttt{unit * (invert unit)}) \\ & i_1' & = (x_1, \ldots, (x_n, i_1) \ldots) \\ & \{x_1, \ldots, x_n\} & = var(i_2) \setminus var(i_1) \\ & e \times e' & = \texttt{fork*}(e\texttt{/}e') \end{array}$$

Fig. 5. Adding Logic Variables

$$rule(f\ p_1\ \ldots\ p_n\ \texttt{=}\ e) = (\texttt{invert}\ (exp(p_1)\texttt{/}\ldots\texttt{/}exp(p_n)))\texttt{*}adp\texttt{*}exp(e)$$
$$\text{where } adp = addfree(int((p_1, \ldots, p_n)), int(e))$$

Fig. 6. Transforming Rules

formed according to Figure 6. The general technique is: invert the transformed pattern then apply interface adaption and finally transform the body of the rule.

$$\begin{array}{ll} cons(c^0) & = c\ \texttt{() = }c' \\ cons(c^{n>1}) & = c\ \texttt{((x1,x2),...,xn) = }c'\ \texttt{x1 ... xn} \\ cOp(f) & = cf\ \texttt{() x = }f\ \texttt{x} \\ op(f_i) & = f\ \texttt{=}\ rule(r_{i1})\ \texttt{?}\ \ldots\ \texttt{?}\ rule(r_{in_i}) \\ prog(P) & = prims\ op(f_1) \ldots op(f_n)\ cOp(f_1) \ldots cOp(f_n)\ cons(c_1) \ldots cons(c_m) \end{array}$$

Fig. 7. Transforming Programs

Finally, the definitions of Figure 7 are employed to transform an entire program P where *prims* are the definitions of `(*)` (7), `(/)` (9), `fork` (11), `unit` (13), `id` (14), `(?)` (16), `invert`(18), `apply` (31) and `curry` (32). In Figure 7 we assume that $\mathcal{O}_P = \{f_1, \ldots, f_n\}$ and $\mathcal{C}_P = \{c_1, \ldots, c_m\}$ and that for each $f_i \in \mathcal{O}_P$, $r_{i1} \ldots r_{in_i}$ to be the rules defining f_i in P. As discussed with Example (15), a new function is introduced for each constructor in $\mathcal{C}_P$ in the first mapping of Figure 7. For simplicity, a new constructor symbol is introduced rather than a new function symbol. *cOp* in Figure 7 introduces the new functions needed to realize higher order, cf. Examples (33) and (36). *op* combines the rules defining an operation by `(?)`, cf. Example (17). In the last equation all three mappings are combined along with the definitions of the primitive operations to obtain the whole result. The resulting program P' is a program with the following signatures, where *rc* replaces constants by unary symbols, i.e., $rc(M) = \{s^1 \mid s^0 \in M\}$.

$$\begin{array}{ll} \mathcal{O}_{P'} & = rc(\mathcal{O}_P \cup \{cf \mid f \in \mathcal{O}_P\} \cup \mathcal{C}_P) \cup Prim \\ \mathcal{C}_{P'} & = rc(\{c' \mid c \in \mathcal{C}_P\}) \\ Prim & = \{\texttt{(*)}^3, \texttt{(/)}^3, \texttt{(?)}^2, \texttt{fork}^1, \texttt{id}^1, \texttt{unit}^1, \texttt{invert}^1, \texttt{apply}^1, \texttt{curry}^1\} \end{array}$$

Of course the question arises, how to give an account of the correctness of the transformation. Although the translation using `(*)` and `(/)` is standard [10], our use of function patterns is a new technique. For future work, we therefore plan to give a stronger formalisation of the operational behaviour of function patterns than [3] and prove the correctness in the framework of an operational semantics like the one introduced in [1].

3 Obtaining Semantics

We define a denotational semantics for Curry by defining a semantics for all point-wise primitives where $assocr = [\pi_1 \circ \pi_1, [\pi_1 \circ \pi_2, \pi_2]]$.

$$\begin{array}{rlrl}
[\![\ \texttt{f * g}\ ]\!] &= [\![\ \texttt{f}\ ]\!] \circ [\![\ \texttt{g}\ ]\!] & [\![\ \texttt{f / g}\ ]\!] &= [\![\ \texttt{f}\ ]\!] \mid\mid [\![\ \texttt{g}\ ]\!] \\
[\![\ \texttt{f ? g}\ ]\!] &= [\![\ \texttt{f}\ ]\!] \cup [\![\ \texttt{g}\ ]\!] & & \\
[\![\ \texttt{fork}\ ]\!] &= [\mathsf{I}, \mathsf{I}] & [\![\ \texttt{invert f}\ ]\!] &= [\![\ \texttt{f}\ ]\!]^{\top} \\
[\![\ \texttt{id}\ ]\!] &= \mathsf{I} & [\![\ \texttt{unit}\ ]\!] &= \mathsf{L} \\
[\![\ \texttt{apply}\ ]\!] &= (\mathsf{I} \mid\mid [\mathsf{I}, \mathsf{L}]) \circ [\mathsf{I}, \mathsf{I}]^{\top} \circ \pi_2 & [\![\ \texttt{curry f}\ ]\!] &= \mathsf{L} \circ [\mathsf{I}, [\![\ \texttt{f}\ ]\!]] \circ assocr
\end{array}$$

There is no one-to-one relation between the semantics of `apply` and `curry` and the corresponding point-wise primitives. Higher order is essentially modelled by manipulating a relational representation of the graph of an operation. We only define the semantics of `apply` and `curry` here to show that we can provide semantics for the full set of primitives. Constructors are represented by injections. That is, if a data type has n constructors and C is the k-th constructor we define the semantics of C by an injection to the k-th position of an n-ary sum. For example, the constructors `true` and `false` defined in (15) are represented as $\mathrm{tt} := \mathsf{I} + \mathsf{O}$ and $\mathrm{ff} := \mathsf{O} + \mathsf{I}$, respectively. In consequence, for the non-deterministic operation `coin`, cf. (17), holds: $[\![\ \texttt{coin}\ ]\!] = [\![\ \texttt{true}\ ]\!] \cup [\![\ \texttt{false}\ ]\!] = \mathrm{tt} \cup \mathrm{ff} = \mathsf{I} + \mathsf{I}$. In the model of the concrete relation algebra, cf. Section 1.3, this can be represented by the set $\{((), \texttt{tt}), ((), \texttt{ff})\}$.

Although the name might suggest thus, call-time choice does not coincide with strictness. It is the way that direct products are defined in relation algebra which implies that the proposed semantics indeed models call-time choice. For example, the semantics of a shared `coin`, i.e., $[\![\ \texttt{coin * fork}\ ]\!]$ is $\{((), \texttt{tt}), ((), \texttt{ff})\} \circ [\mathsf{I}, \mathsf{I}]$. By definition of $\circ$, I and $[\cdot, \cdot]$ this denotes the set $\{((), (\texttt{tt}, \texttt{tt})), ((), (\texttt{ff}, \texttt{ff}))\}$.

In general all sharing is introduced by `fork`. The semantics would be run-time choice iff the two expressions `f * fork` and `fork * (f / f)` are equal regardless of `f`. In contrast, in our semantics the following two properties hold.

$$R \circ [\mathsf{I}, \mathsf{I}] \subseteq [\mathsf{I}, \mathsf{I}] \circ (R \mid\mid R) \qquad R \ \mathit{univalent} \Leftrightarrow R \circ [\mathsf{I}, \mathsf{I}] = [\mathsf{I}, \mathsf{I}] \circ (R \mid\mid R)$$

Note that the given semantics is strict. The conceptual work on extending the given framework to provide a lazy semantics is advanced but not finished yet. The main reason to present a strict semantics is to show the promising new possibilities gained by giving an algebraic derivation of an optimised version of the definition of `last` introduced in (30).

For readability we will use fonts to distinguish between syntax and semantics instead of adding brackets. Names written in italic are the semantics of the same term written in typewriter, e.g., "*cons*" denotes the semantics of `cons`. First, we provide the semantics of `last` defined in (30) and `app` defined in (2). Note that $fst = \pi_1$ and $snd = \pi_2$.

$$last = ((\mathsf{I} \mid\mid (\mathsf{I} \mid\mid nil) \circ cons) \circ app)^\top \circ snd \circ up$$

$$app = (nil \mid\mid \mathsf{I})^\top \circ down \cup (cons \mid\mid \mathsf{I})^\top \circ assocr \circ (\mathsf{I} \mid\mid app) \circ cons$$

We want to calculate an operation called `last'`, that has the same semantics as `last` but does not use a function pattern. On the right hand side of each step, we state the relation-algebraic law that is applied.

$$\begin{aligned}
last &= (\mathsf{I} \mid\mid (\mathsf{I} \mid\mid nil) \circ cons \circ app)^\top \circ snd \circ up && (R \circ S)^\top = S^\top \circ R^\top \\
&= app^\top \circ (\mathsf{I} \mid\mid (\mathsf{I} \mid\mid nil) \circ cons)^\top \circ snd \circ up && (R \mid\mid S)^\top = R^\top \mid\mid S^\top \\
&= app^\top \circ (\mathsf{I}^\top \mid\mid ((\mathsf{I} \mid\mid nil) \circ cons)^\top) \circ snd \circ up && (R \mid\mid S) \circ snd = snd \circ S \\
&= app^\top \circ snd \circ ((\mathsf{I} \mid\mid nil) \circ cons)^\top \circ up
\end{aligned}$$

Now we invert the semantics of the operation `app`. We split *app* in two parts, app_1 and app_2. These are the semantics of the two rules of `app`. We simplify each rule of *app* separately. We use the abbreviation $assocl = [[\pi_1, \pi_2 \circ \pi_1], \pi_2 \circ \pi_2]$.

$$\begin{aligned}
app^\top &= (app_1 \cup app_2)^\top && (R \cup S)^\top = R^\top \cup S^\top \\
&= {app_1}^\top \cup {app_2}^\top
\end{aligned}$$

$$\begin{aligned}
{app_1}^\top &= ((nil \mid\mid \mathsf{I})^\top \circ down)^\top && (R \circ S)^\top = S^\top \circ R^\top \\
&= down^\top \circ (nil \mid\mid \mathsf{I})^{\top\top} && R^{\top\top} = R \\
&= down^\top \circ (nil \mid\mid \mathsf{I})
\end{aligned}$$

$$\begin{aligned}
{app_2}^\top &= ((cons \mid\mid \mathsf{I})^\top \circ assocr \circ (\mathsf{I} \mid\mid app) \circ cons)^\top && (R \circ S)^\top = S^\top \circ R^\top \\
&= cons^\top \circ (\mathsf{I} \mid\mid app)^\top \circ assocr^\top \circ (cons \mid\mid \mathsf{I})^{\top\top} && R^{\top\top} = R \\
&= cons^\top \circ (\mathsf{I} \mid\mid app)^\top \circ assocr^\top \circ (cons \mid\mid \mathsf{I}) && (R \mid\mid S)^\top = R^\top \mid\mid S^\top \\
&= cons^\top \circ (\mathsf{I}^\top \mid\mid app^\top) \circ assocr^\top \circ (cons \mid\mid \mathsf{I}) && \mathsf{I}^\top = \mathsf{I} \\
&= cons^\top \circ (\mathsf{I} \mid\mid app^\top) \circ assocr^\top \circ (cons \mid\mid \mathsf{I}) && assocr^\top = assocl \\
&= cons^\top \circ (\mathsf{I} \mid\mid app^\top) \circ assocl \circ (cons \mid\mid \mathsf{I})
\end{aligned}$$

In the next step we substitute the expression $app^\top$ in *last* by its definition. Again, we can treat the two arguments of the union separately.

$$\begin{aligned}
last &= app^\top \circ snd \circ ((\mathsf{I} \mid\mid nil) \circ cons)^\top \circ up && \text{def. of } app^\top \\
&= (app_1^\top \cup app_2^\top) \circ snd \circ ((\mathsf{I} \mid\mid nil) \circ cons)^\top \circ up && (R \cup S) \circ T = R \circ T \cup S \circ T \\
&= (app_1^\top \circ snd \circ ((\mathsf{I} \mid\mid nil) \circ cons)^\top \circ up) \\
&\quad \cup (app_2^\top \circ snd \circ ((\mathsf{I} \mid\mid nil) \circ cons)^\top \circ up) \\
&= last_1 \cup last_2
\end{aligned}$$

$$\begin{aligned}
\mathit{last}_1 &= \mathit{app}_1^\top \circ \mathit{snd} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} && \text{def. of } \mathit{app}_1^\top \\
&= \mathit{down}^\top \circ (\mathit{nil} \mid\mid \mathsf{I}) \circ \mathit{snd} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} && (R \mid\mid S) \circ \mathit{snd} = \mathit{snd} \circ S \\
&= \mathit{down}^\top \circ \mathit{snd} \circ \mathsf{I} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} && \mathit{down}^\top \circ \mathit{snd} = \mathsf{I} \\
&= \mathsf{I} \circ \mathsf{I} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} && \mathsf{I} \circ R = R \\
&= ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up}
\end{aligned}$$

By applying the transformation in the other direction we get the following rule.

```
last' (x:[]) = x
```

Now we substitute the right hand side of $\mathit{app}_2^\top$ in the definition of last_2 and transform the resulting term.

$$\begin{aligned}
\mathit{last}_2 &= \mathit{app}_2^\top \circ \mathit{snd} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} \\
&\qquad \text{def of } \mathit{app}_2^\top \\
&= \mathit{cons}^\top \circ (\mathsf{I} \mid\mid \mathit{app}^\top) \circ \mathit{assocl} \circ (\mathit{cons} \mid\mid \mathsf{I}) \circ \mathit{snd} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} \\
&\qquad (R \mid\mid S) \circ \mathit{snd} = \mathit{snd} \circ S \\
&= \mathit{cons}^\top \circ (\mathsf{I} \mid\mid \mathit{app}^\top) \circ \mathit{assocl} \circ \mathit{snd} \circ \mathsf{I} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} \\
&\qquad \mathit{assocl} \circ \mathit{snd} = \mathit{snd} \circ \mathit{snd} \\
&= \mathit{cons}^\top \circ (\mathsf{I} \mid\mid \mathit{app}^\top) \circ \mathit{snd} \circ \mathit{snd} \circ \mathsf{I} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} \\
&\qquad (R \mid\mid S) \circ \mathit{snd} = \mathit{snd} \circ S \\
&= \mathit{cons}^\top \circ \mathit{snd} \circ \mathit{app}^\top \circ \mathit{snd} \circ \mathsf{I} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up} \\
&\qquad \mathsf{I} \circ R = R \\
&= \mathit{cons}^\top \circ \mathit{snd} \circ \underbrace{\mathit{app}^\top \circ \mathit{snd} \circ ((\mathsf{I} \mid\mid \mathit{nil}) \circ \mathit{cons})^\top \circ \mathit{up}}_{\mathit{last}}
\end{aligned}$$

By applying the inverse transformation we get a second rule for `last'` that contains a recursive call. Now we unite the results for last_1 and last_2 and get a definition of `last` that does not use a function pattern.

```
last' [x]    = x
last' (_:xs) = last' xs
```

4 Related and Future Work

We are aware that many works on all three topics connected in this paper exist, i.e., on the semantics of functional logic programming languages, the point-free programming style and relation algebra. Here, we can only relate our work to a small selection.

Cunha, Pinto and Proença [10, 9] present a framework for transformations into point-free style. They present a library for point-free programming in Haskell which is similar to our primitives. Furthermore, they have developed a program that transforms arbitrary Haskell programs into point-free style and they present a tool for manipulating point-free programs. Although their focus is different from ours, especially [9] provided valuable insights, e.g., that the opportunities for automatic reasoning about programs is not as straightforward as the formalism might suggest. Therefore we aim to use our approach for program analysis like [7] and to prove (with only half automatic tool support) the correctness of optimisations for functional logic languages.

The book “Algebra of Programming” by Bird and de Moor [6] has been very influential for this work. They present a calculus for the algebraic manipulation of functional programs. We hope that we could give an idea that the framework of functional *logic* languages is an even more natural and promising field for this style of reasoning about programs. The elementary difference is the existence of non-determinism. Whereas in [6] every inversion and every non-deterministic definition resulting from inversion *must* be eliminated, the framework of functional logic languages allows much less restricted use of algebraic methods. The same is true a fortiori for approaches like [17] that aim on deriving a functional definition to compute the inversion of a given function definition.

Regarding the denotational semantics of functional (logic) languages, we want to relate our approach especially with two papers as future work. [20] proposes a denotational semantics for a functional language employing relation algebra. [12] provides a denotational semantics for functional logic languages based on cones. There are many interesting extensions to the framework of [12] which we want to investigate. However, our work presents a promising step towards covering function patterns for the first time.

Apart from relating with the existing approaches, we plan to extend our approach to cover laziness. Conceptual work for this extension already exists. Naturally, we want to work on the correctness of the proposed transformation, most suitably by using a framework like [1]. Furthermore, a formal relation to the semantics of [12] would further ensure the validity of the approach. In the long term we would like to use algebraic methods like demonstrated in Section 3 for program analysis and optimisation.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
3. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
4. J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

5. R. Berghammer and G. Schmidt. A relational view on gotos and dynamic logic. In Herbert Göttler and Hans Jürgen Schneider, editors, *Proceedings of the 8th Conference on Graphtheoretic Concepts in Computer Science (WG 82)*, pages 13–24. Hanser, 1982.
6. R. Bird and O. de Moor. *Algebra of programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
7. B. Braßel and M. Hanus. Nondeterminism analysis of functional logic programs. In *Proceedings of the International Conference on Logic Programming (ICLP 2005)*, pages 265–279. Springer LNCS 3668, 2005.
8. Carolyn Brown and Graham Hutton. Categories, Allegories, and Circuit Design. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, Los Alamitos, California, July 1994.
9. A. Cunha. *Point-free program calculation.* PhD thesis, Universidade do Minho, Departamento de Informática, 2005.
10. A. Cunha, J. Sousa Pinto, and J. Proença. A Framework for Point-free Program Transformation. In Andrew Butterfield, editor, *Revised Papers of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, number 4015 in Lecture Notes in Computer Science. Springer-Verlag, 2005.
11. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel leaf: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.
12. J. C. González-Moreno, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. Log. Program.*, 40(1):47–87, 1999.
13. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
14. 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.
15. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.
16. Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669. Springer Verlag, 1993.
17. Shin-Cheng Mu. *A Calculational Approach to Program Inversion.* PhD thesis, Oxford University Computing Laboratory, 2003.
18. Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs - Discrete Mathematics for Computer Scientists.* EATCS Monographs on Theoretical Computer Science. Springer, 1993.
19. Hermann von Issendorff. Algebraic description of physical systems. In Roberto Moreno-Díaz, Bruno Buchberger, and José Luis Freire, editors, *EUROCAST*, volume 2178 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2001.
20. Hans Zierer. *Programmierung mit Funktionsobjekten: Konstruktive Erzeugung semantischer Bereiche und Anwendung auf die partielle Auswertung.* PhD thesis, Technische Universität München, Fakultät für Informatik, 1988.