INSTITUT FÜR INFORMATIK

Programmiersprachen und Rechenkonzepte

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

Bad Honnef, 3.-5. Mai 2010

Michael Hanus, Fabian Reck (Hrsg.)

Bericht Nr. 1010

September 2010

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

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

Programmiersprachen und Rechenkonzepte

27. Workshop der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“ Bad Honnef, 3.-5. Mai 2010

Michael Hanus, Fabian Reck (Hrsg.)

Bericht Nr. 1010
September 2010

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

Dieser Bericht enthält eine Zusammenstellung der Beiträge des 27. Workshops Programmiersprachen und Rechenkonzepte, Physikzentrum Bad Honnef, 3.-5. Mai 2010.

Vorwort

Seit 1984 veranstaltet die GI–Fachgruppe „Programmiersprachen und Rechenkonzepte“ 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 interessanten 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 September 2010 Michael Hanus, Fabian Reck

Inhaltsverzeichnis

Typing Coroutines
Konrad Anton und Peter Thiemann 1

Using Haskell's Type Systems to Check Relation Algebraic Programs
Bernd Braßel 10

Minimally Strict Polymorphic Functions
Jan Christiansen 11

Combining Syntactic and Semantic Bidirectionalization
Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda und Meng Wang 12

Towards a Jitting VM for Prolog Execution
Carl Friedrich Bolz, Michael Leuschel und David Schneider 22

An Adaptive Compilation Framework for Data Parallel Programming in SAC
Clemens Grelck, Tim van Deurzen, Stephan Herhut und Sven-Bodo Scholz 23

Reguläres Ausdruckstheater
Sebastian Fischer, Frank Huch und Thomas Wilke 34

Nahtlose und effiziente Integration großer Zahlen in eine Programmiersprache
Christian Heinlein 35

Values in Object-Oriented Programming Languages
Beate Ritterbach und Axel Schmolitzky 36

Formalization of the Java$_\lambda$ type system
Martin Plümicke 44

E-Assessment kreativer Prüfungsleistungen
Herbert Kuchen 55

A new language for algebraic dynamic programming
Georg Sauthoff und Robert Giegerich 56

An experiment with the fourth Futamura projection
Robert Glück 57

Signale statt Generatoren!
Wolfgang Jeltsch 58

Twilight STM in Haskell
Anette Bieniusa, Arie Middelkoop und Peter Thiemann 59

Tofu - Towards Practical Church-Style Total Functional Programming
Baltasar Trancón y Widemann 60

Towards an Orchestrated Approach for Annotation Verification
Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec und Markus Schordan 71

A Satisfiability Modulo Theories Memory-Model and Assertion Checker for C
Jakob Zwirchmayr 86

The rCOS methodology and modeler
Volker Stolz 96

Erweiterung der Containerdarstellung von Listen mittels Relationen - Proving Properties About Functions on Lists Involving Element Tests
Daniel Seidel und Janis Voigtländer 108

A Functional, Successor List Based Version of Warshall's Algorithm
Rudolf Berghammer 119

Transforming Functional Logic Programs into Monadic Functional Programs
Bernd Braßel, Sebastian Fischer, Michael Hanus und Fabian Reck 129

Typing Coroutines

Konrad Anton and Peter Thiemann

Institut für Informatik, Universität Freiburg
{anton,thiemann}@informatik.uni-freiburg.de

Abstract. [1] A coroutine is a programming construct between function and thread. It behaves like a function that can suspend itself arbitrarily often to yield intermediate results and to get new inputs before returning a result. This facility makes coroutines suitable for implementing generator abstractions.
Languages that support coroutines are often untyped or they use trivial types for coroutines. This work supplies the first type system with dedicated support for coroutines. The type system is based on the simply-typed lambda calculus extended with effects that describe control transfers between coroutines.

1 Introduction

A coroutine is a programming construct between function and thread. It can be invoked like a function, but before it returns a value (if ever) it may suspend itself arbitrarily often to return intermediate results and then be resumed with new inputs. Unlike with preemptive threading, a coroutine does not run concurrently with the rest of the program, but rather takes control until it voluntarily suspends to either return control to its caller or to pass control to another coroutine. Coroutines are closely related to cooperative threading, but they add value because they are capable of passing values into and out of the coroutine and they permit explicit switching of control.

Coroutines have been invented in the 1960s as a means for structuring a compiler [4]. They have received a lot of attention in the programming community and have been integrated into a number of programming languages, for instance in Simula 67 [5], BETA, CLU [11], Modula-2 [19], Python [17], and Lua [16], and Knuth finds them convenient in the description of algorithms [8]. Coroutines are also straightforward to implement in languages that offer first-class continuations (e.g., Scheme [7]) or direct manipulation of the execution stack (e.g., assembly language, Smalltalk).

The main uses of coroutines are the implementation of compositions of state machines as in Conway's seminal paper [4] and the implementation of generators. A generator enumerates a potentially infinite set of values with successive invocations. The latter use has lead to renewed interest in coroutines and to their inclusion in mainstream languages like C# [14], albeit in restricted form as generators.

[1] An extended version of this paper has been submitted to TFP 2010.

Despite the renewed interest in the programming construct per se, the typing aspects of coroutines have not received much attention. Indeed, the supporting languages are either untyped (e.g., Lua, Scheme, Python), the typing for coroutines is trivalized, or coroutines are restricted so that a very simple typing is sufficient. For instance, in Modula-2, coroutines are created from parameterless procedures so that all communication between coroutines must take place through global variables. Also, for describing generators, a simple function type seems sufficient.

Contribution. We propose a static type system for first-class, stackful coroutines that may be used in both, symmetric and asymmetric ways.[2] Moreover, we permit passing arguments to a coroutine at each start and resume operation, and we permit returning results on each suspend and on termination of the coroutine (and we distinguish between these two events). Our type system is based on the simply-typed lambda calculus. It includes an effect system that describes the way the coroutine operations are used. We present a small-step operational semantics for the language and prove type soundness.

Outline. Sec. 2 describes the language CorDuroy. It starts with some examples (Sec. 2.1) before delving into operational semantics (Sec. 2.2) and the type system (Sec. 2.3). Sec. 3 discusses related work, and Sec. 4 concludes and outlines directions of further research.

2 CorDuroy

The language CorDuroy is a simply typed lambda calculus with recursive functions and operations for handling coroutines. Fig. 1 specifies the syntax; labels ℓ only occur at run time. We define λ-abstraction as sugar for the fixpoint operator: $\lambda x.e := \mathsf{fix}\,\lambda_.\lambda x.e$.

Coroutines in CorDuroy are run-time entities identified by a label ℓ. The only way to create them is by applying the create operator to a function. Once a coroutine has been created, it can be executed. Unlike threads in a multithreaded language, only one coroutine is active at any given time.

To activate a coroutine, there is a symmetric (transfer) and an asymmetric (resume) operator. The symmetric operator transfer suspends the currently executing coroutine and executes another[3]. The asymmetric operator resume builds a caller-callee relationship: if a coroutine resumes another coroutine, they become

[2] This terminology is due to De Moura and Ierusalemschy [15]. A coroutine is *stackful*, if it can suspend inside nested function calls. Coroutines are *asymmetric* if coroutine activity is organized in a tree-like manner: each coroutine invocation or resumption always returns and yields to its caller. In contract, *symmetric* coroutines can transfer control among each others without restrictions.

[3] We use the keywords established by De Moura and Ierusalemschy [15]. In Simula [5], transfer corresponds to the system procedure `RESUME`, whereas "asymmetric", yield and resume correspond to "semi-symmetric", `DETACH` and `CALL`, respectively.

$$
\begin{aligned}
B &::= \mathsf{Bool} \mid \mathsf{Unit} \mid \ldots \\
k^0 &::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \mid \ldots \\
k^1 &::= \neg \mid \ldots \\
k^2 &::= \wedge \mid \ldots \\
\ell, \ell', \ldots &\in \text{Labels} \\
x, y, f, \ldots &\in \text{Var}
\end{aligned}
\qquad
\begin{aligned}
v &::= k^0 \mid \mathsf{fix}\, \lambda f.\lambda x.e \mid \ell \\
e &::= k^n\, e_1 \ldots e_n \mid \mathsf{fix}\, \lambda f.\lambda x.e \\
&\quad \mid x \mid e\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \\
&\quad \mid \mathsf{create}\, x.e \mid \mathsf{yield}\, e \\
&\quad \mid \mathsf{resume}\, e\, e\, e\, e \mid \mathsf{transfer}\, e\, e \\
&\quad \mid \ell \\
\varphi &::= \bot \mid \tau \overset{..}{\rightsquigarrow} \tau/\tau \mid \top \\
\tau &::= B \mid \tau \xrightarrow{\varphi} \tau \mid \top \mid \bot \mid \tau \rightsquigarrow \tau/\tau
\end{aligned}
$$

Fig. 1: Syntax.

caller and callee. The yield operator inside the callee suspends the coroutine and returns control to the caller. Each of the three operators passes a value. In the remaining paper, we understand "activate" to mean either transfer or resume, but not yield.

The caller-callee relationship is also used when a coroutine finally returns a value, as the value is then passed to the caller. Activating a coroutine after it has returned causes a run-time error; hence, the caller needs to know whether the callee coroutine has terminated. resume requires therefore as its third and fourth parameter two *result functions*, one to call with yielded values and one to call with the returned value[4].

The language includes a countable set of primitive functions k^n, having each an arity $n \geq 0$. Partial application of primitive functions is not allowed.

2.1 Examples

This section contains short examples of CorDuroy programs. We assume that integers and strings are among the basic types B and that there are constants k^n for arithmetic operations, comparison, and printing. We also use the common $\mathsf{let} \cdot = \cdot\, \mathsf{in} \cdot$ sugar for readability.

Divisors. Generators can be used to compute sequences one element at a time. Fig. 2a shows a coroutine which generates the divisors of a number, and a consumer which iterates over the divisors until the generator returns (and the second result function of resume is called).

Mutable references. Coroutines are the only stateful construct in CorDuroy. In Fig. 2b, a mutable reference is simulated by a coroutine which keeps an integer value in a local variable. Whenever it is resumed with a function $\mathsf{Int} \to \mathsf{Int}$, it lets the function update the value and returns the new value. The example also

[4] Alternatively, the λ-calculus could be extended with variant types in order to tag the result of resume with how it was obtained. We chose the two-continuation resume for simplicity.

```
1  let divisors_of = λn.
2    create _.λ_.
3      ((fix λloop.λk.
4        if (> k n) then unit
5        else let rem = (mod n k) in
6          let _= (if (= rem 0)
7            then yield k else unit)
8          in loop (+ k 1 )) 1)
9  in let g = divisors_of 24 in
10   ((fix λf. λ_.
11     resume g unit
12       (λn. let _ = (print_int n)
13           in (f unit) )
14       (λ_. (print_str "finito")))
15     unit)
16 // output: 1 2 3 4 6 8 12 24 finito
```

(a) Compute all divisors.

```
1  let makeref = λx0.
2    let main = fix λloop.λx.λupd.
3        let x' = upd x in
4        let upd' = yield x' in
5          loop x' upd'
6    in create _. main x0
7  in let undef = fix λf.λx.(f x)
8  in let write = λr.λv.
9    resume r (λ_.v)
10     (λ_.unit) (λ_.unit)
11 in let read = λ r.
12   resume r (λx.x) (λx.x) undef
13 in
14  let r = makeref 1 in
15  let _ = print_int (read r) in
16  let _ = write r 2 in
17  print_int (read r)
18 // output: 1 2
```

(b) Mutable references.

Fig. 2: Code examples.

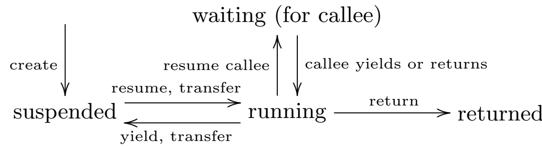

Fig. 3: Life cycle of coroutines.

shows how fix can be used to create a diverging function with any desired return type in the read function[5].

2.2 Operational Semantics

This section presents a small-step operational semantics for CorDuroy, starting with a life-cycle based view on coroutines to motivate the stack-based representation used in the reduction rules in Fig. 5.

The life cycle of a coroutine consists of the states *suspended*, *running*, *waiting* and *returned*, as shown in Fig. 3. At any moment, there is only one running coroutine, and only in the running coroutine, ordinary β-reductions take place.

The running coroutine can apply create to a function, creating a new coroutine which starts life in the suspended state (E-Create). It can also resume a

[5] The language could alternatively be extended with a special variant of the resume operator for coroutines which never return.

$$
\begin{aligned}
C ::= \Box \mid k^n\, v_1 \ldots v_{i-1}\ C\ e_{i+1} \ldots e_n \quad (n \geq i \geq 1) \\
\mid e\, C \mid C\, v \mid \mathsf{if}\, C\, \mathsf{then}\, e\, \mathsf{else}\, e \\
\mid \mathsf{resume}\, C\, e\, e\, e \mid \mathsf{resume}\, v\, C\, e\, e \mid \mathsf{resume}\, v\, v\, C\, e \mid \mathsf{resume}\, v\, v\, v\, C \\
\mid \mathsf{yield}\, C \mid \mathsf{transfer}\, C\, e \mid \mathsf{transfer}\, v\, C
\end{aligned}
$$

$$
S ::= \ell @ e; S^? \qquad \mathrm{labels}(\ell @ e; S^?) = \{\ell\} \cup \mathrm{labels}(S^?)
$$

$$
S^? ::= \epsilon \mid S \qquad \mathrm{labels}(\epsilon) = \emptyset
$$

Fig. 4: Evaluation contexts and stacks.

$$
\frac{x^* \notin \mathrm{free}(e) \cup \{x\} \qquad \ell^* \notin \mathrm{dom}\,(\mu) \cup \mathrm{labels}(S^?) \cup \{\ell\} \qquad v^* = \lambda x^*.((\lambda x.e\, x^*)\, \ell^*) \qquad \mu' = \mu \cup \{(\ell^*, v^*)\}}{\left\langle \ell @ C[\mathsf{create}\, x.e]\, ; S^? \mid \mu \right\rangle \to \left\langle \ell @ C[\ell^*]\, ; S^? \mid \mu' \right\rangle} \text{ E-Create}
$$

$$
\frac{\ell' \in \mathrm{dom}\,(\mu) \qquad e = \mathsf{resume}\, \ell'\, v_a\, v_s\, v_n}{\left\langle \ell @ C\,[e]\, ; S^? \mid \mu \right\rangle \to \left\langle \ell' @ \left(\mu\,(\ell')\ v_a\right) ; \ell @ C\,[e]\, ; S^? \mid \mu \setminus \ell' \right\rangle} \text{ E-Res}
$$

$$
\frac{x^* \text{ fresh} \qquad e_2 = \mathsf{resume}\, \ell\, v_a\, v_s\, v_n}{\langle \ell_1 @ C_1\, [\mathsf{yield}\, v_y]\, ; \ell_2 @ C_2\, [e_2] \mid \mu \rangle \to \langle \ell_2 @ C_2\, [v_s\, v_y] \mid \mu\, [\ell_1 \mapsto \lambda x^*.C_1[x^*]] \rangle} \text{ E-Yie}
$$

$$
\frac{}{\left\langle \ell_1 @ v_r ; \ell_2 @ C_2\, [\mathsf{resume}\, \ell\, v_a\, v_s\, v_n]\, ; S^? \mid \mu \right\rangle \to \left\langle \ell_2 @ C_2\, [v_n\, v_r]\, ; S^? \mid \mu \right\rangle} \text{ E-CoRet}
$$

$$
\frac{}{\left\langle \ell @ C\, [\mathsf{transfer}\, \ell\, v_a]\, ; S^? \mid \mu \right\rangle \to \left\langle \ell @ C\, [v_a]\, ; S^? \mid \mu \right\rangle} \text{ E-TraSelf}
$$

$$
\frac{\ell' \in \mathrm{dom}\,(\mu) \qquad x^* \text{ fresh}}{\left\langle \ell @ C\, \left[\mathsf{transfer}\, \ell'\, v\right] ; S^? \mid \mu \right\rangle \to \left\langle \ell' @ (\mu(\ell')\, v); S^? \mid (\mu \setminus \ell') \cup \{(\ell, \lambda x^*.C[x^*])\} \right\rangle} \text{ E-Tra}
$$

$$
\text{E-TraErr} \quad \frac{\ell' \notin \mathrm{dom}\,(\mu)}{\langle \ell @ C\, \left[\mathsf{transfer}\, \ell'\, v_a\right] \mid \mu \rangle \to \mathsf{Error}} \qquad \text{E-ResErr} \quad \frac{\ell' \notin \mathrm{dom}\,(\mu)}{\langle \ell @ C\, \left[\mathsf{resume}\, \ell'\, v_a\, v_s\, v_n\right] \mid \mu \rangle \to \mathsf{Error}}
$$

Fig. 5: Small-step operational semantics rules for coroutine operations.

suspended coroutine (the *callee*), becoming its *caller* (E-Res). In doing so, it enters the *waiting* state, and the callee becomes *running*.

A running coroutine can also yield, after which it is *suspended* and the caller *running* (E-Yie). If a running coroutine reduces to a value, it is said to *return* that value. The returning coroutine enters its terminal state, and the value is then passed to the caller if there is one (E-CoRet) or becomes the final result of the program.

Alternatively, the running coroutine can transfer control to a suspended coroutine, suspending itself. In this case, the successor coroutine not only enters

the running state, but it also becomes the (only) callee of the predecessor's caller (E-Tra).

In the rules, the state of a program being evaluated is represented as a pair $\langle S \mid \mu \rangle$ of stack and store. The stack S contains, from left to right, the running coroutine, its caller, its caller's caller and so on, each in the form of labeled contexts $\ell @ e$ (see Fig. 4). As the running coroutine is the top of the stack, the reduction rules must never pop the last labeled context off the stack.

All suspended coroutines[6] are kept in the store μ, a function from labels ℓ to values v. The values in the store are the continuations of the yield and transfer expressions which caused the coroutine to be suspended, or, in the case of newly created coroutines, functions which are constructed to be applied likewise.

If a coroutine attempts to activate another coroutine which is not in the store (i.e., not suspended), execution aborts with a run-time error (E-ResErr, E-TraErr)[7]. As an exception to this rule, a coroutine may transfer to itself (E-TraSelf)[8].

2.3 Type system

The type system is based on the simply-typed λ-calculus, with an effect system describing which coroutine actions may occur during the evaluation of an expression.

The effect part of the type and effect system summarizes the yield and transfer expressions which may be evaluated during the evaluation of an expression. The propagation of effects through function application permits a called function to yield and transfer on behalf of the running coroutine in a type-safe way.

If an expression has the effect $\tau_i \stackrel{..}{\rightsquigarrow} \tau_o/\tau_r$, then its execution may yield a value of type τ_o to the calling coroutine and expect a value of type τ_i when it is activated again. It may also transfer execution to a coroutine which yields values of type τ_o or returns a value of type τ_r.

Effects φ form a lattice with bottom element $\bot$ and top element $\top$ (see Fig. 6). $\bot$ means that the expression will under no circumstance ever yield. Effect $\top$ means that yield expressions with different types are possible and nothing can be said about the values.

The type system features basic types B, function types, coroutine types as well as top and bottom types. Function arrows are annotated with the effect which may occur during the function's evaluation. We write $\tau_1 \rightarrow \tau_2$ for $\tau_1 \xrightarrow{\bot} \tau_2$.

A value of type $\tau_i \rightsquigarrow \tau_o/\tau_r$ corresponds to a coroutine which can be resumed with values of input type τ_i and yields values of output type τ_o or returns a value of return type τ_r.

---
[6] An implementation would keep the coroutines within the store all the time and annotate them with their state instead; however, the notion of putting coroutines into the store and taking them out again makes the rules easier to read.

[7] The alternatives to this approach are discussed in Sec. 4.

[8] This feature may be useful in implementations of cooperative multi-tasking: typically, a data structure stores runnable tasks, and if only one task is runnable, it may have to perform a task switch to itself.

$$\bot \sqcup \varphi = \varphi \sqcup \bot = \varphi$$
$$\top \sqcup \varphi = \varphi \sqcup \top = \top$$

$$(\tau_i \ddot{\rightsquigarrow} \tau_o/\tau_r) \sqcup (\tau_i' \ddot{\rightsquigarrow} \tau_o'/\tau_r') = (\tau_i \sqcap \tau_i') \ddot{\rightsquigarrow} (\tau_o \sqcup \tau_o') / (\tau_r \sqcup \tau_r')$$

$$\varphi_1 \sqsubseteq \varphi_2 \quad \text{iff} \quad \exists \varphi_1'.\varphi_1 \sqcup \varphi_1' = \varphi_2$$

(a) Effects.

$$\top \sqcup \tau = \tau \sqcup \top = \top$$
$$\bot \sqcap \tau = \tau \sqcap \bot = \bot$$
$$\bot \sqcup \tau = \tau \sqcup \bot = \top \sqcap \tau = \tau \sqcap \top = \tau$$
$$\tau_1^+ \sqcup \tau_2^+ = \begin{cases} \tau_1^+ & \tau_1^+ = \tau_2^+ \\ \top & \text{otherwise} \end{cases}$$
$$\tau_1^+ \sqcap \tau_2^+ = \begin{cases} \tau_1^+ & \tau_1^+ = \tau_2^+ \\ \bot & \text{otherwise} \end{cases}$$

(b) Types.

Fig. 6: Join and meet.

Types form a flat lattice with bottom $\bot$ and top $\top$. For simplicity, subtyping is not allowed, and subeffecting is only allowed in create and fix expressions. Join and meet on types are defined in figure 6, where τ^+ represent types except for $\top$ and $\bot$.

The rules relating to coroutines are given in Fig. 7. The type environment Γ maps variables to their types. The store typing $\Sigma \subseteq \text{Labels} \times \{\tau_i \rightsquigarrow \tau_o/\tau_r | \tau_{i,o,r} \neq \top, \tau_i \neq \bot\}$ maps labels to the types of the corresponding coroutines at run time. The exclusion of $\top$ and $\bot$ is a consequence of the deliberate absence of subtyping. Note that type rules only extend Γ, not Σ; expressions are type-checked against a fixed Σ, and the preservation property guarantees that some (possibly extended) Σ can be found after each evaluation step.

Most type rules compute the effect of their expression by joining the effects of the subexpressions. The only exceptions are T-Fix and T-Create, in which the effect of the body expression is moved onto the function arrow or into the coroutine type.

yield and transfer contribute an effect with its input type τ_i. Both suspend the current coroutine and expect a value of type τ_i the next time it is activated. The output and return types in the effect of yield describe that yield certainly causes the coroutine to yield a value of that type, but never causes a return. transfer, however, transfers control *and the relationship to the caller* to a coroutine which, in turn, may yield and return. Therefore, T-Tra puts the other coroutine's output and return types into the effect in order to force the surrounding yield and return expressions to match.

3 Related Work

Formalizations of coroutines. De Moura and Ierusalemschy [15] formally define coroutines in an untyped λ-calculus with mutable variables as a model for Lua coroutines. Their work contains a comprehensive overview of the state of the art in coroutines and related techniques.

Wang and Dahl [18] formalize the control-flow aspects of idealized Simula coroutines. The operational semantics of Belsnes and Østvold [1] also focuses on

$$\textsc{T-Fix}\ \frac{\Gamma, f:\tau_1 \xrightarrow{\varphi} \tau_2, x:\tau_1 | \Sigma \vdash e : \tau_2 \& \varphi' \qquad \varphi' \sqsubseteq \varphi}{\Gamma | \Sigma \vdash \mathsf{fix}\, \lambda f.\lambda x.e : (\tau_1 \xrightarrow{\varphi} \tau_2) \& \bot}$$

$$\textsc{T-Label}\ \frac{\Sigma(\ell) = \tau_i \rightsquigarrow \tau_o/\tau_r}{\Gamma | \Sigma \vdash \ell : \tau_i \rightsquigarrow \tau_o/\tau_r \& \bot}$$

$$\textsc{T-Create}\ \frac{\Gamma, x : \tau_i \rightsquigarrow \tau_o/\tau_r | \Sigma \vdash e : \tau_i \xrightarrow{\varphi} \tau_r \& \varphi' \quad \varphi, \varphi' \sqsubseteq \tau_i \stackrel{..}{\rightsquigarrow} \tau_o/\tau_r \quad \tau_{i,o,r} \neq \top, \tau_i \neq \bot}{\Gamma | \Sigma \vdash \mathsf{create}\, x.e : \tau_i \rightsquigarrow \tau_o/\tau_r \& \bot}$$

$$\textsc{T-Res}\ \frac{\begin{array}{c}\Gamma | \Sigma \vdash e_c : \tau_i \rightsquigarrow \tau_o/\tau_r \& \varphi_1 \\ \Gamma | \Sigma \vdash e_a : \tau_i \& \varphi_2 \qquad \Gamma | \Sigma \vdash e_s : \tau_o \xrightarrow{\varphi_3} \tau_q \& \varphi_4 \qquad \Gamma | \Sigma \vdash e_n : \tau_r \xrightarrow{\varphi_5} \tau_q \& \varphi_6\end{array}}{\Gamma | \Sigma \vdash \mathsf{resume}\, e_c\, e_a\, e_s\, e_n : \tau_q \& \bigsqcup_{i=1\ldots6} \varphi_i}$$

$$\textsc{T-Yie}\ \frac{\Gamma | \Sigma \vdash e : \tau_o \& \varphi_1}{\Gamma | \Sigma \vdash \mathsf{yield}\, e : \tau_i \& (\tau_i \stackrel{..}{\rightsquigarrow} \tau_o/\bot) \sqcup \varphi_1}$$

$$\textsc{T-Tra}\ \frac{\Gamma | \Sigma \vdash e_c : \tau_a \rightsquigarrow \tau_o/\tau_r \& \varphi_1 \qquad \Gamma | \Sigma \vdash e_a : \tau_a \& \varphi_2}{\Gamma | \Sigma \vdash \mathsf{transfer}\, e_c\, e_a : \tau_i \& (\tau_i \stackrel{..}{\rightsquigarrow} \tau_o/\tau_r) \sqcup (\varphi_1 \sqcup \varphi_2)}$$

Fig. 7: Typing rules for coroutine operations.

the control-flow aspects but includes threads and thread-coroutine interaction. Laird [10] presents a process calculus in which the coroutine is the basic building block. Berdine and coworkers [2] define coroutines in their process calculus.

Language design. Languages with parameterless coroutines include Simula [5], Modula-2 [19], and 2.PAK [13] and BETA [9]. However, the type systems of these languages need not treat coroutines with much sophistication because the coroutine operations do not pass values.

Some mainstream dynamically-typed languages like Python [17] and Lua [16] pass values to and from coroutines, but without a static type system. C# [14] has static typing and generators (asymmetric coroutines with parameters only for yield), but as the yield-equivalent may only be used lexically inside the generator's body, the type system avoids the complexity involved with stackful coroutines.

Marlin's ACL [12] is a (statically typed) coroutine extension of Pascal in which coroutines can accept parameters. In analogy to the separation between procedures and functions in Pascal, it features separate syntax for symmetric and asymmetric coroutines. The problem of procedures performing coroutine operations on behalf of the enclosing coroutine is solved by referring to the static block structure, which simplifies the type system at the expense of flexibility.

Lazy languages like Haskell [6] get asymmetric coroutines for free: a coroutine can be viewed as a transformer of a stream of input values to a stream of output values, which is straightforward to implement using lazy lists. Blazevic [3] produced a more sophisticated monad-based implementation.

4 Conclusion

We presented CorDuroy, a language with type-safe stackful asymmetric and symmetric first-class coroutines, and proved its soundness. CorDuroy constitutes the first provably sound type system for an eager-evaluated language that supports realistic and expressive facilities for coroutines.

References

1. Belsnes, D., Østvold, B.M.: Mixing threads and coroutines (2005), submitted to FOSSACS 2005, available by mail from `bjarte@nr.no`
2. Berdine, J., O'Hearn, P., Reddy, U., Thielecke, H.: Linear continuation-passing. Higher-Order and Symbolic Computation 15(2-3), 181–208 (2002)
3. Blazevic, M.: monad-coroutine: Coroutine monad transformer for suspending and resuming monadic computations. `http://hackage.haskell.org/package/monad-coroutine` (2010)
4. Conway, M.E.: Design of a separable transition-diagram compiler. Comm. ACM 6(7), 396–408 (1963)
5. Dahl, O.J., Myrhaug, B., Nygaard, K.: SIMULA 67 Common Base Language. Norwegian Computing Center, Oslo (1970), revised version 1984
6. Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition` (Dec 1998)
7. Haynes, C.T., Friedman, D.P., Wand, M.: Obtaining coroutines with continuations. Computer Languages 11(3), 143–153 (1986)
8. Knuth, D.E.: Fundamental Algorithms, The Art of Computer Programming, vol. 1. Addison-Wesley, Reading, MA, 2nd edn. (1968)
9. Kristensen, B.B., Pedersen, B.M., Madsen, O.L., Nygaard, K.: Coroutine sequencing in BETA. In: Proc. of 21st Annual Hawaii International Conference on Software Track. pp. 396–405. IEEE Computer Society Press, Los Alamitos, CA, USA (1988)
10. Laird, J.: A calculus of coroutines. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP. Lecture Notes in Computer Science, vol. 3142, pp. 882–893. Springer (2004)
11. Liskov, B.: CLU reference manual. Springer Verlag (1981), lNCS 114
12. Marlin, C.D.: Coroutines: a programming methodology, a language design and an implementation. Springer (1980)
13. Melli, L.F.: The 2.pak language: Goals and description. In: Proc. of the 4th International Joint Conference on Artificial Intelligence. Tbilisi, USSR (1975)
14. Microsoft Corp.: C# Version 2.0 Specification (2005), `http://msdn.microsoft.com/en-US/library/618ayhy6(v=VS.80).aspx`
15. de Moura, A.L., Ierusalimschy, R.: Revisiting coroutines. ACM Trans. Program. Lang. Syst. 31(2), 1–31 (2009)
16. de Moura, A.L., Rodriguez, N., Ierusalimschy, R.: Coroutines in lua. Journal of Universal Computer Science 10, 925 (2004)
17. Van Rossum, G., Eby, P.: PEP 342 – coroutines via enhanced generators. `http://www.python.org/dev/peps/pep-0342/` (2005)
18. Wang, A., Dahl, O.J.: Coroutine sequencing in a block structured environment. BIT Numerical Mathematics 11(4), 425–449 (1971), `http://www.springerlink.com/content/g870vkxx22861w50`
19. Wirth, N.: Programming in Modula-2. Springer (1982)

Using Haskell's Type Systems to Check Relation Algebraic Programs

Bernd Braßel

Abstract

Relation algebra provides a theoretically well founded framework to state algorithms in a declarative and concise way. Among other properties the language of relations is based on a rigorous typing discipline. Current systems to compute with relations do not, however, provide the user with type inference to ease programming. In addition, the systems lack in other aspects, like the possibility to define new data types, or to use primitive types for, e.g., numbers conveniently, or to easily define new control structures.

We introduce a binding for the lazy functional programming language Haskell to the basic operations implemented in C which underly the relation algebra system RelView. The advantages of such a binding are (at least) twofold:

1) Haskell programmers are provided with the possiblity to write highly efficient relational programs in a concise way.
2) Relational programmer are supported with a means to let infer and check types for their programs. Moreover, they can take advantage of the superior programming possibilities of Haskell.

We will describe three levels of detail for typing relational programs. The most detailed level needs to make use of a number of extensions to the standard Haskell type system.

Minimally Strict Polymorphic Functions

Jan Christiansen
Christian-Albrechts-Universität Kiel
jac@informatik.uni-kiel.de

Abstract

In a non-strict functional programming language like Haskell functions that yield the same results for total arguments can still differ for partial arguments. We can relate functions that agree for total arguments by a less-strict ordering. Naturally the question arises whether one can identify if a function is as non-strict as possible. A tool, called Sloth, assists programmers in checking whether a function is minimally strict.

To test a polymorphic function we have to choose a monomorphic instance of the function. By employing free theorems we show that a polymorphic function is indeed minimally strict if and only if its monomorphic Boolean instance is minimally strict. In fact, we only prove this statement for the polymorphic function type $\forall\alpha.[\alpha] \rightarrow [\alpha]$. But we can generalize the statement by employing the type classes *Foldable* and *Traversable*.

Combining Syntactic and Semantic Bidirectionalization*
(Extended Abstract)

Janis Voigtländer**, Universität Bonn, Germany, jv@iai.uni-bonn.de

Zhenjiang Hu, National Institute of Informatics, Japan

Kazutaka Matsuda, Tohoku University, Japan

Meng Wang, University of Oxford, United Kingdom

1 Introduction

Bidirectionalization is the task to come up, for some function *get* :: $\tau_1 \rightarrow \tau_2$, with a function *put* :: $\tau_1 \rightarrow \tau_2 \rightarrow \tau_1$ such that if *get* maps an *original source* s to an *original view* v, and v is somehow changed into an *updated view* v', then *put* applied to s and v' produces an *updated source* s' in a meaningful way. Two different flavors of automatic bidirectionalization have been proposed: syntactic and semantic. Syntactic bidirectionalization (Matsuda et al. 2007) works on a syntactic representation of (somehow restricted) *get*-functions and synthesizes appropriate definitions for *put*-functions algorithmically. Semantic bidirectionalization (Voigtländer 2009) does not inspect the syntactic definitions of *get*-functions at all, but instead provides a single definition of *put*, parameterized over *get* as a semantic object, that does the job by invoking *get* in a kind of "simulation mode".

Both syntactic and semantic bidirectionalization have their strengths and weaknesses. Syntactic bidirectionalization heavily depends on syntactic restraints exercised when implementing the *get*-function. Basically, the technique of Matsuda et al. (2007) can only deal with programs in a custom first-order language subject to linearity restrictions and absence of intermediate results between function calls. Semantic bidirectionalization, in contrast, provides very easy access to bidirectionality within a general-purpose language, liberated from the syntactic corset as to how to write functions of interest. The price to pay for this in the case of the approach of Voigtländer (2009) is that it works for polymorphic functions only, and so far has been unable to deal with view updates that change the shape of a data structure (more on this critical issue below). The syntactic approach, on the other hand, is successful for many such shape-changing updates, and can deal with non-polymorphic functions.

* The full version of this paper appears at ICFP'10.

** The research reported here was performed while this author visited the National Institute of Informatics, Tokyo, under a fellowship by the Japan Society for the Promotion of Science, ID No. PE09076.

Here we present an approach for combining syntactic and semantic bidirectionalization. The resulting technique inherits the limitations in program coverage from *both* techniques. That is, basically only functions that are written in the first-order language, are linear, and treeless in the sense of Wadler (1990), and *moreover* are polymorphic, can be dealt with. What we gain by the combination is improved updatability. To explain what we mean by this, we have to elaborate on the phrase "in a meaningful way" in the first sentence of this introduction. So, when is a get/put-pair "good"? How should s, v, v', and s' in $get\ s \equiv v$ and $put\ s\ v' \equiv s'$ be related? One natural requirement is that if $v \equiv v'$, then $s \equiv s'$, or, put differently,

$$put\ s\ (get\ s) \equiv s\,. \tag{1}$$

Another requirement to expect is that s' and v' should be related in the same way as s and v are, or, again expressed as a round-trip property,

$$get\ (put\ s\ v') \equiv v'\,. \tag{2}$$

But the latter condition is often too hard to satisfy in practice. For fixed get, it can be impossible to provide a put-function fulfilling equation (2) for every choice of s and v', simply because v' may not even be in the range of get. One solution is to make the put-function partial and to only expect this law to hold in case $put\ s\ v'$ is actually defined. Of course, a trivially consistent put-function we could then always come up with is the one for which $put\ s\ v'$ is *only* defined if $get\ s \equiv v'$ and which simply returns s then. Clearly, this choice would satisfy both equations (1) and (2), but would be utterly useless in terms of updatability.

So our evaluation criteria for "goodness" are that get/put should satisfy equation (1), that they should satisfy equation (2) whenever $put\ s\ v'$ is defined, and that $put\ s\ v'$ should be actually defined on a big part of its potential domain, indeed preferably for all s and v' of appropriate type. With this measure in hand, one can compare different bidirectionalization methods. Semantic bidirectionalization as proposed by Voigtländer (2009) has the problem that $put\ s\ v'$ can only be defined when $get\ s$ and v' have the same *shape* (length of a list, structure of a tree, ..., and in some situations even with constraints on the equivalence and relative ordering of elements in data structures). Syntactic bidirectionalization as proposed by Matsuda et al. (2007) does not suffer from such a central and common (to all invocations) updatability weakness, but in many cases also rejects updates that one would really like to see accepted. The benefit of our combined technique now is that on the intersection of the classes of programs to which the original syntactic and semantic techniques apply, we can do strictly better in terms of updatability than either technique in isolation. We are never worse than the better of the two in a specific case.

Before proceeding, we slightly revise the consistency conditions (1) and (2). Since our emphasis is on the updatability inherent in a get/put-pair, we make the partiality of put explicit in the type via optionality of the return value.

Definition 1. *Let* τ_1 *and* τ_2 *be types. Let functions* $get :: \tau_1 \rightarrow \tau_2$ *and* $put :: \tau_1 \rightarrow \tau_2 \rightarrow \mathsf{Maybe}\ \tau_1$ *be given. We say that put is* consistent for *get if:*

- *For every* $s :: \tau_1$*, put* $s\ (get\ s) \equiv \mathsf{Just}\ s$.
- *For every* $s, s' :: \tau_1$ *and* $v' :: \tau_2$*, if put* $s\ v' \equiv \mathsf{Just}\ s'$*, then get* $s' \equiv v'$.

2 Syntactic Bidirectionalization

The technique of Matsuda et al. (2007) builds on the constant-complement approach of Bancilhon and Spyratos (1981). The basic idea is that for a function

$$get :: \tau_1 \to \tau_2$$

one finds a function

$$compl :: \tau_1 \to \tau_3$$

such that the pairing of the two,

$$\begin{aligned}&paired :: \tau_1 \to (\tau_2, \tau_3)\\&paired\ s = (get\ s, compl\ s)\end{aligned}$$

is an injective function. Given an inverse $inv :: (\tau_2, \tau_3) \to \tau_1$ of *paired*, one obtains that

$$\begin{aligned}&put :: \tau_1 \to \tau_2 \to \tau_1\\&put\ s\ v' = inv\ (v', compl\ s)\end{aligned}$$

makes equations (1) and (2) true.

In reality, asking for a full inverse *inv* of *paired* is too much. The function *paired* may not even be surjective. So one relaxes *inv* to be a partial function, either implicitly as Matsuda et al. (2007) do, or explicitly in the type. With

$$inv :: (\tau_2, \tau_3) \to \mathsf{Maybe}\ \tau_1$$

and the requirements that

- for every $s :: \tau_1$, $inv\ (paired\ s) \equiv \mathsf{Just}\ s$, and
- for every $s' :: \tau_1$, $v' :: \tau_2$, and $c :: \tau_3$, if $inv\ (v', c) \equiv \mathsf{Just}\ s'$, then $paired\ s' \equiv (v', c)$,

we obtain that

$$\begin{aligned}&put :: \tau_1 \to \tau_2 \to \mathsf{Maybe}\ \tau_1\\&put\ s\ v' = inv\ (v', compl\ s)\end{aligned}$$

is consistent for *get* in the sense of Definition 1.

The approach of Matsuda et al. (2007) is to perform all the above by syntactic program transformations. We illustrate it based on an example. One generalization over the above picture is that instead of Maybe we will use an arbitrary monad. This allows for more flexible use of the resulting *put*-function, and also enables us to provide informative error messages if desired.

Example 1. Assume our *get*-function is as follows, sieving a list to keep only every second element:

$$
\begin{aligned}
&get :: [\alpha] \to [\alpha]\\
&get\ [\,] \qquad\qquad = [\,]\\
&get\ [x] \qquad\quad\ \ = [\,]\\
&get\ (x : y : zs) = y : (get\ zs)
\end{aligned}
$$

This function fulfills the syntactic prerequisites imposed by Matsuda et al. (2007). In particular, it is *linear* (no variable occurs more than once in a single right-hand side) and there is no function call with anything else than variables in its arguments.

Given the above, the following complement function is automatically derived:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Compl}\ \alpha = \mathsf{C}_1 \mid \mathsf{C}_2\ \alpha \mid \mathsf{C}_3\ \alpha\ (\mathsf{Compl}\ \alpha)\\
&compl :: [\alpha] \to \mathsf{Compl}\ \alpha\\
&compl\ [\,] \qquad\qquad = \mathsf{C}_1\\
&compl\ [x] \qquad\quad\ \ = \mathsf{C}_2\ x\\
&compl\ (x : y : zs) = \mathsf{C}_3\ x\ (compl\ zs)
\end{aligned}
$$

The basic ideas for the derivation of *compl* are that variables dropped when going from left to right in a defining equation of *get* are collected by *compl*, and that, where necessary, different data constructors (of same arity/type) are used on the right-hand sides of *compl* to disambiguate between overlapping ranges of right-hand sides of *get*. (In this specific example, this is not what causes different data constructors to be used. Instead, the simple fact that different arities are required, due to different numbers of dropped variables and recursive calls, leads to different data constructors.)

A tupling transformation (Pettorossi 1977) gives the following definition for the paired function:

$$
\begin{aligned}
&paired :: [\alpha] \to ([\alpha], \mathsf{Compl}\ \alpha)\\
&paired\ [\,] \qquad\qquad = ([\,], \quad \mathsf{C}_1)\\
&paired\ [x] \qquad\quad\ \ = ([\,], \quad \mathsf{C}_2\ x)\\
&paired\ (x : y : zs) = (y : v, \mathsf{C}_3\ x\ c)\\
&\quad \mathbf{where}\ (v, c) = paired\ zs
\end{aligned}
$$

Syntactic inversion, basically just exchanging left- and right-hand sides, plus introduction of monadic error propagation, gives:

$$
\begin{aligned}
&inv :: \mathsf{Monad}\ \mu \Rightarrow ([\alpha], \mathsf{Compl}\ \alpha) \to \mu\ [\alpha]\\
&inv\ ([\,], \quad \mathsf{C}_1) \quad\ = return\ [\,]\\
&inv\ ([\,], \quad \mathsf{C}_2\ x) = return\ [x]\\
&inv\ (y : v, \mathsf{C}_3\ x\ c) = \mathbf{do}\ zs \leftarrow inv\ (v, c)\\
&\qquad\qquad\qquad\qquad\qquad\ \ return\ (x : y : zs)\\
&inv\ _ = fail\ \texttt{"Update violates complement."}
\end{aligned}
$$

Finally,

```
put :: Monad μ ⇒ [α] → [α] → μ [α]
put s v′ = inv (v′, compl s)
```

can be fused (Wadler 1990) to:

```
put :: Monad μ ⇒ [α] → [α] → μ [α]
put []           []         = return []
put [x]          []         = return [x]
put (x : y : zs) (y′ : v′) = do zs′ ← put zs v′
                                 return (x : y′ : zs′)
put _ _ = fail "Update violates complement."
```

Note that for this function, $put\ s\ v'$ fails if (and only if) $length\ v' \neq length\ (get\ s)$.

3 Semantic Bidirectionalization

We will present our combined bidirectionalization technique only for lists, and only for fully polymorphic functions to bidirectionalize. So from now on, let

$$get :: [\alpha] \to [\alpha]$$

be fixed but arbitrary.

Following the technique of Voigtländer (2009) for this special case, we define *put* as follows, using some functions from module Data.IntMap.

```
put :: (Monad μ, Eq α) ⇒ [α] → [α] → μ [α]
put s v′ =
  do let t = [0 .. length s − 1]
     let g = IntMap.fromDistinctAscList (zip t s)
     h ← assoc (get t) v′
     let h′ = IntMap.union h g
     return (map (fromJust ∘ flip IntMap.lookup h′) t)

assoc :: (Monad μ, Eq α) ⇒ [Int] → [α] → μ (IntMap α)
assoc []       []       = return IntMap.empty
assoc (i : is) (b : bs) =
  do m ← assoc is bs
     case IntMap.lookup i m of
       Nothing → return (IntMap.insert i b m)
       Just c  → if b == c
                   then return m
                   else fail "Update violates equality."
assoc _        _        = fail "Update changes the length."
```

Example 2. We continue Example 1. Just as was the case for syntactic bidirectionalization, $put\ s\ v'$ fails if and only if $length\ v' \neq length\ (get\ s)$. Indeed, the two versions of *put* are semantically equivalent (at type $[\tau] \to [\tau] \to$ Maybe $[\tau]$, for τ that is an instance of Eq).

4 Refactoring Semantic Bidirectionalization

Our aim is to combine syntactic and semantic bidirectionalization into a technique that will represent a significant improvement over both. As preparation, we refactor the technique of Voigtländer (2009).

From now on, assume that for every $n :: \mathsf{Int}$, $get\ [0\,..\,n]$ contains no duplicates. We call this property *semantic linearity*. It will clearly be fulfilled if *get*'s syntactic definition is linear.

We define

$$put_{\text{linear}} :: \mathsf{Monad}\ \mu \Rightarrow [\alpha] \to [\alpha] \to \mu\ [\alpha]$$

like *put* (but note the different type), except that the call to *assoc* is replaced by a call, with the same arguments, to the following function:

$$
\begin{aligned}
&assoc' :: \mathsf{Monad}\ \mu \Rightarrow [\mathsf{Int}] \to [\alpha] \to \mu\ (\mathsf{IntMap}\ \alpha)\\
&assoc'\ [\,]\qquad\quad [\,]\qquad\ \ = return\ \mathsf{IntMap}.empty\\
&assoc'\ (i : is)\ (b : bs) = \mathbf{do}\ m \leftarrow assoc'\ is\ bs\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ return\ (\mathsf{IntMap}.insert\ i\ b\ m)\\
&assoc'\ _\qquad\quad\ \ _\qquad\quad\ \ = fail\ \text{“Update changes the length.”}
\end{aligned}
$$

The proof of the following theorem is very similar to the combination of those for Theorems 1 and 2 of Voigtländer (2009), additionally using semantic linearity of *get* in a straightforward way.

Theorem 1. *For every type* τ, $put_{\text{linear}} :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$ *is consistent for* $get :: [\tau] \to [\tau]$.

We now refactor put_{linear} to make the treatment of shapes (list lengths) explicit. To that end, we first define $sput_{\text{naive}}$ as follows:

$$
\begin{aligned}
&sput_{\text{naive}} :: \mathsf{Monad}\ \mu \Rightarrow \mathsf{Int} \to \mathsf{Int} \to \mu\ \mathsf{Int}\\
&sput_{\text{naive}}\ l_s\ l_{v'} = \mathbf{if}\ l_{v'} == length\ (get\ [0\,..\,l_s - 1])\\
&\qquad\qquad\qquad\quad \mathbf{then}\ return\ l_s\\
&\qquad\qquad\qquad\quad \mathbf{else}\ fail\ \text{“Update changes the length.”}
\end{aligned}
$$

Using the above function, we then define put_{refac} as follows:

$$
\begin{aligned}
&put_{\text{refac}} :: \mathsf{Monad}\ \mu \Rightarrow [\alpha] \to [\alpha] \to \mu\ [\alpha]\\
&put_{\text{refac}}\ s\ v' =\\
&\quad \mathbf{do}\ \mathbf{let}\ l_s = length\ s\\
&\qquad\ \ \mathbf{let}\ g = \mathsf{IntMap}.fromDistinctAscList\ (zip\ [0\,..\,l_s - 1]\ s)\\
&\qquad\ \ l' \leftarrow sput_{\text{naive}}\ l_s\ (length\ v')\\
&\qquad\ \ \mathbf{let}\ t = [0\,..\,l' - 1]\\
&\qquad\ \ \mathbf{let}\ h = fromDistinctList\ (zip\ (get\ t)\ v')\\
&\qquad\ \ \mathbf{let}\ h' = \mathsf{IntMap}.union\ h\ g\\
&\qquad\ \ return\ (map\ (fromJust \circ flip\ \mathsf{IntMap}.lookup\ h')\ t)
\end{aligned}
$$

$$fromDistinctList = \mathsf{IntMap}.fromList$$

Our motivation for this refactoring is that we make explicit, in $sput_{\text{naive}}$, what happens on the shape level, namely that only updated views with the same length as the original view can be accepted, and that the length of the source will never be changed. By "playing" with $sput_{\text{naive}}$, we can change that behavior.

5 Combining Syntactic and Semantic Bidirectionalization

Our key idea is to replace $sput_{\text{naive}}$ from Section 4 by an arbitrary shape-bidirectionalizer.

We define *sget* as follows:

$$\begin{aligned}&sget :: \mathsf{Int} \to \mathsf{Int}\\&sget\ l_s = length\ (get\ [0 \mathinner{..} l_s - 1])\end{aligned}$$

The point, later, will be that one can also directly derive a simplified syntactic definition for *sget* from a given definition for *get*. But for the moment, we simply take the above definition.

Next, we assume that some function *sput* is given, with the following type:

$$sput :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Maybe\ Int}\,,$$

and that *sput* is consistent for *sget* (where we only consider the non-negative values of type Int). We now define put_{comb} as follows.

$$\begin{aligned}
&put_{\text{comb}} :: \mathsf{Monad}\ \mu \Rightarrow [\alpha] \to [\alpha] \to \mu\ [\mathsf{Maybe}\ \alpha]\\
&put_{\text{comb}}\ s\ v' =\\
&\quad \textbf{do let } l_s = length\ s\\
&\qquad \textbf{let } g = \mathsf{IntMap}.fromDistinctAscList\ (zip\ [0 \mathinner{..} l_s - 1]\ s)\\
&\qquad l' \leftarrow maybe\ (fail\ \text{“Could not deal with length change.”})\\
&\qquad\qquad\qquad return\\
&\qquad\qquad\qquad (sput\ l_s\ (length\ v'))\\
&\qquad \textbf{let } t = [0 \mathinner{..} l' - 1]\\
&\qquad \textbf{let } h = fromDistinctList\ (zip\ (get\ t)\ v')\\
&\qquad \textbf{let } h' = \mathsf{IntMap}.union\ h\ g\\
&\qquad return\ (map\ (flip\ \mathsf{IntMap}.lookup\ h')\ t)
\end{aligned}$$

The proof of the following theorem is (again) very similar to that by Voigtländer (2009) for his Theorems 1 and 2, but of course additionally uses the assumption that *sput* is consistent for *sget*.

Theorem 2. *Let* τ *be a type.*

- *For every* $s :: [\tau]$, $put_{\text{comb}}\ s\ (get\ s) :: \mathsf{Maybe}\ [\mathsf{Maybe}\ \tau] \equiv \mathsf{Just}\ (map\ \mathsf{Just}\ s)$.
- *For every* $s, v' :: [\tau]$ *and* $s' :: [\mathsf{Maybe}\ \tau]$, *if* $put_{\text{comb}}\ s\ v' :: \mathsf{Maybe}\ [\mathsf{Maybe}\ \tau] \equiv \mathsf{Just}\ s'$, *then* $get\ s' \equiv map\ \mathsf{Just}\ v'$.

Compared to put_{refac}, we see that put_{comb} uses an extra Maybe type constructor in the output list type. This is done to deal with positions in the output list for which no data is known, neither from the original source nor from the updated view. It is usually more convenient to work with a default value, so we define a function *dput* as follows:

$$
\begin{aligned}
&dput :: \mathsf{Monad}\ \mu \Rightarrow \alpha \rightarrow [\alpha] \rightarrow [\alpha] \rightarrow \mu\ [\alpha] \\
&dput\ d\ s\ v' = \mathbf{do}\ s' \leftarrow put_{\text{comb}}\ s\ v' \\
&\qquad\qquad\qquad\quad return\ (map\ (maybe\ d\ id)\ s')
\end{aligned}
$$

The following statement is then a relatively direct consequence of Theorem 2.

Corollary 1. *For every type* τ *and* $d :: \tau$, $dput\ d :: [\tau] \rightarrow [\tau] \rightarrow \mathsf{Maybe}\ [\tau]$ *is consistent for* $get :: [\tau] \rightarrow [\tau]$.

Now, if from a given *get*, we make an *sget*, and find a good *sput* for it, then *dput* will also be good for *get*. This is where we can now plug in the work of Matsuda et al. (2007), as a black box. For functions *get* that are polymorphic and at the same time satisfy the syntactic restrictions imposed by Matsuda et al.'s technique, we can use that technique for deriving *sput* from *sget*.

6 Analysis of the Example

We detail the execution of the just introduced combination idea on the example considered in Sections 2 and 3. We have seen that for

$$
\begin{aligned}
&get :: [\alpha] \rightarrow [\alpha] \\
&get\ [\,] \qquad\qquad = [\,] \\
&get\ [x] \qquad\quad\; = [\,] \\
&get\ (x : y : zs) = y : (get\ zs)
\end{aligned}
$$

both syntactic and semantic bidirectionalization on their own lead to quite limited updatability. Namely, $put\ s\ v'$ only succeeds if $length\ v' = length\ (get\ s)$.

On the other hand, for the combination of the two techniques, we can proceed as follows. The *sget* corresponding to *get* looks as follows:

$$
\begin{aligned}
&sget :: \mathsf{Int} \rightarrow \mathsf{Int} \\
&sget\ 0 \qquad\quad\; = 0 \\
&sget\ 1 \qquad\quad\; = 0 \\
&sget\ (zs + 2) = (sget\ zs) + 1
\end{aligned}
$$

Clearly, a pretty straightforward syntactic program transformation could be used to obtain that definition automatically. For it, the syntactic bidirectionalization method of Matsuda et al. (2007) produces the following complement function:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{SCompl} = \mathsf{SC}_1 \mid \mathsf{SC}_2 \\
&scompl :: \mathsf{Int} \rightarrow \mathsf{SCompl}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{scompl}\ 0 &&= \mathsf{SC}_1\\
&\mathit{scompl}\ 1 &&= \mathsf{SC}_2\\
&\mathit{scompl}\ (\mathit{zs}+2) &&= \mathit{scompl}\ \mathit{zs}
\end{aligned}
$$

Note that the move from $[\alpha]$ to Int in $\mathit{get} \mapsto \mathit{sget}$ has obviated the need to collect any dropped variables in the complement function. As a consequence, with the help of range analysis (telling that the right-hand side of the recursive equation never overlaps, for no instantiation of variables, with any other right-hand side), no data constructor is necessary around the recursive call. For the two non-recursive equations, different data constructors are needed, because the ranges of the original right-hand sides overlap.

Tupling of *sget* and *scompl* leads to:

$$
\begin{aligned}
&\mathit{spaired} :: \mathsf{Int} \to (\mathsf{Int}, \mathsf{SCompl})\\
&\mathit{spaired}\ 0 &&= (0, \quad \mathsf{SC}_1)\\
&\mathit{spaired}\ 1 &&= (0, \quad \mathsf{SC}_2)\\
&\mathit{spaired}\ (\mathit{zs}+2) &&= (v+1, c)\\
&\quad \textbf{where}\ (v, c) = \mathit{spaired}\ \mathit{zs}
\end{aligned}
$$

Inversion gives:

$$
\begin{aligned}
&\mathit{sinv} :: \mathsf{Monad}\ \mu \Rightarrow (\mathsf{Int}, \mathsf{SCompl}) \to \mu\ \mathsf{Int}\\
&\mathit{sinv}\ (0, \quad \mathsf{SC}_1) &&= \mathit{return}\ 0\\
&\mathit{sinv}\ (0, \quad \mathsf{SC}_2) &&= \mathit{return}\ 1\\
&\mathit{sinv}\ (v+1, c) &&= \textbf{do}\ \mathit{zs} \leftarrow \mathit{sinv}\ (v, c)\\
& && \qquad\quad \mathit{return}\ (\mathit{zs}+2)
\end{aligned}
$$

and finally,

$$
\begin{aligned}
&\mathit{sput} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Maybe\ Int}\\
&\mathit{sput}\ s\ v' = \mathit{sinv}\ (v', \mathit{scompl}\ s)
\end{aligned}
$$

can be fused to:

$$
\begin{aligned}
&\mathit{sput} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Maybe\ Int}\\
&\mathit{sput}\ 0 && 0 &&= \mathit{return}\ 0\\
&\mathit{sput}\ 1 && 0 &&= \mathit{return}\ 1\\
&\mathit{sput}\ (\mathit{zs}+2) && 0 &&= \mathit{sput}\ \mathit{zs}\ 0\\
&\mathit{sput}\ s && (v'+1) &&= \textbf{do}\ \mathit{zs} \leftarrow \mathit{sput}\ s\ v'\\
& && && \qquad\quad \mathit{return}\ (\mathit{zs}+2)
\end{aligned}
$$

The benefit of the combination of syntactic and semantic bidirectionalization can be observed by comparing *dput* as obtained from the above *sput*-function to the function *put* from Example 1 in Section 2 (which we have seen is equivalent to *put* as obtained via semantic bidirectionalization). Here are a few representative

calls and their results:

s	v'	$put\ s\ v'$	$dput$ ' ' $s\ v'$
"abcd"	"x"	Nothing	Just "ax"
"abcd"	"xy"	Just "axcy"	Just "axcy"
"abcd"	"xyz"	Nothing	Just "axcy z"
"abcd"	"xyzv"	Nothing	Just "axcy z v"
"abcde"	"x"	Nothing	Just "axc"
"abcde"	"xy"	Just "axcye"	Just "axcye"
"abcde"	"xyz"	Nothing	Just "axcyez "
"abcde"	"xyzv"	Nothing	Just "axcyez v "

Note that when $length\ v' \neq length\ (get\ s)$, $dput$ ' ' $s\ v'$ extends or shrinks the source list by a number of elements that is a multiple of two (to preserve the remainder modulo two, as fixed via *scompl*). All updates can be successfully handled, in contrast to all the versions of *put* we have considered for this example before!

7 Conclusion

We have presented an approach for combining the bidirectionalization methods of Matsuda et al. (2007) and Voigtländer (2009). By separating shape from content, we exploit the respective strengths of the two previous methods maximally. The key insight is that when we simplify the problem of explicit bidirectionalization by posing it only on the shape level (going from *get* to *sget*), the existing syntactic technique can give far better results than for the general problem. The existing semantic technique does the rest.

The full paper considers further examples, and develops ideas for making the approach more practical and more widely applicable.

References

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(3):557–575, 1981.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming, Proceedings*, pages 47–58. ACM Press, 2007.

A. Pettorossi. Transformation of programs and use of tupling strategy. In *Informatica, Proceedings*, pages 1–6, 1977.

J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

Towards a Jitting VM for Prolog Execution

Carl Friedrich Bolz, Michael Leuschel, and David Schneider

Lehrstuhl Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Düsseldorf, Germany

Abstract. Most Prolog implementations are implemented in low-level languages such as C and are based on a variation of the WAM instruction set [5], which enhances their performance but makes them hard to write. We present a high-level continuation-based [3] Prolog interpreter written in RPython, a restricted subset of Python [1]. This interpreter is annotated with hints, so that it can be fed through the PyPy tracing JIT generator, which incorporates partial evaluation techniques [2]. The resulting Prolog implementation is surprisingly efficient: it clearly outperforms existing implementations of Prolog in high-level languages like Java [4]. Moreover, on some benchmarks, our system outperforms state-of-the-art WAM-based Prolog implementations. The talk tries to show that PyPy can indeed form the basis for implementing programming languages other than Python. Furthermore, we believe that our results showcase the great potential of the tracing JIT approach for logic programming languages like Prolog.[1]

References

1. D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64, Montreal, Quebec, Canada, 2007. ACM.
2. C. F. Bolz, A. Cuni, M. Fijakowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, Genova, Italy, 2009. ACM.
3. T. Lindgren. A Continuation-Passing style for Prolog. In *Symposium on Logic Programming*, pages 603–617, 1994.
4. G. Piancastelli, A. Benini, A. Omicini, and A. Ricci. The architecture and design of a malleable object-oriented prolog engine. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 191–197, Fortaleza, Ceara, Brazil, 2008. ACM.
5. P. van Roy. 1983-1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19:385–441, 1994.

[1] This research is partially supported by the BMBF funded project PyJIT (nr. 01QE0913B; Eureka Eurostars).

An Adaptive Compilation Framework for Generic Data-Parallel Array Programming*

Clemens Grelck[1], Tim van Deurzen[1], Stephan Herhut[2], and Sven-Bodo Scholz[2]

[1] University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck,t.v.deurzen@uva.nl
[2] University of Hertfordshire, School of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{s.a.herhut,s.scholz}@herts.ac.uk

Abstract. Generic array programming abstracts from structural properties of arrays, such as rank (number of axes/dimensions) and shape (number of element along each axis/dimension). This allows for abstract program specifications and, as such, is desirable from a software engineering perspective. However, generic programming in this sense does have an adverse effect on runtime performance when executed naively. Static compiler analyses and transformations aim at reconciling software engineering desires for generic code with runtime performance requirements. However, they are bound to fail whenever the required information is not available until runtime.
We propose a compilation framework that overcomes the inherent limitations of static analysis by incrementally adapting a running program to the structural properties of the arrays it operates on. This is achieved by partial recompilation of code at runtime, when all structural properties of arrays are known, and dynamic relinking of the running program with dynamically generated code. We sketch out the general compilation framework architecture and discuss implementation aspects.

1 Introduction

Optimising compilers reconcile the programmer's desire for generic, re-usable programs adhering to software engineering principles such as abstraction and composition and the necessities of executable code to achieve high runtime performance in sequential and, increasingly important, (implicitly) parallel execution. Optimising compilers analyse program code and infer static properties that trigger program transformations as appropriate. The effectiveness of static analysis, however, is essentially limited by two aspects: Firstly, the quality of the analyses implemented in the compiler; and secondly, the availability of required information compile time. As an example of a common compiler optimisation

* This work was supported by the European Union through the FP-7 project ADVANCE (Asynchronous and Dynamic Virtualisation through Performance Analysis to Support Concurrency Engineering), grant no. FP7 248828.

consider loop unrolling. Loop unrolling is triggered by a compiler analysis that infers the trip count of the loop. The compiler then unrolls the loop if the trip count is below a given threshold. However, even the best static analysis is bound to fail if the expression defining the trip count depends on values that are unknown at compile time. For example, they could be obtained from the execution environment at runtime (input), or they may be determined by code located in a different compilation unit.

We propose an adaptive compilation framework for the data-parallel functional array language SaC [1]. SaC advocates shape- and rank-generic programming on multidimensional arrays, i.e. SaC supports functions that abstract from the concrete shape (extent along dimensions) and even from the concrete rank (number of dimensions) of argument arrays and that yield result arrays whose shape and (!) rank are determined by the function itself. Depending on the amount of compile time structural information the type system of SaC distinguishes three classes of arrays at runtime:

- *Arrays of Known Shape (AKS)* where both rank and shape are statically available;
- *Arrays of Known Dimensionality (AKD)* where the rank is statically available, but the concrete shape is computed dynamically; and
- *Arrays of Unknown Dimensionality (AUD)* where neither rank nor shape are known to the compiler.

From a software engineering point of view it is (usually) desirable to specify functions on the most general input type(s) to maximise opportunities for code reuse. Typical examples for rank-generic operations are extensions of scalar operators (arithmetic, logical, relational, etc) to entire arrays in an element-wise way or common structural operations like shifting and rotation along one or multiple axes of an array. In fact, rank-generic functions prevail in the extensive SaC standard array library.

However, genericity comes at a price. In comparison to non-generic code the runtime performance of equivalent operations is substantially lower for shape-generic code and again substantially lower for rank-generic code [2]. The reasons are manifold and their individual impact operation-specific, but three categories can be identified notwithstanding: Firstly, generic runtime representations of arrays need to be maintained, and generic code tends to be less efficient, e.g. no static nesting of loops can be generated to implement a rank-generic multi-dimensional array operation. Secondly, many of the SaC-compiler's advanced optimisations [3, 4] are just not as effective for generic code because the necessary code properties to trigger certain program transformations simply cannot be inferred. Thirdly, in automatically parallelised code [5] many organisational decisions must be postponed until runtime; the ineffectiveness of optimisations leads to excessive barrier synchronisation and superfluous communication.

In order to reconcile the desires for generic code and high runtime performance, the SaC-compiler aggressively specialises rank-generic code into shape-generic code and shape-generic code into non-generic code. However, regardless of the effort put into compiler analyses for rank and shape specialisation, this

approach is fruitless if the the necessary rank and shape information is simply not available at compile time for whatever reason. Data may be read from a file at runtime, or SaC code is called externally from a non-SaC environment via the `sac4c` foreign language interface [6]. This gains further relevance as we use SaC in conjunction with the component-based coordination language S-Net [7].

To mitigate the negative effect of generic code on runtime performance where specialisation is not an option for one or more of the aforementioned reasons, we propose an adaptive compilation framework that incrementally adapts shape- and rank-generic code to the concrete shapes and ranks used in a specific program instantiation. Our approach is motivated by the observation that the number of different array shapes that effectively appear in generic array code, although theoretically unbounded, often is relatively small in practice.

What sets our adaptive compilation framework apart from existing just-in-time compilation and dynamic optimisation/code tuning approaches is twofold. Firstly, we dynamically adapt generic code to structural properties of the data it operates on, whereas just-in-time compilation of byte code (or similar) aims at adapting code to the execution environment, e.g. by generating native machine code. The second and probably more far-reaching difference is that we inherently assume a multicore execution environment where computing resources are available in abundance and can often not completely exploited by a running program in an efficient way. Although the SaC-compiler is equipped with very effective implicit parallelisation technology [5], experience says that the difference between using 30 cores of a 32-core machine and using all cores for running a given program is often marginal because the additional overhead for organising parallel execution more and more outweighs the benefit with each core joining in into collaborative execution. At this point we propose to set apart a small (configurable) number of cores for the purpose of incrementally adapting the binary code base to the array shapes actually appearing during a program run. Our approach takes dynamic recompilation out of the critical path of an application. This property is instrumental in using a heavy-weight, highly optimising compiler like `sac2c` in an online setting.

The remainder of the paper is organised as follows. Section 2 provides a few more details on the design of Single Assignment C. We present our ideas on adaptive compilation in more detail in Section 3 and discuss implementation issues in Section 4. Eventually, we browse through related work in Section 5 and draw conclusions in Section 6.

2 SAC in a Nutshell

As the name "Single Assignment C" suggests, SaC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This is meant to facilitate familiarisation for programmers who rather have a background in imperative languages than in declarative languages. Core SaC is a functional, side-effect free subset of C: we interpret assignment sequences as nested let-expressions, branching constructs as conditional expressions and loops

as syntactic sugar for tail-end recursive functions; details on the design of SaC and the functional interpretation of seemingly imperative code can be found in [1]. Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the operational behaviour expected by imperative programmers. This allows programmers to choose their favourite interpretation of SaC code while the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation [5].

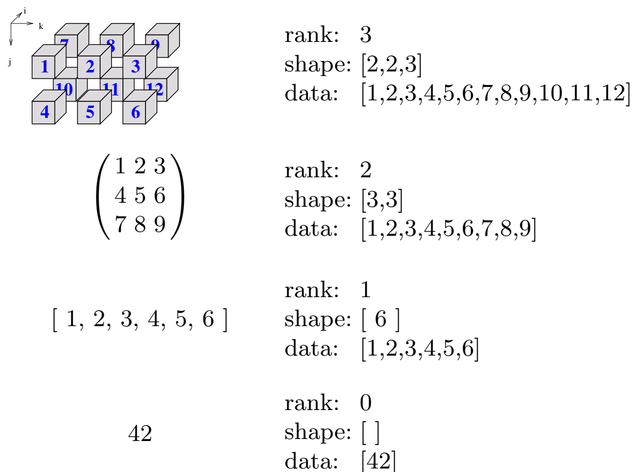

Fig. 1: Truly multidimensional arrays in SaC and their representation by data vector, shape vector and rank scalar

On top of this language kernel SaC provides genuine support for processing truly multidimensional and truly stateless/functional arrays advocating a shape- and rank-generic style of programming. Conceptually, any SaC expression denotes an array; arrays can be passed to and from functions call-by-value. A multidimensional array in SaC is represented by a *rank scalar* defining the length of the *shape vector*. The elements of the shape vector define the extent of the array along each dimension and the product of its elements defines the length of the *data vector* The data vector contains the array elements (in row-major order). Fig. 1 shows a few examples for illustration. Notably, the underlying array calculus nicely extends to scalars, which have rank zero and the empty vector as shape vector. Furthermore, we achieve a complete separation between data assembled in an array and the structural information (rank and shape).

The type system of SaC (at the moment) is monomorphic in the element type of an array, but polymorphic in the structure of arrays, i.e. rank and shape. As illustrated in Fig. 2, each element type induces a conceptually unbounded number of array types with varying static structural restrictions on arrays. These

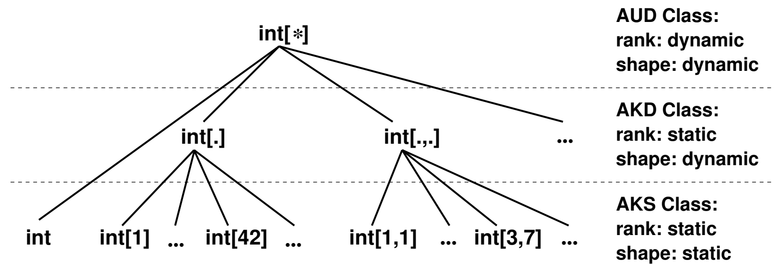

Fig. 2: Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

array types essentially form a hierarchy with three levels. On the lowest level we find non-generic types that define arrays of fixed shape, e.g. `int[3,7]` or just int. On an intermediate level of genericity we find arrays of fixed rank, e.g. `int[.,.]`. And on the top of the hierarchy we find arrays of any rank, e.g. `int[*]`. The hierarchy of array types induces a subtype relationship, and SaC supports function overloading with respect to subtyping.

SaC only provides a small set of built-in array operations. Essentially, there are primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`) or its shape (`shape(`*array*`)`). A selection facility provides access to individual elements or entire subarrays using a familiar square bracket notation: *array*`[`*idxvec*`]`. The use of a vector for the purpose of indexing into an array is crucial in a rank-generic setting: if the number of dimensions of an array is left unknown at compile time, any syntax that uses a fixed number of indices (e.g. comma-separated) makes no sense whatsoever.

While simple (one-dimensional) vectors can be written just like in C and other C-inspired languages, i.e. as a comma-separated list of expressions enclosed in square brackets, any rank- or shape-generic array is defined by means of WITH-loop expressions. In fact, the WITH-loop is a versatile SaC-specific array comprehension or map-reduce construct. Since the ins and outs of WITH-loops are not essential to know for reading the rest of the paper, we skip any detailed explanation here and refer the interested reader to [1] for a complete account.

3 Adaptive Compilation Framework

The architecture of our adaptive compilation framework is sketched out in Fig. 3. On the bottom of the figure we have an executable (binary) SaC program generated by our SaC compiler `sac2c`. It (generally) consists of binary versions of shape-specific, shape-generic and rank-generic functions. Any shape-generic or rank-generic function, however, is called indirectly through a *dispatch function* that selects the correct instance of the function to be executed in the presence of function overloading by the programmer and static function specialisation

by the compiler. This dispatch function serves as an ideal hook to add further instances (specialisations) of functions created at runtime. Since adding more and more instances also affects function dispatch itself, we need to change the actual dispatch function whenever we add further instances. To achieve this we no longer call the dispatch function directly, but through a pointer indirection that allows us to exchange the dispatch function dynamically. We call this the *dispatch function registry*.

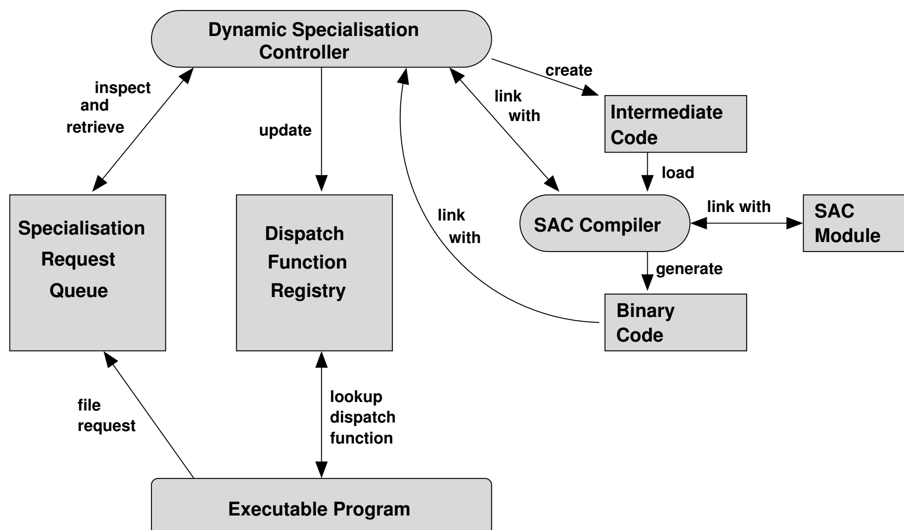

Fig. 3: Architecture of our adaptive compilation framework

Before actually calling the dispatch function retrieved from the registry, we also file a specialisation request in the *specialisation request queue*. Next to the function name, the module name where the function originates from, etc, this request contains the concrete shape parameters of all generic parameters of that function. Queueing a specialisation request is a very lightweight operation. This makes sure that the original program execution is delayed by adaptive recompilation as little as possible.

In the same process that runs the *executable program* one thread is set apart to run the *dynamic specialisation controller*. This is in charge of the main part of the adaptive compilation infrastructure; it runs concurrently with the program itself. The dynamic specialisation controller inspects the specialisation request queue and retrieves specialisation requests as they appear. It first checks whether the specialisation requested already exists or is currently in the process of being constructed. If so, the request is just discarded. Otherwise, the dynamic specialisation controller creates the (compiler-) intermediate representation of a new SaC-module. This consists among others of an import-statement of the func-

tion symbol from the original module and specialisation directive to the compiler generated from the specialisation request data.

The dynamic specialisation controller also links the binary executable with the entire SaC-compiler `sac2c`, which already comes as a shared library. So, having created the stub module, the dynamic specialisation controller effectively turns itself into the SaC-compiler. As such, it now dynamically links with the (compiled) module the function stems from and retrieves a partially compiled intermediate representation of the function's implementation and potentially further dependent code from the binary of the module. This, again, exploits a standard feature of the SaC module system that was originally developed to support inter-module (compile time) optimisation [8].

Eventually, the SaC-compiler (with the help of a backend C compiler) generates another shared library containing binary versions of the specialised function(s) and one or more new dispatch function taking the new specialisations into account in their decision. Following the completion of the SaC-compiler, the dynamic specialisation controller regains control. It still has two tasks to do before attending to the next specialisation request. Firstly, it links the running process with the newly created shared library. Secondly, it updates the dispatch function registry with the new dispatch function(s) from that library. As a consequence, any subsequent call to that function originating from the running program will directly be forwarded to the specialised instance rather than the generic version and benefit from (potentially) substantially higher runtime performance without further overhead.

Our adaptive compilation framework is carefully designed such that the associated runtime overhead in the executable program is minimal. Essentially, it boils down to an indirection in calling the dispatch function and the filing of a specialisation request. All the remaining work is done concurrently to the execution of the program itself by one or more dynamic specialisation controllers. Our assumption is that these run on different processors or cores and as such use resources that would otherwise remain unused or whose exploitation for running the program itself would at most have a marginally positive effect on overall performance.

4 Implementation Aspects

For our prototype implementation, we have extended the existing SaC compiler and runtime system in three aspects. Firstly, we have modified the code generation of the compiler to provide the required profiling information to the specialisation controller. Secondly, we have implemented hooks in the compiler that allow the specialisation controller to initiate the specialisation of requested functions. And last but not least, we have implemented the specialisation controller itself as part of the SaC runtime system.

To control the collection and reporting of runtime information, we have added an additional flag to the compiler. The option `-runtimespec` will enable the required extension to code generation. The produced executable differs from

standard executables in three main aspects. Firstly, we extend the dynamic dispatch code that is generated for function applications where we cannot statically determine the matching instance. Additionally to dynamically choosing the appropriate instance, the extended dispatch code communicates the actual parameter shapes found at runtime to the specialisation controller. As functions are always dispatched statically with respect to the base types of arguments, this information mainly comprises the rank and dimensionality of each argument. Furthermore, we send the index into the global registry that corresponds to the called function. This information is used two-fold: It allows us to later identify which entry in the registry to update. More importantly, however, the index can be used as a unique token to identify the function to specialise. We use this token to lookup the information that is required in the communication with the compiler. Note here that we send that shape information blindly. In particular, we do not perform any checks on whether a specialisation is actually necessary. To keep the runtime overhead within the actual program as low as possible, we offload these checks into the specialisation controller.

Secondly, we reroute all function applications via the central registry. By using the registry instead of calling functions directly, we are able to dynamically rebind function applications to updated implementations. All that is required is an update to the function pointer in the registry. Lastly, we have modified the static dispatch, as well. If no runtime specialisation is requested, we usually dispatch a function call statically as soon as we can identify a single matching instance. However, such instance could still be relatively generic. For instance, a most-generic instance might be defined for arguments of unknown dimensionality (AUD). When using runtime specialisation, such a dispatch is not desirable. As we use the dynamic dispatch code to trigger runtime specialisation, an application that has been statically dispatched would never be optimised. Therefore, when runtime specialisation is enabled, we only dispatch a function application statically if we were able to derive full shape knowledge for the arguments and the matching instance is an exact match for those shapes. In those situations, no further specialisation would be possible.

The second work package in our implementation, the special version of the SaC compiler that creates new specialisations on the fly, turned out to require only limited implementation effort. We mainly make use of existing compiler features. The heavy lifting of creating the actual specialisations and updated dynamic dispatch code is performed by the SaC module system [8]. To allow for specialisation across module boundaries at compile time, SaC modules already contain, apart from the compiled binary, a condensed representation of the definition of each function. We reuse the same information for the creation of specialisations at runtime. Furthermore, we use the ability of the SaC module system to extend functions from a different module by new instances, forming an updated version of the function in the current module. Lastly, we use a language feature of the SaC-compiler, i.e. *forced specialisations*, to express the runtime specialisation request at the language level. To ensure that a function is specialised for certain argument types, the programmer can simply provide

the desired function signature prepended with the keyword `specialize`. This will trigger a specialisation to that signature at compile time of the module or program that contains the `specialize` directive.

As an example for the interplay of these three features during runtime specialisation, consider the following scenario: Assume we have a function `add` that expects two arguments, yields one return value and is defined in module `Math`. We now want to specialise this function for two arguments of type `int[7]`. This can be achieved by the following regular SaC code:

```
module DynSpec1;
import Math : {add};
export : all;
specialize int[*] add( int[7] x, int[7] y);
```

First, we create a new container for the resulting extended function `add` in form of the module `DynSpec1`. The module name is of no importance as long it is unique. Next, we trigger the module system to load the existing instances of the function `add` from its defining module `Math` by means of an `import` directive. As we want to make the resulting instances available to the running program, we flag them for export. Lastly, we add a specialisation directive to ensure that the new function `add` in module `DynSpec1` contains the desired instance for 7-element integer vectors. When the above sample code is compiled, the vanilla SaC compiler already creates a new module with the desired instances. That new module can then be used as new provider of the `add` function instead of the original `Math` module. In particular, the new module can be used as source for further specialisations of the function `add`. The same technique as in the above example can be applied where yet another module is created that imports the existing instances from the `DynSpec1` module created in the first round.

All that needs to be done to exploit the existing machinery for runtime specialisation is to create the above code, at least in form of an abstract syntax tree in memory, start the compilation process and dynamically add the resulting library. This functionality, amongst other bookkeeping, is implemented in the specialisation controller. In the simplest case, the controller dequeues a specialisation request, creates the corresponding abstract syntax tree to trigger the specialisation, enacts the compiler and collects back the updated library. That library is then dynamically linked to the program and the global registry is updated.

However, as the augmented program submits specialisation requests blindly, we might end up with many duplicate requests. To prevent this, the controller stores all requests and automatically disregards identical requests. As a further optimisation the controller can block multiple specialisations of the same function into a single abstract syntax tree. It suffices to include multiple `specialize` directives, one for each specialisation request. Lastly, as the controller has a global overview over the requests submitted by the program over time, it can perform a form of frequency scheduling: Those specialisation requests that are enqueued particularly often can be acted upon first.

5 Related Work

A wealth of related work can be found in the area of runtime partial evaluation, often also referred-to as dynamic specialisation. Systems such as Tempo [9, 10], Fabius [11] or DyC [12] are based on user annotations which indicate to the compiler where dynamic specialisations can be expected. These systems then generate specific runtime specialisers leading to a staged compilation process. This measure keeps the overhead introduced by the compilation at runtime low. In contrast, our approach is based on the idea to specialise programs concurrently and asynchronously. This allows us to apply the full-fledged compiler to an annotated source code.

Further related work concerns approaches that operate on the code that is being executed. They typically analyse different instruction paths at runtime. When it turns out that a certain path is used more frequently, this path is further optimised. Dynamo [13] and DynamoRio [14] both identify hot spots in programs. When a hotspot has been detected, execution is paused and optimised code is generated for it. As interpreting is expensive, Dynamo tries to store as many optimised traces as possible in a trace cache. The next time a trace is executed Dynamo points it to the optimised code stored in its cache.

Another approach, ADORE (Adaptive Object code RE-optimisation) [15], uses hardware performance monitoring to identify performance bottlenecks. Similar to the approach presented in this paper, ADORE uses two threads: One thread runs the application as it would have normally and the second thread runs the optimisation functions. However, the optimisations performed in the ADORE system primarily target insertions of data cache prefetching to improve the cache behaviour in subsequent runs.

6 Conclusion

We have presented an adaptive compilation framework for generic array programming that virtually achieves the quadrature of the circle: to program code in a generic, reuse-oriented way abstracting from concrete structural properties of the arrays involved, and, at the same time, to enjoy the runtime performance characteristics of highly specialised code when it comes to program execution. We have implemented the proposed compilation framework in the SaC compiler `sac2c` and obtained encouraging preliminary results, still any detailed evaluation is future work.

The proposed approach gives rise to a wealth of research questions. For example, given a number of available cores, what is a profitable division of cores into one group of cores that collaboratively execute the program and another group of cores that run dynamic specialisation controllers. Another area of research relates to the number of specialisations to be performed at once: just one, as described, or all pending requests at once, or anything in between. It may also be worthwhile to dynamically run the SaC compiler with a dedicated parameter set to increase compilation speed.

Last not least, dynamic specialisation only makes sense for functions that actually benefit from the availability of more detailed structural information on argument arrays. This definitely holds for computationally intensive functions, but to a much lesser extent for I/O-related functions. Discriminating functions accordingly is another interesting future research question.

References

1. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multi-threaded execution. Intern. Journal of Parallel Programming **34** (2006) 383–427
2. Kreye, D.: A Compilation Scheme for a Hierarchy of Array Types. In Implementation of Functional Languages, 13th International Workshop (IFL'01), Stockholm, Sweden, Selected Papers. LNCS 2312, Springer-Verlag (2002) 18–35
3. Grelck, C., Scholz, S.B.: SAC — From High-level Programming with Arrays to Efficient Parallel Execution. Parallel Processing Letters **13** (2003) 401–412
4. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing **32** (2006) 507–522
5. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
6. Marcussen-Wulff, N., Scholz, S.B.: On Interfacing SAC Modules with C Programs. 12th International Workshop on Implementation of Functional Languages (IFL'00), Aachen, Germany (2000) 381–386
7. Grelck, C., Scholz, S.B., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming **38** (2010) 38–67
8. Herhut, S., Scholz, S.B.: Towards Fully Controlled Overloading Across Module Boundaries. In 16th International Workshop on the Implementation and Application of Functional Languages (IFL'04), Lübeck, Germany (2004) 395–408
9. Consel, C.: A general approach for run-time specialization and its application to C. In: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, USA, ACM Press (1996) 145–156
10. Noel, F., Hornof, L., Consel, C., Lawall, J.L.: Automatic, template-based run-time specialization: Implementation and experimental study. In: International Conference on Computer Languages, IEEE Computer Society Press (1998) 132–142
11. Leone, M., Lee, P.: Dynamic specialization in the fabius system. ACM Computing Surveys **30** (1998)
12. Grant, B., Philipose, M., Mock, M., Chambers, C., Eggers, S.J.: An evaluation of staged run-time optimizations in DyC. ACM SIGPLAN Notices **34** (1999) 293–304
13. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. SIGPLAN Not. **35** (2000) 1–12
14. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. Symposium on Code Generation and Optimization (2003)
15. Lu, J., Chen, H., Yew, P.C., Hsu, W.C.: Design and implementation of a lightweight dynamic optimization system. J. Instruction-Level Parallelism **6** (2004) 1–24

Reguläres Ausdruckstheater

Sebastian Fischer, Frank Huch und Thomas Wilke

Institut für Informatik
Christian-Albrechts Universität zu Kiel

Cody, Hazel und Theo, zwei erfahrene Haskell Programmierer und ein Experte für Automatentheorie, entwickeln ein elegantes Haskellprogramm zum Matching regulärer Ausdrücke. Das Programm ist rein funktional; es rechnet in beliebigen Semiringen, was es nicht nur erlaubt, das gewöhnliche Matchingproblem zu lösen, sondern auch andere Probleme wie die Berechnung des längsten linkesten Matchings oder der Anzahl aller Matchings mit einem einzigen Algorithmus zu lösen; und durch bedarfsgesteuerte Auswertung lassen sich nicht nur reguläre Sprachen sondern beliebige kontextfreie Sprachen erkennen.

Das entwickelte Programm basiert auf einem alten Ansatz, reguläre Ausdrücke in endliche Automaten zu übersetzen und ist daher sowohl aus komplexitätstheoretischer als auch aus praktischer Sicht effizient: obwohl es sehr einfach ist, besteht es im Vergleich mit einem gerade veröffentlichten *C++* Programm zum Matching regulärer Ausdrücke.

Nahtlose und effiziente Integration großer Zahlen in eine Programmiersprache

Christian Heinlein
Studiengang Informatik
Fakultät Elektronik und Informatik
Hochschule Aalen – Technik und Wirtschaft

Programmiersprachen wie C, C++ und Java bieten unterschiedlich große Typen für ganze und Gleitkommazahlen an, z. B. `byte`, `short`, `int` und `long` für ganze Zahlen sowie `float`, `double` und eventuell `long double` für Gleitkommazahlen. Außerdem gibt es Bibliotheken wie beispielsweise die GNU Multiple Precision Library für C und C++ sowie die Klassen `BigInteger` und `BigDecimal` für Java, mit denen Berechnungen mit beliebig großen ganzen Zahlen und beliebig genauen Gleitkommazahlen durchgeführt werden können.

Aus konzeptueller Sicht ist die Bereitstellung einer Vielzahl numerischer Typen mit unterschiedlicher Größe bzw. Genauigkeit unnötig. Ein einziger ganzzahliger Typ, dessen Werte prinzipiell beliebig groß sein können, sowie ein einziger Gleitkommatyp, dessen Werte prinzipiell beliebig genau sein können, wäre ausreichend. Allerdings ist die Verwendung der entsprechenden Typen je nach Sprache relativ umständlich – beispielsweise muss man `x.add(new BigInteger("2").multiply(y))` schreiben, um die Berechnung `x + 2*y` mit Java-`BigInteger`-Werten auszudrücken – und darüber hinaus ziemlich ineffizient, sofern die tatsächlich verwendeten Werte „klein“ sind.

Im Vortrag wird gezeigt, wie in einem 32-Bit-`int`-Wert entweder eine „kleine“ ganze Zahl oder aber ein Zeiger auf eine „große“ Zahl gespeichert werden kann und wie sich die beiden Fälle effizient unterscheiden lassen, damit Operationen auf „kleinen“ Zahlen, die normalerweise deutlich überwiegen, mit möglichst geringem Zusatzaufwand ausgeführt werden können. Auf ähnliche Art und Weise kann in einem 64-Bit-`double`-Wert entweder eine Gleitkommazahl gemäß IEEE 754 mit „kleiner“ Genauigkeit oder aber ein Zeiger auf eine Zahl mit „großer“ Genauigkeit gespeichert werden.

Durch Kapselung der jeweiligen Werte in C++-Klassen mit zugehörigen überladenen Operatordefinitionen erhält man Typen zur Repräsentation beliebig großer bzw. genauer Zahlen, die syntaktisch genauso verwendet werden können wie die eingebauten Typen der Sprache und deren Operationen für „kleine“ Werte fast genauso effizient sind. Bei einer direkten Integration dieser Typen in den Compiler einer Programmiersprache sind weitergehende Optimierungsmöglichkeiten denkbar.

Values in Object-Oriented Programming Languages

Beate Ritterbach and Axel Schmolitzky

University of Hamburg, Department Informatik,
Vogt-Koelln-Str. 30, D-22527 Hamburg, Germany
+49.4122.82770 +49.40.42883 2302
beate.ritterbach@studium.uni-hamburg.de schmolitzky@acm.org

Abstract. This work suggests a notion of values that does not focus on immutability only but emphasizes another crucial property of values: un-creatability. It sketches a dedicated type constructor for value types that guarantees their essential properties and that can be incorporated in an object-oriented language. By regarding values and objects as distinct abstractions ("*not* everything is an object"), and supporting both of them safely and clearly it is possible to attain an object-functional language that enables both purely functional and stateful programming.

Keywords: Values, Value-Types, Object-functional language

0 Introduction

Our starting point are object-oriented languages. A basic principle of object-orientation is the conviction that „Everything is an object." Yet many abstractions - like numbers, points, dates or strings - do not fit within this scheme: values. This work will argue for values and for incorporating them in object-oriented languages in a way that considers the characteristics of both values and objects. It is structured as follows: 1. As a base for further work we suggest a conceptual definition which specifies four essential properties for values. 2. We claim that values require dedicated language support and we propose a type constructor that enables programmers to define and use values clearly, safely and easily. 3. We argue that supporting both values and objects - i.e. both functional and stateful programming - in one language, but clearly separating them, is a step towards an object-functional language that preserves purity of both paradigms.

1 A Concept of Values

The all-embracing object metaphor can cause problems. Some abstractions - e.g. numbers, strings, dates, IP-addresses, points, intervals, ... - do not behave like "typical" objects. Dealing with such abstractions in object-oriented languages can become intricate and error-prone. E. g. in Java, strings usually should be compared

with equals (instead of ==). Or when returning a java.util.Date from a method, the programmer should remember to make a defensive copy first, so as to avoid aliasing problems.

There are approaches that distinguish "value objects" from "reference objects" and tackle them with software engineering techniques like coding conventions or patterns, e.g. the Value Object pattern [10].

Similar approaches can be found in [3] and [4]. They manage values mainly by making them immutable, i.e. by coding classes that do not provide mutating operations. (A number of code examples can be found in [4].)

Yet it is not exactly clear what values are, and if values and immutable objects are the same thing. The methodologies in [3] and [4] describe values as "small simple objects", "whose instances can be copied freely", "that represent a description aspect". There is work that considers values from a different perspective, such as efficient implementation [1] or that subsumes them under the more general concept of relation types [12], and here also only immutability is mentioned as their major property. Before further work on values will be attempted precise criteria for being a value are given.

Our definition for values follows MacLennan [5], who describes values as timeless abstractions and who lists the following properties:

1. Immutability/ 2. Un-creatability:
Values can not be changed, and they can neither be created nor destroyed.

3. Side-effect-freeness/ 4. Referential transparency:
Value operations do not engender side-effects,
and when called twice with the same parameters they always yield the same result.

1. to 4. should be understood as *conceptual* properties, exhibited by values and value operations towards clients, independent of the underlying implementation.

Definitions, by their nature, are not correct or incorrect, they serve as a starting point for further work. The *motivation* for this definition, i.e. for choosing 1. to 4. as the essential properties of values, are firstly primitive data types as prototypes for values, and secondly the trial to state more precisely the notion of timelessness.

Firstly, primitive data types like integers, floating point numbers or booleans, actually do exhibit the properties 1. to 4.: they are immutable and can not be created, their operations are free of side-effects and referentially transparent. By selecting these properties as the defining properties we regard values as an abstract concept that comprises and generalizes primitive data types. Secondly, creation and change are processes that take place in time, and thus are not compatible with the notion of values as timeless abstractions. This also refers to side effects and referential opaqueness, concepts that are time-related because they presuppose mutable state.

With this notion of values, un-creatability is considered as an essential property, unlike some previous work we do not focus on immutability only. Additionally, with properties 3. and 4., value operations are also taken into account.

Absence of side-effects and referential transparency complement each other. If an operation has no side-effects then it does not *induce* state changes. If an operation is

referentially transparent then it does not *reveal* state changes. In combination these conditions specify that values are "stateless". Both is important in a surrounding that in general does permit mutable state.

(Additionally, 3. implies 1. and 4. implies 2. It would have been possible to define values using conditions 3. and 4. only, making the definition more concise yet less descriptive.)

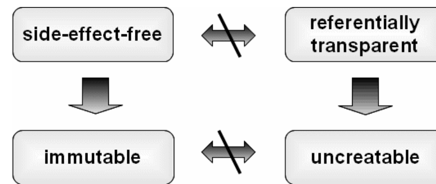

Fig. 1. Dependencies between essential properties of values

According to that definition, values are purely functional abstractions. Object oriented languages do not support values, side-effect-freeness and referential transparency well. Purely functional languages are not the solutions we are looking for, either. Many abstractions can not be regarded as values. Real world entities (such as persons, shipments, contracts, ..) usually are stateful, and functional languages do not support state and real world simulation adequately. We do not aim at making every abstraction a value and every operation a function. We want values where values are appropriate, and objects where object are appropriate.

2 Language Support for Values

In the following we focus on statically typed languages. Some object-oriented languages provide a small set of built-in value types (mostly integers, floating point numbers and characters), however this support is rather limited. In practice often the need for subject-specific value types like Date, Point, Timestamp, Complex or MonetaryAmount will arise. It should be possible to define value types freely, just as can be done with object types in object-oriented languages.

Although it is possible to model value types with object types (e.g. with the value object pattern, see above) these approaches lead to code that is error-prone, hard to understand and difficult to maintain. That is chiefly because objects must always be created, therefore un-creatability is hard to simulate.

Values require dedicated language support. It is our goal that the language support meets software-engineering qualities, in particular clarity (values should be recognizable as values in the source code), safety (the language environment should ensure the essential properties of values) and simplicity (defining and using values should be straightforward for the application programmer).

We propose a dedicated type constructor for value types, distinct from the type constructor for object types ("class") in many object-orientated languages. So as to emphasize that values and objects are separate abstractions and none is a special case of the other, it is appropriate to employ distinct keywords if possible, such as "valClass" and "objClass" respectively. (The word "class" as part of the keyword was chosen to indicate that the construct defines both a type and its implementation.)

Code-Example: Definition of a valClass "Date"

```
valClass Date {
  Int day, Int month, Int year

  Int day(Date date){ return date.day }
  Int month(Date date){ return date.month }
  Int year(Date date){ return date.year }

  Int dayOfYear(Date date) {
    ...
  }
  Date dateYMD (Int year, Int month, Int day) {
    return select(day, month, year)
  }
  Date dateYearAndDayOfYear (Int year, Int dayOfYear) {
    Int month := ...
    Int day := ...
    return dateYMD(year, month, day)
  }
  Date addDays(Date date, Int days) {
    Int day := date.day + days
    if (day > 31) ...
    return select(day, month, year)
  }
  ...
}
```

Code-Example: Use of valClass "Date"

```
var Date d1, d2, d3, d4
d1 := dateYMD(2010,1,30)             // Jan. 30th 2010
addDays(d1, 7)                       // compiletime-
                                     //  error!
d2 := addDays(d1, 7)                 // Feb. 6th 2010
if (month(d1) = month(d2)) ..        // false
d3 := dateYearAndDayOfYear(2010, 37) // Feb. 6th 2010
if (d2 = d3) ..                      // true
d4 := dateYMD(2010,2,30)             // runtime-error
```

Explanations of this code, its keywords and rules will be given in the following. Note that the code examples are written in an ad hoc syntax. It uses "=" as the symbol for equality and ":=" as the symbol for assignment, and it employs a functional notation for calling a value operation. Accessing a value's data member (possible only in the implementation of the valClass) is done with the dot-notation. A huge variety of syntactic notations would be conceivable.

Additionally it is assumed that a value type Int is already existing (no matter whether it is a built-in type or if it was defined as a valClass).

To guarantee the essential properties of values, a set of rules are postulated for valClasses, which can be checked at compile time:

- restrictions for the data members (2.1)
- value selectors, no constructors (2.2),
- equality based on the data members (2.3)
- no access to objects (2.4)

2.1 Restrictions for the Data Members

Data members of a valClass must be immutable. I.e. they can not be assigned, a statement like

```
date.month := aMonth   // error!
```

would lead to a compile-time error. The type of a data member must also be a valClass. Both conditions ensure that values are immutable.

2.2 Value Selectors, no Constructors

As a consequence of un-creatability, a valClass does not have constructors. A statement such as

```
Date d := new Date(2010,5,3)   // error!
```

is not possible, it would be highly confusing and obscure the meaning of the valClass.

However, there is the need for accessing a value. This is achieved with operations that yield a value of their own valClass, like `dateYMD` or `dateYearAndDayOfYear`. The parameters of these operations specify which particular value is requested. We call such operations *value selectors*, because they can be used to "select" a specific value from the universe of all values of that type (which are existing eternally). (Note that for built-in types there is another means to access a specific value: literals.)

Value selectors differ from object constructors fundamentally: Multiple calls of a value selector with the same parameters always yield the same value - because value selectors are value operations and therefore must be referentially transparent. Conversely, multiple calls of an object constructor with the same parameters always yield a different object.

From the client's perspective, value selectors do not differ from other value operations. E.g. they can be named arbitrarily, they can have 0 to many parameters, etc. From the perspective of the programmer of the valClass, there is something

special about value selectors: The programmer of, say, `dateYMD` needs to return an element of Date - but from the outset there is no means to do so.

A language mechanism is required; we propose a special operation that yields an element of the valClass and that specifies which one, based on its data members. We name this language provided operation the "primary selector". It is called by a special keyword ("`select`"), its parameters match the data members of the valClass in number and sequence, and its result is the value which has the specified parameters as data members. (For encapsulation reasons, we assume that the primary selector is accessible in the implementation of the valClass only.) In the Date example, `dateYMD` and `addDays` make use of the primary selector. As illustrated by `dateYearAndDayOfYear`, once a value selector has been coded, other value selectors may use it instead of calling `select` directly.

With this language mechanism it is possible to define valClasses without constructors and thus reveal the values' un-creatability in the source code.

2.3 Equality based on the Data Members

For a valClass, equality must be based on its data members - otherwise the primary selector would not be referentially transparent. Conversely, comparing objects is based on the concept of identity, usually implemented in the language environment by a storage address or a surrogate key, it does not involve the object's data members. Despite the different implementation and the different naming (for values: equality, for objects: identity) the underlying concept is the same: two expressions denoting the same thing.

In the simplest (and most common) case equality for valClasses can be defined as comparing the matching data members and associating the results with logical "AND". In the example above, the equality of Date - with input d1 and d2 - behaves as if it were implemented like this:

```
d1.day   = d2.day   AND
d1.month = d2.month AND
d1.year  = d2.year.
```

This behavior need not be programmed manually, it can be provided by the language environment. It can be invoked by the usual equality symbol ("=").

The primary selector and an equality that is based on the data members complement each other in bringing about the un-creatability of values. Together they make every element of a valClass, say, the date "October 27th 1990", appear as if it exists ever since.

2.4 No Access to Objects

For value operations, the formal parameters and the return type must also be valClasses. This can be explained as follows: If a parameter could be an object, the value operation could call the parameter's methods which in general can produce side effects or break referential transparency. Thus objects as parameters are precluded. So if an object shall be returned from a value operation it could only be produced by a

constructor call, yielding a different object for each call of the value operation and thus destroying referential transparency. It can be argued similarly that local variables must not refer to objects.

Jointly the rules 2.1 to 2.4 guarantee that the essential properties of values are ensured by the language, at compile-time. They need not be taken care of by the application programmer.

3 Unifying Functional and Object-oriented Languages

The rules described above constitute an asymmetrical dependency between values and objects. Values cannot access objects. Vice versa, objects can use values in arbitrary ways, e.g. as parameters, result types, data members and local variables.

As a result, every system is divided into a "value-kernel" and an "object-oriented shell". The value kernel consists of values and their operations, it is purely functional. The object-oriented shell consists of objects and their operations, it is characterized by mutable state and state oriented processing.

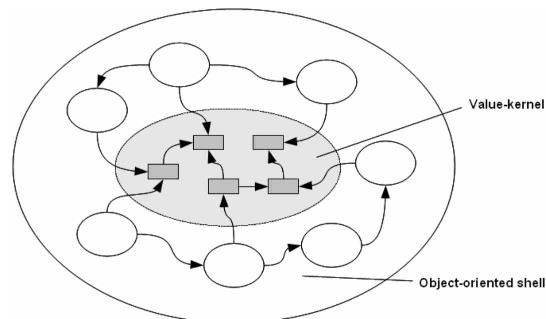

Fig. 2. Value-kernel and object-oriented shell

There are approaches to merge object-oriented and functional languages, thus making the best of both paradigms.

E.g. in Scala [7], OCaml [6], Python [9] or Ruby [11]), the core semantics is imperative, and support for functional programming is added, any function can potentially have side effects. They can be classified, according to [2], as "hybrid" object-functional languages.

In contrast, the core semantics of "pure" object-functional languages is functional, and imperative aspects of object-oriented programming are supported using monadic programming, uniqueness-types, etc. One such language is (not further developed) O'Haskell [8].

As claimed by [2], "*pure-hybrid" object-functional languages* - where procedures and functions are separate types - are not impossible. Currently no example of this kind of language has been presented.

Integrating object types and value types as outlined above into one language is a way to obtain a pure-hybrid object-functional language. In the functional kernel, referential transparency and absence of side effects are ensured, all value operations are pure functions. The object-oriented shell enables stateful processing without a need to deploy constructs like monads. Object operations can change and retrieve the corresponding objects' encapsulated state in a straightforward way.

References

1. Bacon, D.F.: Kava: A Java dialect with a uniform object model for lightweight classes. Concurrency—Practice and Experience 15(3–5), 185–206, 2003
2. Cunningham & Cunningham, Inc., Wiki, ObjectFunctional languages. URL: http://www.c2.com/cgi/wiki?ObjectFunctional.
3. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley, Boston, 2004
4. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley, Boston, 2008
5. MacLennan, B.J.: Values and Objects in Programming Languages. ACM SIGPLAN Notices 17,12, pp. 70-79, 1982
6. OCaml, URL: http://caml.inria.fr/ocaml/
7. Odersky, M. ; Spoon, L., Venners, B.: Programming in Scala. Mountain View, Calif., 2008.
8. O'Haskell, URL: http://www.haskell.org/haskellwiki/O'Haskell
9. Python. URL: http://www.python.org
10. Riehle, D.: Value object. Proceedings of the 2006 conference on Pattern languages of programs, pp. 1-6, 2006
11. Ruby. URL: http://www.ruby-lang.org/
12. Vaziri, M., Tip, F., Fink, S., Dolby, J.: Declarative Object Identity Using Relation Types. ECOOP 2007: pp. 54-78, 2007

Formalization of the Java$_\lambda$ type system

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart Campus Horb
Department of Information Technology
Florianstraße 15, D–72160 Horb
m.pluemicke@hb.dhbw-stuttgart.de

Abstract. In Java 7 the language will be expanded by closures (λ–expressions) and function types.
In our contribution we give a formal definition for an abstract syntax of a reduced language Java$_\lambda$, define the type system, and formalize the subtyping relation.
We define the set of types as an extension of the generic type definition for Java 5 types.
Finally, we give type inference rules, which describe the typings of Java$_\lambda$ expressions and statements and sketch a type inference algorithm.

1 Introduction

In several steps the Java type system is extended by features, which we know from functional programming languages. In Java 5.0 [GJSB05] generic types are introduced. Furthermore a reduced form of existential types (bounded wildcards) is introduced. For Java 7 it is announced, that function types should be introduced. Accordingly, closures (λ–expressions) should be introduced.
Type systems like this require to define syntactically large types. For example

```
Vector<? super Vector<? extends List<Integer>>>
```

or

```
##Matrix(#Matrix(Matrix,Matrix))(Matrix)
```

are correct types.
Considering all that, it is often rather inconvenient to give types like this, explictly. Furthermore it is often difficult for a programmer to decide whether such a complex type is the correct one for a given method or not.
This has caused us to develop a Java type inference system which assists the programmer by calculating types automatically. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principal types. In [Plü07] we presented a type inference algorithm for a core Java 5.0 language.
The extension of function types in Java are considered in several contributions. An early approach was given for PIZZA [OW97]. There are three newer approaches [BGGvdA,LLB,CS], which consider different aspects. Our approach is

following the Java language specification [lam10]. This approach is limited following Mark Reinhold's Blog (Principal Engineer Java SE and OpenJDK) to two key features:

- A literal syntax, for writing closures, and
- Function types, so that closures are first-class citizens in the type system.

Additionally, to integrate closures with the rest of the language and the platform two more features are needed:

- Closure conversion to implement a single-method interface or abstract class and
- Extension methods.

Our goal is to extend the type inference algorithm to the features of Java 7. For this we consider the type inference algorithm of Fuh and Mishra [FM88]. This algorithm infers types for a type system with subtyping and without additional overloading. The Java 7 preconditions are very similar.
The paper is organized as follows. In the next section we define an abstract syntax for a reduced language Java_λ. In the third section we consider the Java_λ types and the subtyping relation. In the fourth section we give the type inference rules. In the fifth section we consider the adaption of the Fuh and Mishra's type inference algorithm. Finally we close with a summary and an outlook.

2 The language

Source	$:= \mathit{class}*$
class	$:= \mathsf{Class}(\mathit{stype}, [\ \mathsf{extends}(\ \mathit{stype}\ ),]\mathit{IVarDecl}*, \underline{\mathit{FunDecl}*})$
$\mathit{IVarDecl}$	$:= \mathsf{InstVarDecl}(\ \mathit{stype}, \mathit{var}\ )$
$\underline{\mathit{FunDecl}}$	$:= \underline{\mathsf{Fun}(\ \mathit{fname}, [\boldsymbol{type}], \mathit{lambdaexpr}\ )}$
block	$:= \mathsf{Block}(\ \mathit{stmt}*\ )$
stmt	$:= \mathit{block} \mid \mathsf{Return}(\ \mathit{expr}\ ) \mid \mathsf{While}(\ \mathit{bexpr}, \mathit{block}\ )$
	$\mid \mathsf{LocalVarDecl}(\ \mathit{var}[, \boldsymbol{type}]\ ) \mid \mathsf{If}(\ \mathit{bexpr}, \mathit{block}[, \mathit{block}]\ ) \mid \mathit{stmtexpr}$
$\underline{\mathit{lambdaexpr}}$	$:= \underline{\mathsf{Lambda}(\ ((\mathit{var}[, \boldsymbol{type}]))*, (\mathit{stmt} \mid \mathit{expr})\ )}$
$\mathit{stmtexpr}$	$:= \mathsf{Assign}(\ \mathit{var}, \mathit{expr}\ ) \mid \mathsf{New}(\ \mathit{stype}, \mathit{expr}*\ ) \mid \underline{\mathsf{Eval}(\ \mathit{expr}, \mathit{expr}*\ )}$
expr	$:= \underline{\mathit{lambdaexpr}} \mid \mathit{stmtexpr} \mid \mathsf{this} \mid \mathsf{This}(\ \mathit{stype}\ ) \mid \mathsf{super}$
	$\mid \mathsf{LocalOrFieldVar}(\ \mathit{var}\ ) \mid \mathsf{InstVar}(\ \mathit{expr}, \mathit{var}\ ) \mid \underline{\mathsf{InstFun}(\ \mathit{expr}, \mathit{fname}\ )}$
	$\mid \mathit{bexp} \mid \mathit{sexp}$

Fig. 1. The abstract syntax of Java_λ

The language Java_λ (Fig. 1) is an an abstract representation of a core of Java 7. It is an extension of our language in [Plü07]. The additional features are underlined. Beside instance variables functions can be declared in classes. A function

is declared by its name, its type, and the λ–expression. Methods are not considered in this framework. A λ–expression consists of an optionally typed variable and either an statement or an expression. Furthermore the statement expressions respectively the expressions are extended by evaluation-expressions, the λ–expressions, and instances of functions.
The concrete syntax in this paper is oriented at the syntax of [lam10].
The optional type annotations [***type***] are the types, which can be inferred by the type inference algorithm.

3 Types and subtyping

As a base for the type inference algorithm we have to make a formal definition of the Java 7 types. First we give again the definition of simple types (first-order types). The definition is connected to the corresponding definitions in [GJSB05], Section 4.5. and [Plü07], Section 2.

Definition 1 (Simple types). *Let $BTV^{(ty)}$ be the set of bounded type variables and TC a $(BTV)^*$–indexed set of type constructors (class names). Then, the set of* simple types $\mathsf{SType}_{TS}(\,BTV\,)$ *for the given type signature* $(\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *is defined as the smallest set satisfying the following conditions:*

- *For each type ty:* $\underline{BTV^{(ty)}} \subseteq \mathsf{SType}_{TS}(\,BTV\,)$
- $\underline{TC^{()}} \subseteq \mathsf{SType}_{TS}(\,BTV\,)$
- *For* $ty_i \in \mathsf{SType}_{TS}(\,BTV\,)$
  $\cup \{\, \texttt{?} \,\}$
  $\cup \{\, \texttt{? extends}\ \tau \mid \tau \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$
  $\cup \{\, \texttt{? super}\ \tau \mid \tau \in \mathsf{SType}_{TS}(\,BTV\,)\,\}$
  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. 3).*
- *The* <u>*set of implicit type variables with lower or upper bounds*</u> *belongs to* $\mathsf{SType}_{TS}(\,BTV\,)$

Simple types are the first-order base-types of the Java_λ type system. Simple types are used in Java_λ explictly in the extension relation, which defines the subtyping ordering.
The set of simple types is extended to the set of Java_λ types by adding function types.

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

Definition 2 (Types). *Let* $\mathsf{SType}_{TS}(\,BTV\,)$ *be a set of simpletypes. The set of* Java_λ *types* $\mathsf{Type}_{TS}(\,BTV\,)$ *is defined by*

- $\mathsf{SType}_{TS}(\,BTV\,) \subseteq \mathsf{Type}_{TS}(\,BTV\,)$
- *For* $ty, ty_i \in \mathsf{Type}_{TS}(\,BTV\,)$

$$\#\, ty\, (ty_1, \ldots, ty_n) \in \mathsf{Type}_{TS}(\,BTV\,)^2$$

Analogously, the definition of the subtyping relation on simple types is extended to the subtyping relation on Java_λ types.

Definition 3 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(\,BTV\,)$).
Let $TS = (\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *be a type signature of a given* Java *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* $\sigma_1,\ \sigma_2 : BTV \rightarrow \mathsf{SType}_{TS}(\,BTV\,)$*, 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. [GJSB05] §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* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.
- $\theta \leq^* {}_{\theta}|T$

Definition 4 (Subtyping relation $\leq^*$ on $\mathsf{Type}_{TS}(\,BTV\,)$). *Let* $\leq^*$ *be a subtyping relation on simple types* $\mathsf{SType}_{TS}(\,BTV\,)$*. Then, the continuation on* $\mathsf{Type}_{TS}(\,BTV\,)$ *is defined as:*

$$\#\,\theta\,(\theta'_1, \ldots, \theta'_n) \leq^* \#\,\theta'\,(\theta_1, \ldots, \theta_n) \quad \textit{iff} \quad \theta \leq^* \theta' \ \textit{and}\ \theta_i \leq^* \theta'_i.$$

Example 1. Let the following Java_λ program be given.

```
class Matrix extends Vector<Vector<Integer>> {

  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(Matrix.this, m)))
```

[2] Often function types $\#\, ty\, (ty_1, \ldots, ty_n)$ are written as $(ty_1, \ldots, ty_n) \rightarrow ty$.

```
  #Matrix(Matrix, Matrix)
  mul = #(Matrix m1, Matrix m2)
    (Matrix ret = new Matrix ();
      for(int i = 0; i < size(); i++) {
          Vector<? extends Integer> v1 = m1.elementAt(i);
          Vector<Integer> v2 = new Vector<Integer> ();
          for (int j = 0; j < v1.size(); j++) {
            int erg = 0;
            for (int k = 0; k < v1.size(); k++) {
                erg = erg + v1.elementAt(k)
                    * (m2.elementAt(k)).elementAt(j);
            }
            v2.addElement(erg);
          }
        ret.addElement(v2);
      }
    return ret;)

  public static void main(String[] args) {
    Matrix m1 = new Matrix(...);
    Matrix m2 = new Matrix(...);
    m1.op.(m2).(m1.mul);}
}
```

`op` is a curried function with two arguments. The first one is a matrix and the second one is a function which takes two matrices and results another matrix. The function `op` applies its second argument to its own object and its first argument.
`mul` is the ordinary matrix multiplication in closure representation.
Finally, in `main` the function `op` of the matrix `m1` is applied to the matrix `m2` and the function `mul` of `m1`.
From $\texttt{Matrix} \leq^* \texttt{Vector<Vector<Integer>>}$ follows

$$\#\,\texttt{Matrix}\,(\texttt{Vector<Vector<Integer>>}, \texttt{Vector<Vector<Integer>>}) \\ \leq^* \#\,\texttt{Vector<Vector<Integer>>}\,(\texttt{Matrix}, \texttt{Matrix}).$$

4 Type inference rules

In this paper we consider only type inference for function declarations *FunDecl*.
For the type inference system we need some additional definitions:
A set of *type assumptions* O is a map indexed by class names, which maps function names to types (e.g. $O_{\texttt{Matix}} = \{\, \texttt{mul} : \#\,\texttt{Matrix}\,(\texttt{Matrix}, \texttt{Matrix})\,\}$).
In the following σ denotes a *substitution*, which substitutes some (bounded) type variables in a type.
Finally, we need two implications $\triangleright_{Expr}$ and $\triangleright_{Stmt}$. $(O, \tau, \tau') \triangleright_{Expr}$ *exp* $: \theta$ means that under the type assumptions O in the class τ, which direct superclass is τ', the expression *exp* has the type θ.

$(O, \tau, \tau') \triangleright_{Stmt}\ stmt : \theta$ means that under the type assumptions O in the class τ, which direct superclass is τ' the statement *stmt* has the type θ.
First, we consider the type inference rules for λ–expressions. As the body of the λ–expressions either can be a statement or an expression, two rules are necessary.

$$[\mathbf{lambda}_{stmt}]\ \frac{(O \cup \{\, x_i : \theta_i \,\}, \#\,\theta\,(\theta_1, \ldots, \theta_n), \texttt{Object})\ \triangleright_{Stmt}\ s : \theta}{(O, \tau, \tau')\ \triangleright_{Expr}\ \textsf{Lambda}(\,(x_1, \ldots, x_n), s\,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

$$[\mathbf{lambda}_{expr}]\ \frac{(O \cup \{\, x_i : \theta_i \,\}, \#\,\theta\,(\theta_1, \ldots, \theta_n), \texttt{Object})\ \triangleright_{Expr}\ e : \theta}{(O, \tau, \tau')\ \triangleright_{Expr}\ \textsf{Lambda}(\,(x_1, \ldots, x_n), e\,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

Fig. 2. λ–expression rules

As λ–expressions are implemented as inner classes, the type of the class of the λ–expression is given as the type of the λ–expression itself.

For all statements of $\textsf{Java}_\lambda$ there is a type inference rule. We give the most important rules in Fig. 3.

$$[\mathbf{Return}]\ \frac{(O, \tau, \tau')\ \triangleright_{Expr}\ e : \theta}{(O, \tau, \tau')\ \triangleright_{Stmt}\ \textsf{Return}(\,e\,) : \theta}$$

$$[\mathbf{BlockInit}]\ \frac{(O, \tau, \tau')\ \triangleright_{Stmt}\ stmt : \theta}{(O, \tau, \tau')\ \triangleright_{Stmt}\ \textsf{Block}(\,stmt\,) : \theta}$$

$$[\mathbf{Block}]\ \frac{\begin{array}{c}(O, \tau, \tau')\ \triangleright_{Stmt}\ s_1 : \theta,\ \ (O, \tau, \tau')\ \triangleright_{Stmt}\ \textsf{Block}(\,s_2; \ldots; s_n;\,) : \theta' \\ \overline{\theta} \in \mathbf{MUB}(\theta, \theta')\end{array}}{(O, \tau, \tau')\ \triangleright_{Stmt}\ \textsf{Block}(\,s_1; s_2; \ldots; s_n;\,) : \overline{\theta}}$$

$$[\mathbf{If}]\ \frac{\begin{array}{c}(O, \tau, \tau')\ \triangleright_{Stmt}\ \textsf{Block}(\,B_1\,) : \theta, (O, \tau, \tau')\ \triangleright_{Stmt}\ \textsf{Block}(\,B_2\,) : \theta' \\ (O, \tau, \tau')\ \triangleright_{Expr}\ e : \texttt{boolean},\ \ \overline{\theta} \in \mathbf{MUB}(\,\theta_1, \theta_2\,)\end{array}}{(O, \tau, \tau')\ \triangleright_{Stmt}\ \textsf{If}(\,e, \textsf{Block}(\,B_1\,), \textsf{Block}(\,B_2\,)\,) : \overline{\theta}}$$

Fig. 3. Statement Rules

The type of a block of statements is basically defined by the type of the expression of the finishing Return-statement. The **MUB**-function determines the types of minimal upper bounds in the subtyping ordering.

The not presented statements Assign, New, and Eval have the type `void`, as no result is returned.

For all expressions of Java_λ there is a type inference rule. In addition to Fig. 2 we give the most important rules in Fig. 4. In different rules σ is a substitution, which substitutes (bounded) type variables by type-correct expressions.

$$[\textbf{Assign}]\quad \frac{(O,\tau,\tau') \rhd_{Expr}\ e_1 : \theta', (O,\tau,\tau') \rhd_{Expr}\ e_2 : \theta}{(O,\tau,\tau') \rhd_{Expr}\ \mathsf{Assign}(\, e_1, e_2\,) : \theta'}\ \theta \leq^* \theta'$$

$$[\textbf{This}]\quad \frac{}{(O,\tau,\tau') \rhd_{Expr}\ \mathsf{this} : \sigma(\,\tau\,)}$$

$$[\textbf{This_enc}]\quad \frac{}{(O,\tau,\tau') \rhd_{Expr}\ \mathsf{This}(\,\theta\,) : \sigma(\,\theta\,)}$$

$$[\textbf{Super}]\quad \frac{}{(O,\tau,\tau') \rhd_{Expr}\ \mathsf{super} : \sigma(\,\tau'\,)}$$

$$[\textbf{InstVar}]\quad \frac{(O,\tau,\tau') \rhd_{Expr}\ re : \sigma(\,\overline{\theta}\,), \qquad (v : \theta) \in O_{\overline{\theta}}}{(O,\tau,\tau') \rhd_{Expr}\ \mathsf{InstVar}(\, re, v\,) : (\sigma' \circ \sigma)(\,\theta\,)}$$

$$[\textbf{InstFun}]\quad \frac{(O,\tau,\tau') \rhd_{Expr}\ re : \sigma(\,\overline{\theta}\,), \qquad (f : \#\,\theta\,(\theta_1, \ldots, \theta_n)) \in O_{\overline{\theta}}}{(O,\tau,\tau') \rhd_{Expr}\ \mathsf{InstFun}(\, re, f\,) : (\sigma' \circ \sigma)(\,\#\,\theta\,(\theta_1, \ldots, \theta_n)\,)}$$

Fig. 4. Expression rules

The **Assign**-rule is canonically defined.
The **This**–rule types the expression this by its class τ. As each λ–expression is implemented as an anonymous inner class, τ is the type of the most nested λ–expression. In contrast the **This_enc**–rule types the expression this of an enclosing class θ.
The rules InstVar and InstFun types identifiers which are defined in a class $\overline{\theta}$ as fields and functions, respectively.

5 Type inference algorithm

In the late eighties Fuh and Mishra [FM88] gave an algorithm for type inference in the λ-calculus with subtyping. Their calculus indeed allows subtyping but no additional overloading. This means that the type system of Java_λ and the

type system of their language are equivalent. There is only one small difference: Their subtyping relations are finite, while in Java_λ infinite chains are possible in subtyping relations. We considered this problem in [Plü09]. The result is that there is finite number of representants for the infinite chains. We will use this result in the type inference algorithm.

5.1 Summary of Fuh and Mishra's algorithm

The algorithm is called WTYPE, which stands for *well-typing*. A *well-typing* is data-structure $C, A \vdash N : t$, where C is a set of consistence coercions (unsolved unequations), A is a set of type assumptions, N is an expression, and t is the derived type. The algorithm has the following signature:

$$\mathbf{WTYPE} : \texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{WellTyping} + \{ \mathit{fail} \}$$

This means the result of the algorithm is a type with a set of constraints. **WTYPE** consists of four functions.

TYPE : $\texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{Type} \times \texttt{CoercionSet}$ maps a fresh type variable to each subterm of the input expression and determines coercions, which contain the function and the tuple constructors derived from the structure of the λ–expression.

MATCH : $\texttt{CoercionSet} \rightarrow \texttt{Substitution} + \{ \mathit{fail} \}$ determines a substitution σ. σ applied to the input coercion set C results in a minimal matching instance of C. The algorithm **Match** is an adoption to coercions of the Martelli, Montanari unification algorithm [MM82], which is used in the Damas Milner type inference algorithm [DM82].

SIMPLIFY : $\texttt{CoercionSet} \rightarrow \texttt{AtomicCoercionSet}$ eliminates the type constructors, especially the *function* and the *tuple-constructor*.

CONSISTENT : $\texttt{AtomicCoercionSet} \rightarrow \texttt{Boolean} + \{ \mathit{fail} \}$ checks if a set of atomic coercions is consistent, by determining all possible instances of each variable. If finally for each variable there is a non-empty set of instances, the set of atomic coercions is consistent.

This leads to the following algorithm:

$$\begin{array}{ll}
\mathbf{WTYPE}(\,A, e\,) = & \mathbf{let}\ (\theta, C) = \mathbf{TYPE}(\,A, e\,)\ \mathbf{in} \\
& \mathbf{let}\ \sigma = \mathbf{MATCH}(\,C\,)\ \mathbf{in} \\
& \mathbf{let}\ C' = \mathbf{SIMPLIFY}(\,\sigma(\,C\,)\,)\ \mathbf{in} \\
& \mathbf{if}\ \mathbf{CONSISTENT}(\,C'\,)\ \mathbf{then} \\
& \quad (C', A) \vdash e : \sigma(\,\theta\,) \\
& \mathbf{else} \\
& \quad \mathit{fail}
\end{array}$$

5.2 Adaption to the Java_λ type system

Now we give a sketch how to adapt the type inference algorithm to Java_λ λ–expressions.
The functions **SIMPLIFY** and **CONSISTENT** have to be changed.

SIMPLIFY: The functions of the unification algorithm of Martelli and Montanari [MM82] are substituted by the corresponding functions of the Java type unification algorithm [Plü09].

CONSISTENCE: In the functions ↑ and ↓, which determine iteratively all supertypes and all subtypes, respectively, of a set of types, the functions `greater` and `smaller` from [Plü09] are used, such that only a finite set of representants is determined.

We give an example, how to use the adopted algorithm.
Let the program from 1 be given again, where the function `op` is not explictly typed:

```
op = #(m)(#(f)(f(Matrix.this, m)))
```

TYPE$(\emptyset,$ `#(m)(#(f)(f(Matrix.this, m)))` $) = (t_{\texttt{op}}, C)$, where
$C = \{ (t_m \to t_{\#f} \lessdot t_{\texttt{op}}), (t_f \to t_{f(\,M.this,m\,)} \lessdot t_{\#f}), (t_f \lessdot (t_1, t_2) \to t_3),$
$(\texttt{Matrix} \lessdot t_1), (t_m \lessdot t_2), (t_3 \lessdot t_{f(\,M.this,m\,)} \})$

MATCH$(\,C\,)$**:**

- From $(t_m \to t_{\#f} \lessdot t_{\texttt{op}})$ follows

$$\sigma_1 = \{ (t_{\texttt{op}} \mapsto \beta \to \beta') \}.$$

- From $(t_f \to t_{f(\,M.this,m\,)} \lessdot t_{\#f})$ follows

$$\sigma_2 = \sigma_1 \cup \{ (t_{\#f} \mapsto \gamma_1 \to \gamma_1') \}.$$

- From $(t_f \lessdot (t_1, t_2) \to t_3)$ follows

$$\sigma_3 = \sigma_2 \,\cup\, \{ (t_f \mapsto (\epsilon_1, \epsilon_1') \to \epsilon_1'') \}.$$

- From $\sigma_3(\,t_f \to t_{f(\,M.this,m\,)} \lessdot t_{\#f}\,) =$
$(((\epsilon_1, \epsilon_1') \to \epsilon_1'') \to t_{f(\,M.this,m\,)} \lessdot \gamma_1 \to \gamma_1')$ follows

$$\sigma_4 = \sigma_3 \,\cup\, \{ (\gamma_1 \mapsto (\epsilon_2, \epsilon_2') \to \epsilon_2'') \}.$$

- From $\sigma_4(\,t_m \to t_{\#f} \lessdot t_{\texttt{op}}\,) = \ t_m \to ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1' \lessdot \beta \to \beta'$ follows

$$\sigma_5 = \sigma_4 \cup \{ \beta' \mapsto ((\epsilon_3, \epsilon_3') \to \epsilon_3''') \to \gamma_2' \}.$$

Result: $\sigma = \{ (t_{\texttt{op}} \mapsto \beta \to ((\epsilon_3, \epsilon_3') \to \epsilon_3'') \to \gamma_2'),$
$(t_{\#f} \mapsto ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1'), (t_f \mapsto (\epsilon_1, \epsilon_1') \to \epsilon_1'') \}$

It holds: $\sigma(\,C\,) = \{ \quad t_m \to ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1' \lessdot \beta \to ((\epsilon_3, \epsilon_3') \to \epsilon_3'') \to \gamma_2',$
$((\epsilon_1, \epsilon_1') \to \epsilon_1'') \to t_{f(\,this,m\,)} \lessdot ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1',$
$(\epsilon_1, \epsilon_1') \to \epsilon_1'' \lessdot (t_1, t_2) \to t_3,$
$\texttt{Matrix} \lessdot t_1,$
$t_m \lessdot t_2,$
$t_3 \lessdot t_{f(\,this,m\,)} \}$

SIMPLIFY$(\sigma(C)) = C'$, where

$$C' = \{ \beta \lessdot t_m, \epsilon_2 \lessdot \epsilon_3, \epsilon'_2 \lessdot \epsilon'_3, \epsilon''_3 \lessdot \epsilon''_2, \gamma'_1 \lessdot \gamma'_2 \\ \epsilon_1 \lessdot \epsilon_2, \epsilon'_1 \lessdot \epsilon'_2, \epsilon''_2 \lessdot \epsilon''_1, t_{f(this,m)} \lessdot \gamma'_1 \\ t_1 \lessdot \epsilon_1, t_2 \lessdot \epsilon'_1, \epsilon''_1 \lessdot t_3 \\ \texttt{Matrix} \lessdot t_1 \\ t_m \lessdot t_2 \\ t_3 \lessdot t_{f(this,m)} \}$$

CONSISTENT(C')**:** For each $\theta \lessdot \theta' \in C'$ we have to determine I_θ respectively $I_{\theta'}$. If all $I_\theta \neq \emptyset$ the atomic coercion set is consistent.

In the following table we consider the iteration steps[3].

It	*Coercion*	$I_{\texttt{Matrix}}$	I_{t_1}	I_{ϵ_1}	I_{ϵ_2}	I_{ϵ_3}	...
0		M	*	*	*	*	*
1	$\texttt{M} \lessdot t_1$	M	M, V<V<Int>>	*	*	*	*
1	$t_1 \lessdot \epsilon_1$	M	M, V<V<Int>>	M, V<V<Int>>	*	*	*
1	$\epsilon_1 \lessdot \epsilon_2$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	*	*
1	$\epsilon_2 \lessdot \epsilon_3$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	*
1	...	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	*
2	...	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	*

This means **CONSISTENT**$(\sigma(C')) = \texttt{true}$.

From this follows **WTYPE**$(\emptyset,$ `#(m)(#(f)(f(Matrix.this, m)))` $) =$

$$(C', \emptyset) \vdash \texttt{\#(m)(\#(f)(f(Matrix.this, m)))} : \beta \rightarrow ((\epsilon_3, \epsilon'_3) \rightarrow \epsilon''_3) \rightarrow \gamma'_2.$$

As in the function CONSISTENCE all possible instances are determined, it holds:

$$\epsilon_3 = \texttt{Matrix} \text{ or } \texttt{Vector<Vector<Integer>>}.$$

Furthermore from C' follows, that it holds:

- $\beta \lessdot t_m \lessdot t_2 \lessdot \epsilon'_1 \lessdot \epsilon'_2 \lessdot \epsilon'_3$
- $\epsilon''_3 \lessdot \epsilon''_2 \lessdot \epsilon''_1 \lessdot t_3 \lessdot t_{f(this,m)} \lessdot \gamma'_1 \lessdot \gamma'_2$,

which describe corelations of type variables of the result type.

6 Conclusion and future work

We have considered the Java 7 extensions closures and function types as first-class citizens. We gave an abstract definition of the subtyping relation and define the type inference rules for a small core language Java_λ. The properties of the Java_λ type system are very similar to the type system, which is considered by

[3] We consider only the non-wildcard types.

Fuh and Mishra [FM88]. Therefore, we gave a sketch of the adoption of their type inference algorithm to Java_λ.
The result of Fuh and Mishra's algorithm are well-typings $C, A \vdash N : t$, where C is a set of consistence coercions (unsolved unequations), A is a set of type assumptions, N is an expression, and t is the derived type. For an implementation of the type inference algorithm in a Java IDE the realization of well-typings is an open problem, as the Java type systems indeed allows bounded type variables, but no constraints on type variables.
Furthermore we have to prove correctness and soundness of the Java_λ adopted type inference algorithm.
Finally, after solving the problem of well-typing realization, we have to implement the algorithm.

References

[BGGvdA] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Closures for the java programming language (aka BGGA). `http://www.javac.info`.

[CS] Stephen Colebourne and Stefan Schulz. First-class methods: Java-style closures (aka FCM). `http://docs.google.com/Doc?id=ddhp95vd_6hg3qhc`.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Proceedings 2nd European Symposium on Programming (ESOP '88)*, pages 94–114, 1988.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.

[lam10] Project lambda: Java language specification draft. `http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt`, 2010. Version 0.1.5.

[LLB] Bob Lee, Doug Lea, and Josh Bloch. Concise instance creation expressions: Closures without complexity (aka CICE). `http://docs.google.com/Doc.aspx?id=k73_1ggr36h`.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Wrzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

E-Assessment kreativer Prüfungsleistungen

Herbert Kuchen

Universität Münster

E-Assessment-Systeme ermöglichen die voll- oder halb-automatische Korrektur von Prüfungsleistungen sowohl in Übungen als auch in Klausuren. Sie ermöglichen eine drastische Senkung des Korrekturaufwands und der Korrekturzeit. Außerdem garantieren Sie eine objektive und einheitliche Bewertung. Heutige E-Assessment-Systeme unterstützen typischerweise nur simple Aufgabentypen wie Multiple-Choice oder Kurztext. Das an der Uni Münster entwickelte System EASy geht hierüber hinaus, indem es beispielsweise die Korrektur von mathematischen Beweisen und Programmieraufgaben unterstützt. Die Handhabung mathematischer Beweise basiert intern auf Termersetzuung. Bei der Korrektur von Programmieraufgaben werden aus der Musterlösung systematisch und automatisch erzeugte Glass-Box-Testfälle verwendet. EASy ist modular aufgebaut und erlaubt über eine Schnittstelle den Ausbau um weitere Aufgabentypen. Das System wurde in Lehrveranstaltungen mit bis zu 250 Studierenden evaluiert.

A new language for algebraic dynamic programming

Georg Sauthoff, Robert Giegerich

Faculty of Technology, Bielefeld University

Algebraic Dynamic Programming

Algebraic Dynamic Programming (ADP) is a declarative style of dynamic programming, which has emerged in the area of biosequence analysis. Based on the concepts of signatures, algebras, and regular tree grammars, a dynamic programming algorithm can be expressed at a convenient level of abstraction, obviating the development and the tedious debugging of the matrix recurrences typical of dynamic programming. The perfect separation of search space composition and evaluation of solution candidates, as enforced in ADP, leads to unprecedenced versatility in the combination of different analyses via product algebras. The ADP method has been used in implementing a good number of bioinformatics tools in the area of RNA structure prediction.

Historically, the implementation of ADP was prototyped in the lazy functional language Haskell, but efficiency as well as proliferation concerns require a stand-alone implementation.

Bellman's GAP and its compiler

Bellman's GAP is a new domain specific language for writing programs in the ADP paradigm. Bellman's GAP contains C/Java-like syntax elements and a notation for tree grammars resembling function calls. Compiling the declarative source code essentially means the derivation of and code generation for efficient dynamic programming recurrences. The current compiler implements non-trivial semantic analyses for yield size analysis and table design. Further examples are the automatic generation of different backtracing schemes or the generation of OpenMP-parallelized code.

Overview of the presentation

The talk will give a short introduction to the declarative concepts of algebraic dynamic programming. We will examplify the use of the new language with simple textbook style examples. We then report on some optimization by the compiler which lead to implementations competitive with handwritten code.

An experiment with the fourth Futamura projection

Robert Glück

Dept. of Computer Science, University of Copenhagen, Denmark

We have experimentally validated the theoretical insight, that a compiler generator is a generating extension of a program specializer, by showing that an existing offline partial evaluator can perform the fourth Futamura projection. Specifically, an online and an offline partial evaluator for an imperative flowchart language were transformed into two new compiler generators by Romanenko's classical partial evaluator Unmix. The two partial evaluators are described, as is a novel recursive method for polyvariant specialization. These results strongly indicate that existing partial evaluation techniques can be put to work on several new applications.

Reference

1. Glück R., An experiment with the fourth Futamura projection. In: Pnueli A. at al. (eds.), Perspectives of System Informatics. Lecture Notes in Computer Science, volume 5947, pages 135-150, Springer-Verlag 2010. `http://dx.doi.org/10.1007/978-3-642-11486-1_12`

Signale statt Generatoren!*

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Lehrstuhl Programmiersprachen und Compilerbau
Postfach 101344, 03013 Cottbus

Zusammenfassung Funktionale Reaktive Programmierung (FRP) benutzt Signale, um zeitliches Verhalten zu beschreiben. Ereignisbasierte FRP-Implementierungen vermeiden Neuberechnungen von Signalwerten in bestimmten Situationen, indem sie Datenabhängigkeiten berücksichtigen. Jedoch unterstützen diese Implementierungen typischerweise Signale nicht direkt. Stattdessen werden Signale von Signalgeneratoren erzeugt. Das mehrfache Verwenden desselben Generators führt dabei zu unnötigen Mehrfachberechnungen sowie zu Abhängigkeit der Signale von ihrer Erzeugungszeit. Das Ergebnis ist geringe Skalierbarkeit sowie eine komplizierte Semantik.
In diesem Vortrag wird eine ereignisbasierte FRP-Implementierung vorgestellt, welche die beschriebenen Probleme nicht hat. In dieser Implementierung spielen verzögerte Auswertung, Polymorphie zweiter Stufe und impredikative Polymorphie eine essentielle Rolle. Man erhält ein skalierbares FRP-System, welches dem Nutzer direkten Zugang zu Signalen als dem Schlüsselkonzept der FRP verschafft.

Literatur

1. Jeltsch, W.: Signals, not generators! In Horváth, Z., Zsók, V., Achten, P., Koopman, P., eds.: Trends in Functional Programming. Volume 10. Intellect, Bristol, UK (2010)

* Die Ideen dieser Arbeit entstammen dem (bis auf Sprache) gleichnamigen Artikel des Autors [1].

Twilight STM in Haskell

Annette Bieniusa[1], Arie Middelkoop[2], Peter Thiemann[1]

[1] Institut für Informatik, Arbeitsbereich Programmiersprachen, Universität Freiburg
{bieniusa,thiemann}@informatik.uni-freiburg.de
[2] Department of Information and Computing Sciences, Universiteit Utrecht
amiddelk@cs.uu.nl

Transactional Memory (TM) hat sich zu einer vielversprechende Alternative bei multi-threaded Anwendungen entwickelt. Im Vergleich zu traditionellen locking-basierten Synchronisationsverfahren verbessert es die Gesamtproduktivität des Programmierers, indem es feingranular Daten vor nebenläufigem Zugriff sichert und zugleich Modularität, Skalierbarkeit und Wiederverwendung von Code in anderem Kontext bietet. Erste industriereife Implementierungen sind mittlerweile verfügbar, die sich kompetitiv zu anderen Synchronisationsparadigmen verhalten.

Programmierer, die Transactional Memory in ihren Anwendungen einsetzen wollen, stehen jedoch weiterhin vor einem Problem: Die Verwendung von bestehenden APIs ist häufig nicht möglich, da diese in der Regel auf Protokollen mit Handshake oder Locking basieren, um I/O und andere Systemservices bzw. nebenläufige Datenstrukturen zu nutzen. Dies widerum steht im Konflikt zur Nutzung von TM Operationen.

Twilight STM ist eine Erweiterung des STM Programmiermodells. Es unterteilt Transaktionen in zwei Phasen: eine Transaktionsphase und eine *Twilight-Phase*, welche durch eine *twilight* Operation voneinander getrennt sind. Zu Begin der Twilight-Phase werden zunächst die transaktionalen Speicheroperationen auf Konsistenz geprüft, bevor die in der Transaktionsphase geschriebenen Werte für andere Transaktionen zum Lesen und Schreiben freigegeben werden. Im Vergleich zu anderen Zweiphasen-Commitprotokollen kann der Programmierer anschliessend flexibel und in Abhängigkeit der jeweiligen Anwendung auf Konflikte reagieren. Durch Korrektur der zu schreibenden Werte kann dann beispielsweise auf ein Rollback verzichtet werden. Eine erweiterte API unterstützt die Inspektionen von Konflikten und deren Korrektur bzw. Reparatur.

Darüberhinaus können in der Twilight Zone I/O Operationen ausgeführt werden. Diese externe Operationen sind direkt global sichtbar und erlauben es auch, dass Transaktionen miteinander direkt kommunizieren.

Der Vortrag stellt eine Implementierung von Twilight für Haskell vor. Durch den Einsatz von parametrisierten Monaden stellt das Typsystem eine korrekte Anwendung der API Operationen sicher. Ein Mechanismus zum Taggen erlaubt das Gruppieren von transaktionalen Variablen und erleichtert das Auffinden und Auflösen von Konflikten.

Wir zeigen ausserdem eine Formalisierung des Systems basierend auf einem einfach getypten Lambda-Kalkül, das um Monaden und Threads erweitert wurde. Das formalisierte System Λ_{Twi} dient als Grundlage zum Beweis diverser semantischer Eigenschaften von Haskell Twilight, wie beispielsweise der Serialisierbarkeit von Transaktionen oder auch der Integrität transaktional verwalteter Daten.

Tofu – Towards Practical Church-Style Total Functional Programming

Baltasar Trancón y Widemann

Universität Bayreuth

Abstract. According to Turner, the mathematical notion of a function is not represented accurately by conventional functional programming. The problem is caused by partial functions and nonterminating reductions. The semantic difference between mathematical and programmed functions has particular impact on formal methods of software engineering, where one document may be read both as a mathematical model and as an executable prototype.

Accurate representations of mathematical total functions are available in the form of type-theoretic calculi. However, such a calculus has to be enhanced substantially with syntactic and/or semantic sugar before it is usable as a programming language. In particular, typical features of functional languages, such as recursive defining equations of functions and types, type inference or type classes, are missing.

The prototypic language TOFU is based on Coquand's calculus of constructions and aims at providing a useful programming environment through enhanced syntax alone, with uncompromising semantic purism. We present selected features of Tofu, demonstrate how they interact synergetically and discuss their potential as replacements of the aforementioned functional programming constructs.

1 Introduction

1.1 Total Functional Programming

Throughout the history of functional programming, theoreticians have taken the full power of Turing-complete languages as granted and devised mathematical models for the corresponding universe of partial functions. From a pragmatic viewpoint, partial functions are a useful extension of the total functions of traditional mathematics. From a theoretical viewpoint however, there is a significant cost to this extension, concisely summarized by Turner:

> *The driving idea of functional programming is to make* ***programming*** *more closely related to* ***mathematics****. A program in a functional language* [...] *consists of equations which are both* ***computation*** *rules and a basis for* ***simple*** *algebraic* ***reasoning*** [...]*. The existing model of functional programming, although elegant and powerful, is* ***compromised*** *to a greater extent than is commonly recognised by the presence of* ***partial*** *functions.* [20, emphasis added]

Beside the semantic calculi defined in terms of partial functions, there are also a variety of calculi of total functions, usually based on some kind of constructive set theory and therefore associated with strong static typing. These total calculi include Church's *simply typed lambda calculus*, Reynolds's *System F* and Coquand's *calculus of constructions* (CC) [4] (all vertices of Barendregt's *lambda cube* [1]) as well as Martin-Löf's *type theory*. By virtue of the Curry–Howard correspondence, calculi of total functions are also powerful proof systems of intuitionistic logic, hence numerous mechanic theorem provers are based on extensions of the latter two calculi.

Turner acknowldeges the theoretical merit of such calculi, but discards them as impractical for didactic reasons:

> [Martin-Löf's type] *theory was developed as a foundational language for constructive mathematics. It is certainly possible to program in it* [...], *but it would hardly be suitable as an introductory language to teach programming.* [20]

The total functional programming language Tofu has been designed for application areas where teaching is not an issue, but the unity of computation of functions and simple reasoning about their properties is of utmost importance. We propose that prime examples of such applications can be found in mathematical approaches to software engineering.

1.2 Applications in Software Engineering

The functions and relations that arise in the mathematical description of software systems frequently decompose into a large number of distinct cases. For safety-critical software, organizing and inspecting those cases in a systematic way is of crucial importance. For this task, tabular expressions have been proposed and popularized by Parnas and others [14, 13]. A collection of table types [11] with fixed structure and semantics have become known as the "Parnas tables". Software tools for the computer-aided management and evaluation of tabular expressions of these types have been developed [17, 15].

More recently, a generalized table model that subsumes both the standard table types and a wide variety of user-definable custom types is emerging. In this model, a tabular expression is essentially a pair of a data structure containing atomic subexpressions with free variables and an evaluation algorithm that gives its semantics as a function of the free variables. A formulation of this model in terms of functional programming has been proposed in [19]. This formulation realizes the unity of computation and logic by

1. a canonical method of obtaining an executable function from a tabular expression by specializing the evaluation algorithm with the expression content (partial evaluation), and
2. reasoning about the properties of that function with the usual algebraic techniques of program transformations and equivalence proofs for purely functional programs.

The logic of tabular expressions as defined in literature is a first-order logical language with partially defined value expressions, but totally defined propositions [12]. In other words, a predicate applied to an undefined expression (arising from a partial function applied outside its domain) is not undetermined but false.[1] Consequently, tabular expression can not be evaluated effectively in a calculus of partial functions, where undefinedness subsumes nontermination. [19] proposed a semantically sound implementation in a total functional programming environment that has since evolved into Tofu.

2 Church-Style and Curry-Style Typing

Tofu is rather unconventional for a functional language with respect to the style of its type system following the tradition of Church rather than the tradition of Curry. The differences between these two schools have been discussed hotly in type system theory and practice. See [1, 16] for thorough theoretical treatments, or [18] for a short paper on related issues in contemporary language design.

The most obvious distinction is the question whether the types of variables are declared upon binding (Church) or inferred from the context (Curry). [16] refers to this distinction as *intrinsic* vs. *extrinsic* typing. Outside the functional programming community, statically typed languages traditionally favour the former, whereas for dynamically typed languages only the latter makes sense.

This seemingly trivial details has wide ramifications, with the Church-style having numerous theoretical advantages:

Expression Types Church-style types can be computed strictly bottom-up, whereas Curry-style types must be computed bidirectionally using unification algorithms, Milner's W being the most prominent.

Type Signatures Type signatures for top-level expressions are redundant in Church-style systems, whereas many advanced features of Curry-style systems may fail if a guiding type signature is missing.

Power Church-style systems can incorporate very avanced and powerful features without incurring decidability problems, whereas the Curry-style version of System F is already undecidable [1].

Polymorphism Church style has explicit binding of type variables and supports higher-rank polymorphism out of the box, whereas Curry style defaults to implicit universal quantification of type variables. Curry-style higher-rank polymorphic types are available as a Haskell language extension, but interfere severly with type inference [21].

On the practical side, Curry-style systems have the huge advantage that they relieve the programmer from the task of declaring a substantial amount of (often apparently redundant) type information. We propose that the key to the design of a practical Church-style language is a technique or rather a collection of techniques to reduce the "formal noise" incurred by Church-style type declarations.

[1] A similar logic is part of the IEEE 754 standard on floating-point arithmetics.

TOFU is a faithful and pure implementation of CC. As such, it has a Church-style type system with only one type constructor, the *dependent product operator* $\prod$, and a single constant $*$ for the kind of types. The dependent product may be thought of as a space of *choice functions*: Informally, the type $\prod x : A.\ B_x$ contains all A-indexed families (b_a) such that $b_a : B_a$ for all $a : A$. This operator doubles as the function space constructor by defining $A \to B = \prod x : A. B$ where x is not free in B and, by Curry–Howard correspondence, as the universal quantifier of intuitionistic higher-order logic. The remainder of this paper describes techniques designed to make TOFU a practical Church-style total functional programming language in the above sense.

3 Associating Variables and Types

3.1 Variable Declarations

The first technique for reducing the type-related burden on the programmer is simple, but rather effective: reuseable variable declarations. The rationale is that well-written programs, and mathematical documents for that matter, use variables consistently in fixed roles throughout, as recommended by Knuth:

> *14. Don't use the same notation for two different things. Conversely, use consistent notation for the same thing when it appears in several places. For example, don't say "A_j for $1 \leq j \leq n$" in one place and "A_k for $1 \leq k \leq n$" in another place unless there is a good reason. It is often useful to choose names for indices so that i varies from 1 to m and j from 1 to n, say, and to stick to consistent usage.* [...] [6, § 1 p. 3]

Such a consistent style can be leveraged by making variable declarations first-class citizens of the language. In TOFU one may declare

$$\textbf{var}\ n : \mathbb{N}$$

and bind the variable n, without mentioning its type, wherever this declaration is in scope. For instance, one may write

$$\textbf{def}\ \mathit{square}(n) = \mathit{mult}(n,\ n) \qquad \text{— as opposed to}\ \ \mathit{square}(n : \mathbb{N})$$

just like in a Curry-style language. Variable declarations obey the same scoping and import rules as constant (function or type) declarations in the TOFU module system. They can even be gathered in a "glossary" module shared by the rest of a larger program. Systematic use of variable declarations not only makes a program very reader-friendly, but covers a surprisingly large fraction of all bound variables.

3.2 Open Declarations and Context

Declaring a variable in advance works well only with *ground* types; the simple mechanism described above does not extend to polymorphic variables whose types contain, or consist of, a type variable. Consider the usage convention that the variable a denotes elements of type A, which itself is a type variable occurring frequently in a library of polymorphic functions. In TOFU one may declare

$$\textbf{var open}\,(A\ :\ *)\ \Rightarrow\ a\ :\ A$$

to state that a may be bound without mentioning a type in a context where A is bound to a type (indicated by the kind constant $*$). The declaration of A in the context constraint on a is an ordinary variable binding, so given

$$\textbf{var}\ A\ :\ *$$

it can be shortened to

$$\textbf{var open}\,(A)\ \Rightarrow\ a\ :\ A$$

When all variables in a context contraint are pre-declared, the context can even be inferred, hence one may write

$$\textbf{var open}\ a\ :\ A$$

to the same effect as above. Now both variables can be used in the declaration or definition of a polymorphic identity function:

$$\begin{array}{l}\textbf{const}\ id\ :\ \prod A.\, A \to A\\ \textbf{def}\ id(A)(a)\ =\ a\end{array}$$

4 Bottom-Up Inference

TOFU has a limited unidirectional type (and value) inference mechanism, based on an idea due to Coquand and Huet [4]. The key observation is that, in a so-called *pure type system* such as CC, instantiations of polymorphic functions and dependent types use the same application construct as ordinary function arguments. The only difference is that the former occur in the *type* of the application result, while the latter occur only in the *value*; cf. the definition of the arrow operator in section 2. If the result is another function that is applied to more arguments, then the first argument might be inferred from the type of the following ones by simple unification.

Consider the identity function as defined above. The first argument A occurs trivially in the type of the second argument a. Hence the first argument can be omitted if the second is given. In TOFU one may write (with the variable declarations given above still in scope)

$\mathbf{def}\ \mathit{id}[A](a) \ = \ a$

with a bracket to indicate that the first argument A may be omitted. The bracket annotation is inherited by the type of an expression, such that the type computed for *id* is $\prod[A].\ A \to A$. Arguments for functions of bracketed product type may be given explicitly, also in brackets, or they may be omitted and one or more unbracketed arguments follow. In the latter case, the type of the following arguments is matched with the body type of the product to infer the omitted information. For example, the expression $\mathit{id}(n)$ omits a bracketed argument that can be either inferred or given explicitly as $\mathit{id}[\mathbb{N}](n)$.

The mechanism works not only for trivial examples such as the identity function, but for more complex and useful definitions as well. The following examples show, without going into the details, inferrable arguments in various advanced situations.

1. Nontrivial polymorphic functions:

   $\mathbf{def}\ \mathit{singleton}[A](a) \ = \ \mathit{cons}(a,\ \mathit{nil}(A))$

2. Type contructors, or generally type-level functions:

   $\mathbf{var}\ F \ : \ * \to *$
   $\mathbf{def}\ \mathit{Functor}(F) \ = \ \prod\,[A,\ B].\ (A \to B) \to F(A) \to F(B)$

3. Higher-order predicate logic with dependent types:

   $\mathbf{var\ open}\ p \ : \ A \to *$
   $\mathbf{def}\ \mathit{exists}[A](p) \ = \ \prod\,[C].\ \big(\prod\ a.\ p(a) \to C\big) \to C$

The missing declaration of the variables $B,\ C$ is left as an exercise to the reader.

Inference by unification of types is possible even though Tofu has full higher-order type-level functions rather than the nominal injective type constructors found in most type systems. The key observation is that the primitive type operator $\prod\ x \ : \ A.\ B$ is injective in both A and B. Tofu has a few additional inference rules, but to describe those is beyond the scope of this paper.

5 Defining (Co)Recursive Data Types and Functions

It has been well-known for a long time how to define common data types such as booleans, natural numbers or lists, in a purely functional calculus. The basic technique is called *Church encoding*, which gives a hint at how ancient it is. In System F and its extensions, including CC, Church encoding can be systematized in a strongly typed way, allowing definitions of advanced constructs such as free data types (initial algebras) and cofree data (codata) types (final coalgebras) of polynomial functors [22].

5.1 Generic Type Definitions

Strongly typed Church encoding can be thought of as the application of a continuation-passing transform, or equivalently Yoneda's Lemma, followed by massage with natural isomorphisms such as Schönfinkel–Curry. For instance, consider the following derivations of encodings of Cartesian product and disjoint union:

$$\begin{aligned} A \times B &= \textstyle\prod C.(A \times B \to C) \to C \\ &= \textstyle\prod C.(A \to B \to C) \to C \end{aligned} \qquad \begin{aligned} A \uplus B &= \textstyle\prod C.(A \uplus B \to C) \to C \\ &= \textstyle\prod C.\big((A \to C) \times (B \to C)\big) \to C \\ &= \textstyle\prod C.(A \to C) \to (B \to C) \to C \end{aligned}$$

For recursive data types, Wadler [22] gives a canonical encoding with initial algebra semantics as the least fixpoint of a functor, in TOFU notation:

$$\mathit{Init}(F) \;=\; \prod\; C.\;\big(F(C) \to C\big) \;\to\; C$$

This type constructor is complemented by a set of three polymorphic interface operations:

$$\begin{aligned} \mathit{fold} &: \textstyle\prod\,[F].\,\mathit{Functor}(F) \to \prod\,[C].\,(F(C) \to C) \to \mathit{Init}(F) \to C \\ \mathit{in} &: \textstyle\prod\,[F].\,\mathit{Functor}(F) \to F\big(\mathit{Init}(F)\big) \to \mathit{Init}(F) \\ \mathit{out} &: \textstyle\prod\,[F].\,\mathit{Functor}(F) \to \mathit{Init}(F) \to F\big(\mathit{Init}(F)\big) \end{aligned}$$

The implementation is left as an exercise to the reader. An encoding with final coalgebra semantics is completely dual.

5.2 Practical Considerations

There are some issues with Church-encoded recursive data types in practice. A recent blog entry judges bluntly: "*Are Church encodings practical? Not really...*" [9]. Wadler argues more subtly:

> *Regarding pragmatics, it is well known that the embedding of least fixpoints is less efficient than one would like. For instance, the operation to find the tail of a list takes time proportional to the length of the list: one would hope that this takes constant time.* [...] *Is there a way of coding lists in polymorphic lambda calculus that (a) uses space proportional to the length of the list, (b) performs cons in constant time, and (c) performs tail in constant time? Or is there a proof that this is impossible? So far as I know, this is an open question.* [22]

We propose to solve Wadler's problem by cheating. Consider the following complete theory of the abstract interface of initial data types, essentially Lambek's Lemma and the universal homomorphism property:

$$\mathit{Init}(F) \cong F\big(\mathit{Init}(F)\big)$$
$$\mathit{in} \circ \mathit{out} = \mathit{id} \qquad \mathit{out} \circ \mathit{in} = \mathit{id}$$
$$\mathit{fold}(f)(k) \circ \mathit{in} = k \circ f\big(\mathit{fold}(f)(k)\big)$$

We are not aware of a strongly typed realization of this theory that satisfies Wadler's constraints either, but we can give an untyped realization that does:

$$\begin{gathered}\big\{ Init(F) = F\big(Init(F)\big)\big\} \\ in = out = id \\ fold(f)(k) = Y\big(\lambda\, g.\, k \circ f(g)\big)\end{gathered}$$

where Y is the usual fixpoint combinator. This results in a variant of Church encoding equivalent to the one proposed in [5]. Since TOFU compiles to Java via an intermediate untyped functional language, the strongly typed version can be replaced safely behind the scenes.

5.3 Recursive Functions

Syntactic recursion is forbidden in TOFU; recursive total functions are defined as homomorphisms from an initial or into a final data type, in the spirit of the *Squiggol* formalism and the infamous banana brackets [8]. These homomorphisms are provided by the function *fold* described in the preceding section and its dual *unfold*.

Since the times of *Charity* [3], one of the first total functional languages, it has become fashionable to define the Ackermann function as an example of total recursion. Unlike the Charity version, which uses coiteration over an infinite table, the TOFU version uses iteration over a function space. Given a data type of natural numbers,

$$\begin{aligned}&\mathbf{def}\ \mathbb{N} \ =\ Init(Peano)\\ &\mathbf{const}\ succ\ :\ \mathbb{N} \to \mathbb{N};\ zero\ :\ \mathbb{N}\\ &\mathbf{var}\ f\ :\ \mathbb{N} \to \mathbb{N}\\ &\mathbf{const}\ iter\ :\ \textstyle\prod\, [A].\ (A \to A) \to A \to \mathbb{N} \to A\end{aligned}$$

where $iter(f,\ e)\big(succ^n(zero)\big)\ =\ f^n(e)$, the Ackermann function is defined as follows:

$$\mathbf{def}\ ack\ =\ iter(\lambda\, f.\ iter(f,\ f(succ(zero))),\ succ)$$

6 Type Classes and Algebraic Programming

The language Haskell has introduced *type classes* as a technique for organizing ad-hoc polymorphism (homonymous but distinct functions operating on distinct types). While much has been written about the relation of type classes to classes in the object-oriented paradigm, we wish to point out a different perspective: Type classes are closely related to algebraic signatures and the instances of a type class to the algebras of a signature, as evidenced by the *dictionary* implementation of type class instances in Haskell compilers. The only difference between type-class programming and universal algebra is the peculiar constraint

that *the carrier of an instance must determine the operations uniquely.* This severe restriction leads to a number of peculiar style issues, particularly in connection with signatures that have multiple algebras such as *Monoid*, and hence need to be disambiguated on the type level.

TOFU already has the apparatus to define signatures and algebras for the purpose of user-defined data types. The recommended style for ad-hoc polymorphism is therefore *algebraic programming*, where ad-hoc polymorphic functions abstract explicitly from the algebras they employ.

For cases where there is arguably one preferred instance of a type, TOFU provides a declaration mechanism. For example, to state that there is a default function of type $\prod [A].\ A \to \mathit{Unit}$ where *Unit* is the type containg a single value *unit*, one may give it explicitly as

$$\textbf{instance}\ \lambda\,[A],\ a.\ \mathit{unit}$$

Instances are first-class citizens of the language: They behave exactly like ordinary definitions, except that they are identified by *type* rather than by *name*. An instance of type T can be referred to as **some** T, or it can be substituted implicitly for omitted arguments where inference is not possible.

7 Conclusion

Total functional programming works! It provides a semantically sound basis for the evaluation of the stylized mathematical documents that arise from certain formal methods in software engineering. The expressive power of TOFU and the underlying CC are sufficient to express any typical function occurring in such a document, Turing-completeness is not required.

With the techniques presented in this paper, the formal overhead imposed by the strong type system can be reduced below the level widely accepted for strongly typed non-functional programming languages such as Java or C#.

TOFU is intended to be compiled and actually run. Whether the performance is sufficient to produce actual system prototypes or test oracles, or whether TOFU is doomed to share the fate of other total languages as an academic sandbox, is yet unknown.

7.1 Related Work

There is a variety of total functional programming languages around. While some such as Charity have been designed to demonstrate mathematical beauty, the more common goal is to provide, via Curry–Howard correspondence, a logically sound language for an interactive proof system, for instance in Coq [2], Epigram [7] or Agda [10]. All of these have in common that (co)inductive data type definitions and case distinction by pattern matching are built into the core calculus. TOFU, on the other hand, is based on pure CC and uses the powers of the type system to define data types at the program level, via the categorical notions of functors and (co)algebras.

7.2 Future Work

TOFU is an enhanced reimplementation of its precursor FCN described in [19]. Both are implemented in and compile to Java. While the core of design concepts has become clearer in the process, the system is still missing some compiler infrastructure, to the effect that the implementation of Parnas's generic model of tabular expressions, which had been running in FCN, is currently not working in TOFU. Other examples of useful real-world programs are not yet available. The practical performance of the TOFU compiler and its heart, the type checker, appears satisfactory when compiling the TOFU base library, but remains to be tested on a larger scale.

Besides expansion of the TOFU library which currently covers very basic topics such as categories, simple (co)free data types and utility functions, two particularly desirable enhancements of the type system are a topic of active research:

1. The bottom-up approach to inference does not match well with the point-free style of programming encouraged by the absence of pattern matching. Often, one has to choose between either eta-expanding an expression or giving optional arguments explicitly. An additional top-down inference would be needed to solve this dilemma.
2. The mechanism for declaring preferred instances of types works only for single closed instances. A mechanism to declare a family of instances, indexed by one or more *free* type variables, at once as it possible in Haskell, is currently not available. To what degree instances of polymorphic types with *bound* type variables can play the same role is unknown.

References

1. Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
3. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report 92/480/18, Department of Computer Science, The University of Calgary, 1992.
4. Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.
5. Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In *Trends in Functional Programming*, volume 7, pages 157–172. Intellect, 2007.
6. Donald E. Knuth, Tracy Larrabee, and Paul M. Roberts. *Mathematical Writing*. Mathematical Association of America, 1989.
7. Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2005.

8. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144. Springer, 1991.
9. Ivan Miljenovic. Functions all the way down! lambda calculus and church encoding. `http://ivanmiljenovic.files.wordpress.com/2009/05/functionsallthewaydown.pdf`, 2009. Blog entry, retrieved 2010–06–20.
10. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
11. David L. Parnas. Tabular representation of relations. Technical Report 260, Telecommunications Research Institute of Ontario, McMaster University, Hamilton, 1992.
12. David L. Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, 1993.
13. David L. Parnas. Inspection of safety critical software using function tables. In *Proceedings of IFIP World Congress 1994*, volume III, pages 270–277, 1994.
14. David L. Parnas and J. Madey. Functional documentation for computer systems engineering. Technical Report 237, Telecommunications Research Institute of Ontario, McMaster University, Hamilton, 1992.
15. David L. Parnas and Dennis K. Peters. An easily extensible toolset for tabular mathematical expressions. In *Proceedings of the Fifth International Conference on Tools And Algorithms For The Construction and Analysis Of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 1999.
16. Frank Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. In *Studies in Logic and the Foundation of Mathematics, IFCoLog*, 2008. Festschrift in Honor of Peter B. Andrews on His 70th Birthday.
17. H. Shen, J.I. Zucker, and D.L. Parnas. Table transformation tools: Why and how. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96)*, pages 3–11. IEEE, 1996.
18. Baltasar Trancón y Widemann. Church vs. Curry. In Michael Hanus and Bernd Braßel, editors, *Programmiersprachen und Rechenkonzepte*, number 0915 in Technische Berichte. Institut für Informatik, Christian-Albrechts-Universität zu Kiel, 2009.
19. Baltasar Trancón y Widemann and David L. Parnas. Tabular expressions and total functional programming. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages (IFL 2007), Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 219–236. Springer, 2008.
20. David A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.
21. Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: type inference for higher rank and impredicativity. In *Proceedings of ICFP*, pages 251–262, 2006.
22. Philip Wadler. Recursive types for free! `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`, 1990. Manuscript, retrieved 2010–06–20.

Towards an Orchestrated Approach for Annotation Verification*

Adrian Prantl[1], Jens Knoop[1], Raimund Kirner[2], Albrecht Kadlec[2], and Markus Schordan[3]

[1] Vienna University of Technology, Institute of Computer Languages, A-1040 Vienna, Austria, {adrian,knoop}@complang.tuwien.ac.at
[2] Vienna University of Technology, Institute of Computer Engineering, A-1040 Vienna, Austria, {raimund,albrecht}@vmars.tuwien.ac.at
[3] University of Applied Sciences Technikum Wien, A-1200 Vienna, Austria, schordan@technikum-wien.at

Abstract. WCET analyzers commonly rely on user-provided annotations such as loop bounds, recursion depths, region- and program constants. This reliance on user-provided annotations has an important drawback. It introduces a Trusted Annotation Base into WCET analysis without any guarantee that the user-provided annotations are safe, let alone tight. Hence, safety and accuracy of a WCET analysis cannot be formally established. In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding tighter time bounds. Fundamental to our approach is to apply *model checking* in concert with other less expensive program analysis techniques, and the coordinated application of two algorithms for *binary tightening* and *binary widening*, which control the application of the model checker and hence the computational costs of the approach. Though in this paper we focus on the control of model checking by binary tightening and widening, this is embedded into a more general approach in which we apply an array of analysis methods of increasing power and computational complexity for proving or disproving relevant time bounds of a program. First practical experiences using the sample programs of the Mälardalen benchmark suite demonstrate the usefulness of the overall approach. For most of these benchmarks we were able to empty the trusted annotation base completely, and to tighten the computed WCET considerably.

Keywords: Real-time systems, worst-case execution time analysis, program annotations, program analysis, binary tightening and widening.

* This article is an extended version of the paper published in the Proceedings of the 9th Int'l Workshop on Worst-Case Execution Time Analysis (WCET'09). This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (CoSTA) under contract No P18925-N13 and the research project "'Sustaining Entire Code-Coverage on Code Optimization" under contract No P20944-N13 (SECCO), and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

1 Motivation

The computation of loop bounds, recursion depths, region- and program constants is undecidable. It is thus commonly accepted that WCET analyzers rely to some extent on user-assistance for providing bounds and constants. Obviously, this is tedious, complex, and error-prone. State-of-the-art approaches to WCET analysis thus provide for a fully automatic preprocess for computing required bounds and constants using static program analysis. This unburdens the user since it reduces his assistance to bounds and constants, which cannot be computed automatically by the methods employed by the preprocess. Typically, these are classical data-flow analyses for constant propagation and folding, range analysis and the like, which are particularly cost-efficient but may fail to verify a bound or the constancy of a variable or term. WCET analyzers then rely on user-assistance to provide the missing bounds which are required for completing the WCET analysis. This introduces a *Trusted Annotation Base* (*TAB*) into the process of WCET analysis. The correctness (safety) and optimality (tightness) of the WCET analysis depends then on the safety and tightness of the bounds of the TAB provided by the user.

In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding tighter time bounds.

Figure 1 illustrates the general principle of our approach. At the beginning the entire annotation base is given by the user, which is assumed and trusted to be correct, thus we call it TAB. Using model checking we aim to verify as many of these user-provided facts as possible. In this process we shrink the trusted fraction of the annotation base and establish a growing verified annotation base. In Figure 1 the current state in this process is visualized as the horizontal bar. In our approach we are lowering this bar, representing the decreasing fraction of trust to an increasing fraction of verified knowledge, the so-called *Verified Annotation Base* (VAB), and thus transfer *trusted user-belief* into *verified knowledge* by proving or disproving the annotations. During this verification we also strive for tightening the proven annotations, given by the fraction of *tightened* annotations in Figure 1.

Besides the verification of the annotation base we also use static program analysis to refine the time bounds of the annotation base. Using this technique we extend the annotation base, denoted by the additional *extended* annotations at the rightmost side of Figure 1.

2 Verifying and Sharpening the Trusted Annotation Base

The process of transforming the trusted annotation base into a verified annotation base comprises two major steps. Firstly we discuss how formal verification techniques can be used to shrink the annotation base. Secondly, we also discuss how these techniques can be applied to improve the precision of the verified annotations.

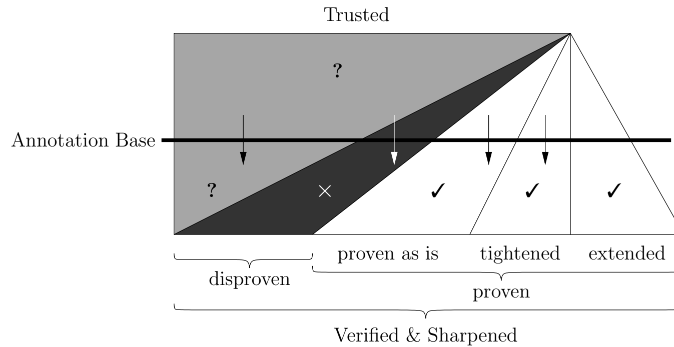

Fig. 1. The annotation base: shrinking the trusted annotation base and establishing verified respective falsified knowledge about the program

2.1 Shrinking and Verifying the Trusted Annotation Base

The automatic computation of bounds by the preprocesses of current approaches to WCET analysis is a step towards keeping the trusted annotation base small. In our approach we go a step further to shrinking the trusted annotation base. In practice, we often succeed to empty it completely.

A key observation is that a user-provided bound – which the preprocessing analyses were unable to compute – can not be checked by them either. Hence, verifying the correctness of the corresponding user annotation in order to move it *a posteriori* from the trusted annotation base to the verified knowledge base requires another, more powerful and usually computationally more costly approach. For example, there are many algorithms for the detection of copy constants, linear constants, simple constants, conditional constants, up to finite constants detecting the different classes of constants at different costs [7]. This provides evidence for the variety of available choices for analyses using the example of constant propagation and folding. While some of these algorithms might in fact well be able to verify a user annotation, none of these algorithms is especially prepared and suited for solely verifying a data-flow fact at a particularly chosen program location, a so-called *data-flow query*. This is because these algorithms are exhaustive in nature. They are designed to analyze whole programs. They are not focused towards deciding a data-flow query, which is the domain of *demand-driven program analyses* [3, 6]. Like for the more expressive variants of constant propagation and folding, however, demand-driven variants of program analyses are often not available.

In our approach, we thus propose to use *model checking* for the *a posteriori* verification of user-provided annotations. Model checking is tailored for the verification of data-flow queries. Moreover, the development of software model checkers made tremendous progress in the past few years and they are now

available off-the-shelf, such as Blast [1] and the CBMC [2] model checkers. In our experiments reported in Section 4 we used the CBMC model checker.

The following example demonstrates the ease and elegance of using a model checker to verify a loop bound, which we assume could not be automatically bounded by the program analyses used. The program fragment on the left-hand side of Figure 2 shows a loop together with a user-provided annotation of the loop. The program on the right-hand side shows the transformed program which is presented to CBMC to verify or refute the user-provided annotation shown in the program on the right-hand side:

```
int binary_search(int x) {
  int fvalue, mid, low = 0, up = 14;
  fvalue = (-1);
  /* all data are positive */


  while(low <= up){
#pragma wcet_trusted_loopbound(0..7)

    mid = low + up >> 1;
    if (data[mid].key == x) {
      /* found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else if (data[mid].key > x)
    up = mid - 1;
    else low = mid + 1;


  }



  return fvalue;
}
```

→

```
int binary_search(int x) {
  int fvalue, mid, low = 0, up = 14;
  fvalue = (-1);
  /* all data are positive */

  unsigned int _bound = 0;
  while(low <= up){


    mid = low + up >> 1;
    if (data[mid].key == x) {
      /* found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else if (data[mid].key > x)
    up = mid - 1;
    else low = mid + 1;

    _bound += 1;
  }
  assert(_bound >= 0
      && _bound <= 7);

  return fvalue;
}
```

Fig. 2. Providing loop bound annotations for the model checker

In this example the CBMC model checker comes up with the answer “yes,” i.e., the loop bound provided by the user is safe; allowing thus for its movement from the trusted to the verified annotation base. If, however, the user were to provide a bound ≤ 3 as annotation, model checking would fail and produce a counter example as output. Though negative, this result would still be most valuable. It allows for preventing usage of an unsafe trusted annotation in a subsequent WCET analysis. Note that the counter example itself, which in many applications is the indispensable and desired output of a failed run of a model checker, is not essential for our application. It might be useful, however, to present it to the user when asking for another candidate of a bound, which can then be subject to *a posteriori* verification in the same fashion until a safe bound is eventually found.

2.2 Sharpening the Time Bounds

We introduce a more effective approach to come up with a safe and even tight bound, if existing, which does not even rely on any user interaction. Fundamental for this are the two algorithms *binary tightening* and *binary widening* and their coordinated interaction. The point of this coordination is to make sure that model checking is applied with care as it is computationally expensive.

Binary Tightening. Suppose a loop bound has been proven safe, e.g. by verifying a user-provided bound by model checking or by a program analysis. Typically, this bound will not be tight. In particular, this will hold for user-provided bounds. In order to exclude channeling an unsafe bound into the trusted annotation base, the user will generously err on the side of caution when providing a bound. This suggests the following iterative approach to tighten the bound, which is an application of the classical pattern of the binary search algorithm, thus called *binary tightening* in our scenario.

Let b_0 denote the value of the initial bound, which is assumed to be safe. Per definition b_0 is a positive integer. Then we call procedure *binaryTightening* with the interval $[0..b_0]$ as argument, where $binaryTightening([low..high])$ is defined as follows:

1. Let $m = \lceil \frac{low+high}{2} \rceil$.
2. ModelCheck(m is a safe bound):
3. yes: $low = m$: **return** m
   $low = m - 1$: ModelCheck(low is a safe bound)
   yes: **return** low no: **return** m
   otherwise: **return** $binaryTightening([low..m])$
4. no: $high = m$: **return** $false$
   $high = m + 1$: ModelCheck($high$ is a safe bound)
   yes: **return** $high$ no: **return** $false$
   otherwise: **return** $binaryTightening([m..high])$

Obviously, *binaryTightening* terminates. If it returns *false*, a safe bound tighter than that of the initial bound b_0 could not be established. Otherwise, i.e., if it returns value b, this value is the least safe bound. This means b is tight. If it is smaller than b_0, we succeeded to sharpen the bound.

Binary widening described next allows for proceeding in the case where a safe bound is not known *a priori*. If a safe bound (of reasonable size) exists, binary widening will find one, without any further user interaction.

Binary Widening. Binary widening is dual to binary tightening. Its functioning is inspired by the risk-aware gambler playing roulette, who exclusively bets on 50% chances like red and black. Following this strategy, in principle, any loss can be flattened by doubling the bet the next game. In reality, the maximum bet allowed by the casino or the limited monetary resources of the gambler, whatever is lower, prevent this strategy to work out in reality. Nonetheless, the idea of an

externally given limit yields the inspiration for the *binary widening* algorithm to avoid looping if no safe bound exists. A simple approach is to limit the number of recursive calls of binary widening to a predefined maximum number. The version of binary widening we present below uses a different approach. It comes up with a safe bound, if one exists, and terminates, if the size of the bound is too big to induce a useful WCET bound, or does not exist at all. This directly corresponds to the limit set by a casino to a maximum bet.

Let b_0 be an arbitrary number, $b_0 \geq 1$, and let max be the maximum value for a safe bound considered reasonable. Then we call procedure $binaryWidening$ with b_0 and max as arguments, where $binaryWidening(b, limit)$ is defined as follows:

1. if $b > limit$: **return** false
2. ModelCheck(b is a safe bound):
3. yes: **return** b
4. no: **return** $binaryWidening(2 * b, limit)$

Obviously, $binaryWidening$ terminates.[4] If it returns $false$, at most an unreasonably large bound exists, if at all. Otherwise, i.e., if it returns value b, this value is a safe bound. The rationale behind this approach is the following: if a safe bound exists, but exceeds a predefined threshold, it can be considered practically useless. In fact, this scenario might indicate a programming error and should thus be reported to the programmer for inspection. A more refined approach might set this threshold more sophisticatedly, by using application dependent information, e.g., such as a coarse estimate of the execution time of a single execution of the loop and a limit on the overall execution time budgeted for this loop.

Coordinating Binary Widening and Tightening. Once a safe bound has been determined using binary widening, binary tightening can be used to compute the uniquely determined safe and tight bound. Because of the exponential resp. logarithmic behavior in the selection of arguments for binary widening and tightening, model checking is called moderately often. This is the key for the practicality of our approach, which we implemented in our WCET analyzer TuBound, as described in Section 3. The results of practical experiments we conducted with the prototype implementation are promising. They are reported in Section 4.

3 Implementation within TuBound

TuBound [9] is a research WCET analyzer tool working on a subset of the C++ language. It is unique for uniformly combining static program analysis, optimizing compilation and WCET calculation. Static analysis and program optimization are performed on the abstract syntax tree of the input program. TuBound is

[4] In practice, the model checker might run out of memory before verifying a bound, if it is too large, or may take too much time for completing the check.

built upon the SATIrE program analysis framework [12] and the TERMITE program transformation environment.[5] TuBound features an array of algorithms for loop analysis including sophisticated methods for nested loops. The supported algorithms exhibit different trade-offs between accuracy and computation cost. A detailed account of these methods can be found in [8].

3.1 Shrinking the TAB

By using a model checker it is possible to verify or falsify user-provided annotations. It can thus be used to find errors in the trusted annotation base. In our implementation (which is detailed in the following section) we achieve this by translating annotations into a combination of counter variables and assertion statements. The result of this is illustrated in Figure 3: In contrast to loop counters, which are initialized upon each entry of a loop, *markers* provide a mechanism to describe global relations (*constraints*) between entities of the control flow graph of a program. They are therefore translated into static variables that are initialized to zero at the program start.

If the model checker succeeds, it is safe to move the annotations from the trusted annotation base into the verified knowledge base. If the model checker comes up with a counterexample, the conflicting annotations need to be removed.

3.2 Binary Widening and Tightening

The interfacing with the model checker necessary for the binary widening/tightening algorithms is implemented by means of a dedicated TERMITE source-to-source transformer T. This requires to translate user-provided annotations into assert statements. For simplicity and uniformity we assume that all loops are structured. In our implementation unstructured goto-loops are thus transformed into while-loops, where possible. This is done in a pre-pass by another TERMITE transformer T'. On while-loops the transformer T works by locating the first occurrence of a `wcet_trusted_loopbound(Lower..Upper)` annotation in the program source and then proceeds to rewrite the encompassing loop as illustrated in the example of Figure 5. An excerpt of the source-to-source transformer T is given in Figure 4.[6] Surrounding the loop statement, a new compound statement is generated, which accommodates the declaration of a new unsigned counter variable which is initialized to zero upon entering the loop. Inside the loop, an increment statement of the counter is inserted at the very first location. After the loop, an assertion is generated which states that the count is at most of value N, where this value is taken from the annotation (cf. Figure 3).

The application of the transformer is controlled by a driver, which calls the transformer for every trusted annotation contained in the source code. Depending on the result of the model checker and the coordinated application of the

[5] `http://www.complang.tuwien.ac.at/adrian/termite`

[6] For better readability, the extra arguments containing file location and other bookkeeping information are replaced by "...".

```
...

#pragma wcet_marker(_label194)
for (k = 0; k < row; k++)
{
#pragma wcet_marker(_label188)

  wmax = 0;


    for (i = k; i < row; i++)
    {
#pragma wcet_marker(_label184)

      w = fabs (a[i][k]);
      if (w > wmax)
      {
#pragma wcet_marker(_label182)

        wmax = w;
        r = i;
      }
#pragma wcet_trusted_loopbound(3..3)
#pragma wcet_trusted_constraint(
          _label184=<_label194*6)
    }




  pivot = a[r][k];
  api = fabs (pivot);
  ...
#pragma wcet_trusted_loopbound(3..3)
#pragma wcet_trusted_constraint(
          _label188=<_label194*3)
}

...
```

Annotated source code

→

```
...
static unsigned int _label194=0;
_label194++;
unsigned int _loop_label188=0;
for (k = 0; k < row; k++)
{
  static unsigned int _label188=0;
  _label188++;
  wmax = 0;
  {
    unsigned int _loop_label184=0;
    for (i = k; i < row; i++)
    {
      static unsigned int _label184=0;
      _label184++;
      w = fabs (a[i][k]);
      if (w > wmax)
      {
        static unsigned int _label182=0;
        _label182++;
        wmax = w;
        r = i;
      }

      _loop_label184++;
      assert (_label184 <= _label194*6);
    }
    assert ((_loop_label184 <= 3)
         && (_loop_label184 >= 3));
  }

  pivot = a[r][k];
  api = fabs (pivot);
  ...
  _loop_label188++;
  assert (_label188 <= (_label194 * 3));
}
assert ((_loop_label188 <= 3)
     && (_loop_label188 >= 3));
```

Intermediate code for the model checker

Fig. 3. Verification of Annotations

```
assertions(..., Statement, AssertedStatement) :-
  Statement = while_stmt(Test, basic_block(Stmts, ...), ...),
  get_annot(Stmts, wcet_trusted_loopbound(Lower..Upper), _),

  var_decl(unsigned_int, '_bound', 0, ..., CounterDecl),
  var_ref_exp('_bound', unsigned_int, Counter),
  plusplus_expr_stmt(Counter, ..., Count),
  build_expr_stmt(assert((Counter =< Higher) && (Counter >= Lower)),
                  ..., CounterAssert),

  AssertedStatement =
    basic_block([CounterDecl,
                 while_stmt(Test, basic_block([Count|Stmts], ...), ...),
                 CounterAssert], ...).
```

Fig. 4. Excerpt from the source-to-source transformer T

algorithms for binary widening and tightening, the value and the status of each annotation is updated. In the positive case, this means the status is changed from *trusted annotation* to *verified knowledge*, and the value of the originally trusted bound is replaced by the now verified, possibly sharper, bound. Figure 5 shows a snapshot of processing the *janne_complex* benchmark. In this figure, the status and value changes are highlighted by different colors.

```
...
int complex(int a, int b)
{
  while(a < 30) {
#pragma wcet_trusted_loopbound(0..30)


      while(b < a) {
#pragma wcet_trusted_loopbound(0..30)


        if (b > 5)
          b = b * 3;
        else
          b = b + 2;
        if (b >= 10 && b <= 12)
          a = a + 10;
        else
          a = a + 1;
      }



      a = a + 2;
      b = b - 10;
    }
  return 1;
}
...
```

Containing two trusted loop annotations

→

```
...
int complex(int a, int b)
{
    while(a < 30) {
#pragma wcet_loopbound(0..16)
      {
        unsigned int _bound = 0;
        while(b < a){
#pragma wcet_trusted_loopbound(0..30)
          ++_bound;

         if (b > 5)
           b = b * 3;
         else
           b = b + 2;
         if (b >= 10 && b <= 12)
           a = a + 10;
         else
           a = a + 1;
      }
      assert(_bound >= 0
          && _bound <= 30U);
    }
    a = a + 2;
    b = b - 10;
  }
  return 1;
}
...
```

Outer loop annotation verified and tightened, inner loop currently being checked

Fig. 5. Illustrating trusted bound verification and tightening

4 Experimental Results

We implemented our approach as an extension of the TuBound WCET analyzer and applied the extended version to the well-known Mälardalen WCET benchmark suite. As a baseline for comparison we used the 2008 version of TuBound, which took part in the WCET Tool Challenge 2008 [5], later on referred to as the basic version of TuBound. In the spirit of the WCET Tool Challenge [4, 5] we do encourage authors of other WCET analyzers to carry out similar experiments.

Our experiments were guided by two questions: "Can the number of automatically bounded loops be increased significantly?" and "How expensive is the process?". The benchmarks were performed on a 3 GHz Intel Xeon processor running 64-bit Linux. The model checker used was CBMC 2.9, which we applied

to testing loop bounds up to the size of $2^{13} = 8192$ using a timeout of 120 seconds and a maximum unroll factor of $2^{13}+1$. The "compress" and "whet" benchmarks contained unstructured goto-loops; as indicated in Section 3.2 these were automatically converted into do-while loops beforehand by a separate TERMITE transformation.

Benchmark	Loops	TuBound basic	TuBound with Model Checking	Runtime
bs	1	0/1	1/1	0.03s
janne_complex	2	0/2	2/2	0.18s
nsichneu	1	0/1	1/1	5.59s
statemate	1	0/1	1/1	0.06s
qsort-exam	6	0/6	4/6	0.02s
fft1	11	6/11	9/11	0.43s
minver	17	16/17	17/17	0.06s
duff	2	1/2	1/2	0s
whet	11	10/11	10/11	0s
adpcm	18	15/18	15/18	timeout
compress	8	2/8	2/8	timeout
fir	2	1/2	1/2	timeout
insertsort	2	0/2	0/2	timeout
lms	10	6/10	6/10	timeout
select	4	0/4	0/4	timeout
bsort100	3	3/3	3/3	–
cnt	4	4/4	4/4	–
cover	3	3/3	3/3	–
crc	3	3/3	3/3	–
edn	12	12/12	12/12	–
expint	3	3/3	3/3	–
fdct	2	2/2	2/2	–
fibcall	1	1/1	1/1	–
jfdctint	3	3/3	3/3	–
lcdnum	1	1/1	1/1	–
ludcmp	11	11/11	11/11	–
matmult	5	5/5	5/5	–
ndes	12	12/12	12/12	–
ns	4	4/4	4/4	–
qurt	1	1/1	1/1	–
sqrt	1	1/1	1/1	–
st	5	5/5	5/5	–
recursion	0	–/–	–/–	–
Total	170	131/170	144/170	–

Table 1. Results for the Mälardalen benchmarks

Our findings are summarized in Table 1. Column three of this table shows the number of loops that can be bounded by the basic version of TuBound; column

four shows the total number of loops the extended version of TuBound was able to bound. It is important to note that the model checker was only invoked for benchmarks where at least one loop could not be bounded by the basic version of TuBound. The last column shows the accumulated runtime of the model checker.

Comparing columns three and four reveals the superiority of the extended version of TuBound over its basic variant. The extended version raises the total number of bounded loops from 77% to 85%.

Considering column five, it can be seen that the model checker terminates quickly on small problems but that the runtime and space requirements can increase to practically infeasible amounts on problems suffering from the state explosion problem. Such a behavior can be triggered, if the initialization values which are part of the majority of the Mälardalen benchmarks are manually invalidated by introducing a faux dependency on e.g. `argc`. This demonstrates that model checking is to be used with care or the model checker be fed with additional information guiding and simplifying the verification task.

The fully-fledged variant of our approach, which we highlight in the next section is tailored towards this goal.

5 Extensions: The Fully Fledged Approach

The shrinking of the trusted annotation base and tightening of time bounds, as described in Section 2, is based on model checking. Based on our experience, we believe that the model checking approach can be especially valuable in the real world when (i) it is combined with advanced program slicing techniques to reduce the state space and (ii) the results of static analyses (like TuBound's variable-interval analysis) are used to narrow the value ranges of variables, thus regaining a feasible problem size. This leads to the following extension of our approach to improve efficiency:

1. By using a pool of analysis techniques with different computational complexity: As shown in Figure 6, model checking is considered as one of the most complex analysis methods. On the other side, techniques like *constant propagation* or *interval analysis* are relatively fast. Thus we are interested in exploiting the fast techniques wherever beneficial and using the relatively complex techniques rarely.
2. By using a smart activation mechanism for the different analysis techniques: As shown in Figure 7 we are interested in the interaction of the different analysis techniques, which is controlled by the *Verification Controller*. We do not aim to use the pool of analysis techniques in waves of ascending complexity, i.e., first applying the fast techniques and then gradually shifting towards the more complex techniques. Instead we aim for a smart interaction of the different analysis techniques.
   For example, the Verification Controller could act in a recursive way as shown by the arrows of data flow in Figure 6: Techniques of similar complexity are applied in a round-robin fashion, until no further improvements are obtained. Then, whenever a technique with relatively high computational complexity

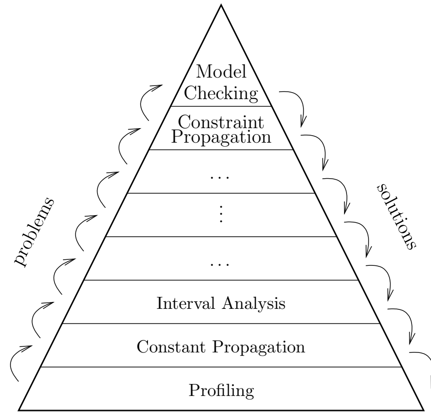

Fig. 6. Different complexity of the analysis techniques

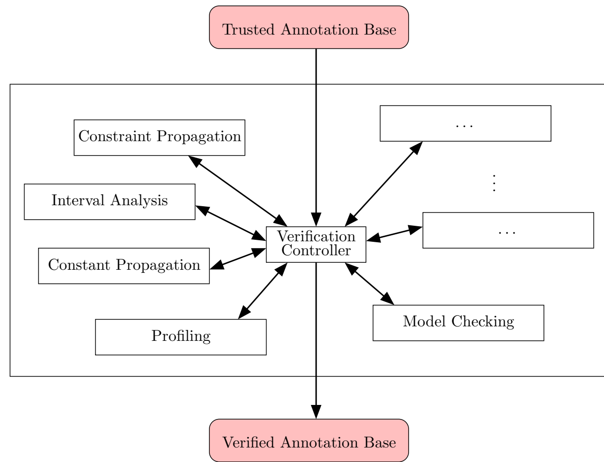

Fig. 7. Pool of complementary analysis techniques

has been applied, we again apply techniques of relatively low complexity to compute the closure of flow information based on previously obtained results; thus *squeezing* the annotation base.

3. By using a tradeoff between the level of trust we accept and the level of tightness we require, we can keep the analysis costs within a reasonable bound. As shown by the vertical line in Figure 8, whenever we bound the permitted verification costs to a certain limit, we can chose the level of trust to tune the tightness of the annotation base. Or if we require a certain level of tightness, we can chose the level of trust to tune the verification costs, as shown by the horizontal dot-dashed line in Figure 8.

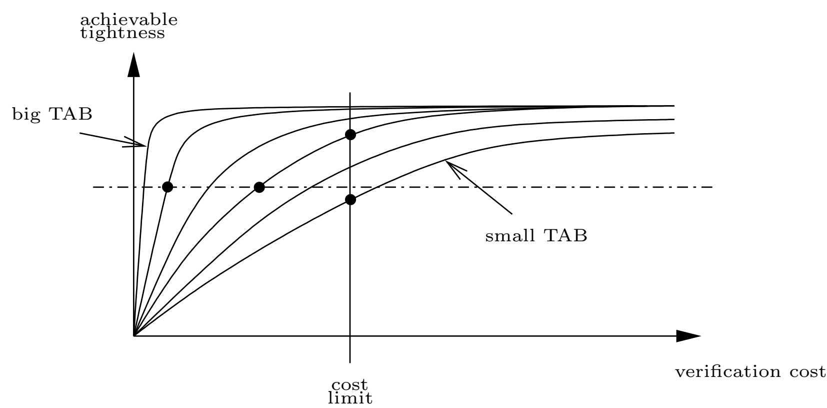

Fig. 8. Tradeoff between accepted trust and achievable tightness of the annotation base

Besides the use of static analysis techniques, we also suggest that profiling techniques are beneficial to guide the heuristics to be used within our static analysis techniques. For example, execution samples obtained by profiling can be used to elicit propositions to be verified by model checking. Profiling can also act as an inexpensive way to filter incorrect user annotations, thus narrowing the TAB by falsification. As with any contradiction between user annotations and analysis results, this should be disregarded in further analysis and be reported to the user.

The fully fledged approach envisioned in this section provides the promising potential as a research platform for complementing program analysis techniques.

6 Conclusions

Model checking has been used before in the context of WCET analyzers. Examples of our own related work are the ForTAS [13], MoDECS [14], and ATDGEN

projects [10, 11], which are concerned with measurement-based WCET analysis. In these three projects, model checking is used to generate test data for the execution of specific program paths. Intuitively, in these applications the model checker is presented with formulæ stating that a specific program path is infeasible. If these formulæ can be refuted by the model checker, the counter examples generated provide the test data ensuring the execution of the particular paths. Otherwise, the paths are known to be infeasible. Hence, the search for test data is in vain. In these applications the counter examples generated in the course of failed model checker runs are the truly desired output, whereas successful runs are of less interest just stopping the search for test data for the path under consideration. This is in contrast to our application of shrinking the trusted annotation base. In our application, the counter example of a failed model checker run is of little interest. We are interested in successful runs of the model checker as they allow us to change a *trusted annotation* into *verified knowledge*. This opens a new application domain for model checking in the field of WCET analysis. Our preliminary practical results demonstrate the practicality and power of this approach.

References

1. Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, October 2007.
2. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
3. Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992 – 1030, 1997.
4. Jan Gustafsson. The WCET tool challenge 2006. In *Preliminary Proc. 2nd International IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.
5. Niklas Holsti, Jan Gustafsson, Guillem Bernat (eds.), Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET Tool Challenge 2008: Report. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 149–171, Prague, Czech Republic, July 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3.
6. Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE-3*), pages 104 – 115, 1995.
7. Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4.

8. Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level wcet analysis. In *Proc. 18th International Workshop on Logic-based methods in Programming Environments (WLPE 2008)*, pages 77–89, Udine, Italy, December 2008.
9. Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound – a conceptually new tool for worst-case execution time analysis. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 141–148, Prague, Czech Republic, July 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3.
10. Bernhard Rieder. *Measurement-Based Timing Analysis of Applications written in ANSI-C*. PhD thesis, Technische Universität Wien, Vienna, Austria, 2009.
11. Bernhard Rieder, Ingomar Wenzel, and Peter Puschner. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement-based WCET analysis. In *Proc. 6th International Workshop on Intelligent Solutions in Embedded Systems (WISES 2008)*, Regensburg, Germany, July 2008.
12. Markus Schordan. Source-To-Source Analysis with SATIrE – an Example Revisited. In *Proceedings of Dagstuhl Seminar 08161: Scalable Program Analysis*. Germany, Dagstuhl, April 2008.
13. Vienna University of Technology and TU Darmstadt. The ForTAS project. Web page (`http://www.fortastic.net`), 2009. Accessed in December 2009.
14. Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. In *Proc. 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, October 2008.

A Satisfiability Modulo Theories Memory-Model and Assertion Checker for C

Jakob Zwirchmayr

Institute of Computer Languages (E185) - Compilers and Languages Group (E185-1)
Vienna University of Technology, jakob@complang.tuwien.ac.at

Abstract. This paper presents SmacC, a tool for software verification and SMT benchmark generation. It builds upon a state-of-the-art SMT solver, Boolector, developed at FMV institute at JKU, Linz. SmacC gets as input a program that lies in the supported subset of C and transforms it to SMT formulas. The SMT representation allows verification of properties that are required to hold on the program. SmacC symbolically executes the programs source code, establishing an SMT (memory-) model for the program. Some statements and expressions require the construction of SMT formulas specifying properties about them, the SMT solver decides their satisfiability. If properties checked do not hold on the SMT representation, they do not hold on the real program. SmacC can generate SMT benchmarks by dumping the SMT instances for those checks.

1 Introduction

SmacC symbolically executes a C program in order to find defects or to create benchmarks to be replayed by an SMT solver. A program consists of a set of instructions and some memory storing instructions and data of the program. When the program is executed, instructions are fetched from memory and then executed by the CPU, repeatedly, in some cases altering data in memory. When the program is symbolically executed by SmacC, instructions are extracted from the source and stored in abstract syntax trees (ASTs), organised in a code-list (CL). The CL is then analyzed, extracting paths through the program. A Boolector array variable models the memory of the program. Execution of a path establishes constraints on the array, reflecting valid memory. Additionally, some statements executed can be checked for defects by constructing an SMT formula representing an error condition and checking its satisfiability using the SMT solver.

The front-end of SmacC consists of an input buffer, a lexer and a parser that parses the source code into ASTs and CLs, respectively. The CLs, containing syntax trees, represent paths through the program. Code-lists are the connection to the back-end that symbolically executes them one ofter another, establishing a memory-model in SMT for each path through the program. Loops are handled by loop-unrolling which transforms loops to sequential if statements. When execution of the program might branch, CLs for each branch are generated.

Checks can be dumped to a file as BTOR or SMT-LIB formula to be used as benchmark for an SMT solver. A check is a BTOR formula that must be satisfiable or unsatisfiable on the SMT representation depending on the statement or expression that triggered it. One can differ between two kinds of checks, verification checks:

- Assertion statement: verify that assertion statement cannot fail,
- Return statements: check if the program returns a specified value in all cases or check if a specified return value is possible,
- Path conditions: check if an if / else condition is unsatisfiable

and defect checks:

- Assignment: checks validity of address a value is assigned to,
- Indirection: checks validity of address being dereferenced,
- Division by zero: checks if division by zero is possible,
- Overflow: checks for overflow on arithmetic operations,

2 Boolector and BTOR Format

BTOR was developed initially as native format for SMT solver Boolector, supporting the theory of bit-vectors and the theory of one-dimensional arrays, as supported by SMT solver Boolector. In addition it supports an extension that can be used for model checking [1].
The SMT solver Boolector was developed at the Institute for Formal Models and Verification of the Johannes Kepler University and is an efficient SMT solver for the combination of the quantifier-free fragment of the theory of bit-vectors and extensional theory of arrays and equality. The quantifier-free theory of bit-vectors enables Boolector to solve formulas including modular arithmetic, comparison, two's complement, logical operations, shifting, concatenation and bit-extraction. The theory of arrays allows natural modelling of memory. Fig. 1 shows the basic usage of Boolector in its stand-alone version.

3 Front-End and supported C Subset

Programs supplied to SmacC must compile with an ANSI C compatible compiler, erroneous programs cannot be handled. Gcc was used as compiler to build SmacC and to compile C examples against which the behaviour of SmacC was tested. In general, a program supplied to SmacC should compile with gcc without warnings, with extra warning flags enabled. The following listing summarizes supported constructs:

- A valid translation unit may only contain global variable declarations of the supported types and one function declaration
- `if-else`, `for`, `assert`, `malloc`, `free`, `sizeof`, `return`, `#include`

```
$> cat example.btor
1 array 8 32
2 var 32 index
3 const 8 00000000
4 write 8 32 1 2 3
5 eq 1 1 4
6 root 1 5
$>
```

```
$> boolector example.btor -m -d
sat
index 0
1[0] 0
$>
```

Fig. 1. BTOR file `example.btor` and output of invoking Boolector. Boolector prints a (partial) model in the SAT case when supplying `-m`, while `-d` enables decimal output. In line 1 an array with element width 8 bit and index width 32 bit is constructed. Line 2 declares a 32 bit bit-vector variable named `index`. Line 3 declares an 8 bit bit-vector constant with value 0 that is written to array 1 on position index (2) in line 4, constructing a new array. Line 5 states that array 1 is equal to array 4. Line 6 sets line 5 as root node such that the formula can be checked with Boolector stand-alone version. Boolector returns 'satisfiable' because it is possible that the element at index `index` of array 4 has the same value as the element at index `index` in array 1.

– Non-augmented assignment statements, compound statement, valid C expressions (some restrictions)

The front-end gets as input a C file that contains a translation unit which lies in the supported subset of C. The lexer tokenizes the input stream and the parser creates ASTs according to the expression grammar, organizing them as statement elements in a CL.

4 Back-End

The back-end gets as its input the full CL that was generated by parsing the translation-unit. It extracts and executes paths through the program symbolically by writing to and reading from the BTOR array representing the memory of the program. It generates SMT formulas for the memory layout and checks satisfiability of properties that must hold. Symbolic execution is split into two phases called path-generation (`pathgen`) and BTOR-generation (`btorgen`). Phase one, path-generation, flattens the full CL by unrolling loops up to a certain bound. After flattening path-generation processes the CL, generating a new CL until meeting an element that represents a branching point in the program. When a branching point is met, the CL is duplicated and both paths are processed further. When a path through the program was extracted, BTOR-generation is responsible for the generation of SMT formulas representing the state of the memory of the program. Some elements in the path require construction of SMT formulas to check for certain programming errors. SmacC can also be configured to dump them in both SMT-LIB or BTOR format.

4.1 Path-Generation

Path-generation phase flattens the CL by unrolling iteration-statements to nested sequences of selection-statements. It can be configured up to which bound SmacC unrolls `for` loops. The resulting flat CL is further processed in the path-generation phase, creating separate CLs for branches through the program. When an element in the flattened CL is of kind selection-statement and execution could branch, the CL representing the path through the program up until this point is duplicated and path-generation is called for both branches, generating a CLs for each of them. When a path through the program is fully extracted either after reaching the last element of the input CL or by processing a return-statement element `pathgen` calls `btorgen` which then symbolically executes the path.

```
int main ()
{
  int cond;
  if (cond)
    return cond;
  return 0;
}
```

```
path 0:
CSENTER @ (1,10)
  CSENTER @ (1,12)
   CDECLL @ (2,8)
   CIF @ (4,0)
    CBBEG @ (4,0)
     CRET @ (4,11)
    CBEND @ (5,0)
   CRET @ (5,8)
  CSEXIT @ (6,0)
CSEXIT @ (6,0)
```

```
path 1:
CSENTER @ (1,10)
  CSENTER @ (1,12)
   CDECLL @ (2,8)
   CELSE @ (5,0)
   CRET @ (5,8)
  CSEXIT @ (6,0)
CSEXIT @ (6,0)
```

Fig. 2. Translation-unit and CLs for both paths through the program.

4.2 Btor-Generation

Btor-Generation constructs BTOR expressions for C statements and expressions resulting in SMT formulas. Additionally constraints for the array modelling the programs memory are generated. If an entry in the CL contains an AST representing C expressions the tree is transformed to BTOR expressions by calling `btorgen_generate`. Some entries lead to (verification- or defect-) checks, usually resulting in one or more SAT-checks by Boolector. Variable declarations require the construction of Boolector variables, stored with the symbols and used as addresses for the memory array. When an identifier is parsed in an expression the Boolector variable for the symbol can be looked up in the AST node for the expression. Variable declarations in the code also require updates to the SMT formula representing constraints on the programs memory.

4.3 Memory Model

The memory model is inspired by the memory model usually used in UNIX systems. It is established by an SMT formula that constrains the array variable

```
int
main ()
{
  return 0;
}
```

```
 2 const 32 000...000        11 ult 1 5 6
 4 var 32 stack_beg         12 ult 1 6 4
 5 var 32 global_beg        13 and 1 7 -8
 6 var 32 heap_beg          14 and 1 13 -9
 7 eq 1 2 2                 15 and 1 14 -10
 8 ult 1 5 5                16 and 1 15 11
 9 ult 1 4 4                17 and 1 16 12
10 ult 1 6 6                18 root 1 17
```

Fig. 3. A C Program and the BTOR instance for the return statement. The BTOR instance for the return statement `return 0;` is depicted on the right and will briefly be explained: line 2 represents the constant 0, line 4, 5 and 6 represent the BTOR variables necessary to construct the memory model. The BTOR formula for the return statement is constructed in line 7. The rest of the lines form the constraints for the memory layout and are conjuncted with line 7 and selected as root in line 18. Lines 8 to 10 are used negated in line 13 to 15 to formulate the properties that the end of stack, global and heap area must be greater or equal to the beginning of stack, global and heap area. Initially the addresses that represent the end of the memory areas are equal to the addresses that represent the begin of the memory areas. Line 10 and 11 establish the general memory layout which requires that the highest global address is smaller than the lowest heap address which is smaller than the last lowest stack address. Line 17 is the conjunction of the properties mentioned and the formula specifying the return value to be equal to zero.

which models the memory of the program. This allows to check whether memory accesses in the program are valid. If a memory access is invalid for the SMT representation it is also invalid for the real program. The UNIX memory model divides memory for a process into three segments [6]:

- Text Segment: machine instructions, executable code
- Global / Data Segment: global variables, constant strings, but also dynamic memory
- Stack Segment: local variables, parameter variables, grows from high address to low address

SmacC simplifies the UNIX memory model, there is no text segment, the data segment is called global area and is only used for global variables. Memory that is allocated in the data segment by calls to `malloc` is modeled by the heap area. The left-hand side of Fig. 4 is a visualization of the memory layout right after initialization, no variables declared, represented by the following formula:

$$global_beg \leq global_end \wedge global_end < heap_beg \wedge heap_beg \leq heap_end \wedge \\ heap_end < stack_end \wedge stack_end \leq stack_beg \wedge global_beg = global_end \wedge \\ stack_beg = stack_end \wedge heap_beg = heap_end$$

When variables are declared or dynamic memory is allocated the memory-model needs to be updated to include constraints about the variable. Consider the

right-hand side of Fig. 4, visualizing the memory model after a few variables were declared, represented by the following updates to the memory model:

$$i = global_beg \wedge j = global_beg + 4 \wedge global_end = global_beg + 8$$
$$heap_v1 = heap_beg \wedge heap_end = heap_beg + 4$$
$$p = stack_beg - 4 \wedge c = stack_beg - 4 - (4 * 1) \wedge stack_end = stack_beg - 8$$

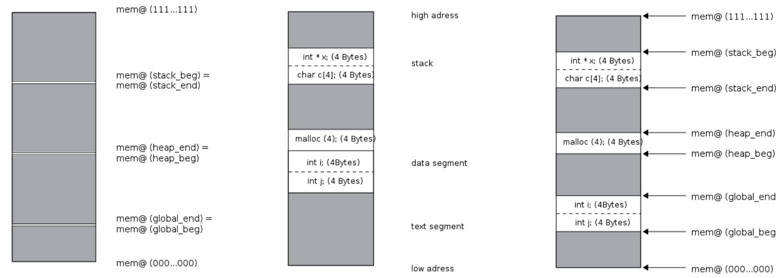

Fig. 4. Simplified view of the UNIX memory-model of a C program and its representation in SmacC. In the left example no variables are declared. In the right example the program has integers `i` and `j` declared as global variables, integer pointer `x` and character array `c` as local variables and 4 bytes allocated on the heap by a call to `malloc`.

SmacC considers the following memory accesses invalid, for the sake of brevity only the first is discussed in this paper.

- Access out of valid memory: an access is considered out of valid memory if it accesses indices that are not indices representing stack area, global area or heap area. Invalid regions are marked grey in Fig. 4.
- Access out-of-bounds: an access is considered out-of bounds if it crosses boundaries of data elements, for example when data from two valid regions is read or written. Out of bounds access can happen at all addresses.

5 Checks

While a path is symbolically executed certain statements and expressions lead to checks. A check is an SMT formula that must be SAT or UNSAT when added to the formulas of the memory-model and checked via Boolector. SmacC checks include those that verify that a memory access is valid in the memory's SMT representation and hence valid in the C program. Furthermore they are used to verify assertions, show that an operation does not lead to an error or show that a path condition cannot be satisfied. The assertion check and the basic memory check are presented.

5.1 Assertion Check

Variations of assertion checks are used to verify program return values and to check for division by zero. Consider the example in Fig. 5.

```
void main ()
{
  int i;
  assert (i);
}
```

btor_vars =
{
$global_beg, global_end,$
$heap_beg, heap_end,$
$stack_beg, stack_end,$
mem, i
}

$$\begin{aligned}\textbf{layout} := {} & global_beg \leq global_end \wedge \\ & global_end < heap_beg \wedge \\ & heap_beg \leq heap_end \wedge \\ & heap_end < stack_end \wedge \\ & stack_end \leq stack_beg \wedge \\ & global_beg = global_end \wedge \\ & heap_beg = heap_end \wedge \\ & i = stack_beg - 4 \wedge \\ & stack_end = stack_beg - 4\end{aligned}$$

$$\begin{aligned}\textbf{assert} := {} & read(mem, i) = 00000000 \wedge \\ & read(mem, i+1) = 00000000 \wedge \\ & read(mem, i+2) = 00000000 \wedge \\ & read(mem, i+3) = 00000000 \wedge\end{aligned}$$

Fig. 5. On the left: assertion statement in a C program and declared Boolector variables. On the right: assumptions about memory layout and the formula representing the assertion.

The conjunction of formulas **layout** $\wedge$ **assert** must be unsatisfiable, otherwise the assertion might fail.

5.2 Basic Memory Check: Arbitrary-but-Fixed

The basic memory check constructs a Boolector bit-vector variable *abf* and uses the SMT formulas for the general memory layout to let *abf* point to an arbitrary address in memory but it is fixed to be outside any valid memory. Then it is checked if the variable *abf* can be equal to the address *addr* being read from or written to. If it is satisfiable that *addr* is equal to *abf* it is shown that the memory access could address an illegal memory address (outside any known memory region, or in a region that was freed by `free`). SmacC checks both the first and last byte of a value being read or written from or to memory. A problem of the basic memory check is that the results of the check can depend on the order in which variables were declared. This effect can also occur in C programs and is hard to capture. The formulas constructed for the check are presented in Fig. 6.

abf_invalid :=
$abf > stack_beg \wedge$
$abf > global_end \wedge$
$abf < heap_beg \wedge$
$abf > heap_end \wedge$
$abf < stack_end \wedge$
$abf < global_beg$

abf_freed :=
$abf \geq free_var_i \wedge$
$abf < free_var_i + free_var_i_size$

check :=
$abf = addr$

Fig. 6. Basic Memory Check: constraining a variable to be outside valid memory or equal to a freed address.

Clearly, because of the constraints on abf, if the SMT formula (**abf_invalid** $\vee$ **abf_freed**) $\wedge$ **check** is satisfiable for any byte of $addr$, then invalid memory is accessed.

6 Limits of the Model

The array memory check (not treated in this paper) has the weaknesses that expressions using pointer arithmetic can fool the array (out-of-bounds) memory check, nevertheless it can be used to verify some pointer arithmetic expressions. If memory allocated by `malloc` is deallocated by `free` it is not used again in following calls to `malloc`. This can lead to out-of-memory situations where `malloc` cannot allocate requested memory, leading to a contradiction in the memory model and hence invalidating reported results. This could even occur if memory deallocated by `free` was reused. If a program allocates all available memory by a call to `malloc` and then allocates additional (unavailable) memory, the assumptions used to construct the memory model can be contradicting, invalidating results of checks following the second call to `malloc`. Assume that the first call to `malloc` allocates all memory from the lowest address to the highest address. Assumptions established for the memory model are (omitting assumptions for global and local memory regions): SmacC assumes that there is no overflow on address calculations. Because of the assumption that the first call to `malloc` forces $heap_end$ to be equal to the highest address, overflow is unavoidable for address calculation of $m2$, a contradiction follows. Another problem emerges from the way path conditions of loops are handled: after unrolling the loop up to the specified bound the loop condition is assumed to be true. If it is the case that the state of the memory contradicts the assumption, then the checks following the loop return wrong results.

7 Related Work

CBMC is a Bounded Model Checker for ANSI C and C++ programs. It allows verifying array bounds, pointer safety, exceptions and user-specified assertions [5]. CBMC takes as input C files and translates the program, merging function definitions from the input files. Instead of producing a binary for execution,

CBMC performs symbolic simulation on the program [4]. CBMC translates refined programs to SAT instances and uses MiniSAT to verify properties.
Recently, preliminary support for SMT solvers (Boolector, CVC3, Yices, and Z3) has been added via the SMT-LIB theory QF_AUFBV [5].
CBMC can also be used to check behavioral consistency of C and Verilog programs (Hardware and Software Equivalence and Co-Verification) [3].
The major difference to SmacC is that CBMC does not establish a full representation for the memory of the program and its layout, instead it uses intermediate variables when accessing variables. CBMC unwinds loops and recursive function calls and transforms the program until it only consists of `if` instructions, assignments, assertions, labels and `goto` instructions [2]. An assertion for each loop verifies that the unwinding bound [2] is large enough, otherwise the bound is increased. Then it is transformed into static single assignment form, consisting of bit-vector equations for constraints and verification conditions. The conjunction of the constraints and the negation of the property is checked for satisfiability. If the conjunction is satisfiable, the property is violated.

8 Benchmarks

The following C files and algorithms were transformed to a BTOR representation, and can be used as benchmarks, timing results are presented in Tab. 1.

- Memcopy: A simple memcpy implementation, copying memory from the source buffer to the destination buffer. Assert that destination buffer contains the same elements as the source buffer after copying.
- Palindrome: implements algorithm to check if a string is a palindrome. If the algorithm concludes that a string is a palindrome, assert that the string fulfills palindrome properties.
- Stringcopy: Similar to memcpy but omitting the third parameter, the number of bytes that must be copied. The loop terminates if null character is read in source buffer which is then copied to the target buffer.
- Power of 3 equality: Compares if a method to compute n^3 using a loop always yields the same result as a method without a loop.

Benchmark	Bound	Boolector	SmacC	CBMC
memcpy.c, array size 30	30	287s	1496s	0.25s
memcpy.c, array size 40	40	565s	5595s	0.33s
memcpy.c, array size 50	50	1114s	7350s	0.34s
palindrome check, n 11	11	639s	3718s	0.18s
palindrome check, n 15	15	1614s	13406s	0.22s
palindrome check, n 16	16	3344s	16220s	0.26s
strcpy array, n 20	20	231s	timeout	0.11s
strcpy array, n 30	30	1430s	timeout	0.15
strcpy array, n 40	40	7684s	timeout	0.20s
power 3 equality	3	timeout	timeout	timeout

Table 1. Benchmarks were run on an Intel® CPU at 2.66GHz with 2GB main memory. Time was measured using the UNIX *time* command. The table compares Boolector stand-alone version to library usage in SmacC and to CBMC.

References

1. Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Lecture Notes in Computer Science (LNCS)*, volume 5505. Springer, 2009. TACAS'09.
2. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. Carnegie Mellon University, 2004.
3. Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. 2003.
4. CProver. The cprover user manual. Available at http://www.cprover.org/cbmc/doc/manual.pdf.
5. Daniel Kroening. Bounded model checking for ansi-c. Available at http://www.cprover.org/cbmc/.
6. Andrew S. Tannenbaum. *Modern Operating Systems, 3rd Edition.* Pearson, Prentice Hall, Upper Saddle River, New Jersey 07458, 2007.

The rCOS methodology and modeler

Volker Stolz*, for the rCOS team

University of Oslo
and
United Nations University
Intl. Institute for Software Technology (UNU-IIST)

Abstract. The rCOS language (Refinement of Component and Object Systems) offers a unique opportunity for teaching formal methods to beginning software engineers. The language combines a mathematical framework for reasoning about component- and object systems with the state of the art in model-based development. UML diagrams are used to capture the static structure of a software system, and the dynamic behaviour of its components. Our use-case driven approach ensures a consistent method of deriving artifacts from an (informal) problem description.
The business logic of the artifacts is then specified in a mathematical framework based on the Unifying Theories of Programming with extension for object orientation. Through refinement (semi-automated or manual), correct executable code can be derived from the specification. Refinement is also applied on the modeling level, for example, to evolve from an object-based system to a component-based system, or transformation of components.
The tool supports various backends for verification, like component compatibility and reactive behaviour (through model checking), refinement checks by theorem proving, and model-based testing. The interplay between tight integration of the different aspects of designing a model and support for different levels of abstractions allows the lecturer to cover the different phases of designing a software system. Yet the rigorous formal foundation ensures consistency and correctness.

1 Introduction

The rCOS methodology is a structured way of designing component-based software in a rigorous manner through a top-down process. It combines long-established practices from object oriented analysis and design [9, 8] with formal methods (see Fig. 1). We show how the integration of various aspects of formal methods lead to a modern teaching- and research tool on introductory Software Engineering.

Based on use cases, a coarse requirements model is developed using the Unified Modeling Language (UML) [15]. We use a standard object oriented model

* The author is partially supported by the ARV grant of the Macao Science and Technology Development Fund.

with classes and single inheritance. Behaviour of provided functionality is specified in a relational calculus, and grouped into components.

Reactive behaviour of a component is given through sequence- and state machine diagrams, where messages/transitions are labelled with method names from the signature of the component. The corresponding contained trace languages have to be consistent. The tool enforces this consistency through model checking.

Then, in semi-automated correct-by-construction refinement steps the developer transforms the model into a platform-independent object-oriented model. Through those transformations, the relational specifications are turned into a pre-executable fragment of the rCOS modeling language (i.e. the fragment that can directly be encoded into an object-oriented programing language).

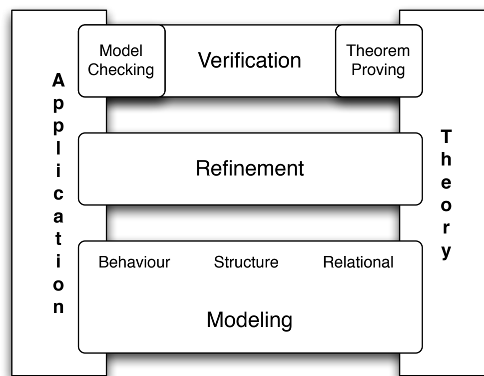

Fig. 1. Overview methodology

Persistent objects are then identified, and allocated to components. Based on the use cases, deployment boundaries are identified that imply different communication mechanisms (direct method invocation on the object-oriented level, or remote method invocation). Code- and test case generation complete the picture.

In the following, we will first discuss the aspects of the initial requirements model in Section 2. Section 3 discussed the refinement steps that will eventually make the model executable. Verification and Validation tasks are described in Section 4.

Editing Process

The entire modeling process in the rCOS modeler is based on what we have termed “a story telling perspective”, or a workflow. As modeling proceeds through the various stages, each stage requires the developer to model a specific aspect of his model. Within a stage, artifacts are related to each other. Sometimes,

such a relation is more semantical, like that the set of labels that can be used in a sequence diagram is defined by the component that the diagram applies to. At other times, a relation is more structural, for example, a use case must be associated to a component, or a component must define a contract. In the latter cases, the editor allows navigation of the strongly related artifacts through context menus, and also presents other cues like highlighting or warnings to the user to remind him of aspects he still has to address in his model.

Our approach also addresses a shortcoming of UML models: the top-level artifact of a UML model is usually an instance of the 'model' meta-class (subclass of the meta-class 'package') and thus an unstructured container of arbitrary elements. This is a pattern that repeats itself over and over in a model: through packages and the meta-class hierarchy of e.g. the 'classifier' meta-class (the super-type of various types such as 'association', 'interface', or 'component'), a dearth of objects of mixed types can appear at arbitrary nesting levels in a model. The story-telling approach gives a user a guideline that allows him to create a structured model, containing the necessary artifacts, in a structured manner. The tool also enforces the visibility of artifacts, for example, visible operations in reactive specifications are limited to those of the containing component.

Operations on the model underlie certain constraints: not every operation may be valid at every time, so the modeler restricts the set of possible actions. Of course, such a restriction should provide feedback to the user, and has to be carefully offset against letting the user do complex editing tasks: if we would simply restrict the user to actions that produce a *valid* model, at least at the moment, we could not guarantee that indeed in every situation she is able to achieve the desired resulting model. Eventually, there will always arise the need for disruptive changes to the model that cannot be consolidated in a single step.

Checking the validity of a particular modeling step can be non-trivial. For example, while most of the refinement steps only have a syntactical criterion which decides whether they are enabled or not, some refinement steps indeed may require a correctness proof. As we expect such a proof to require assistance from a user, like in a theorem prover, we apply the refinement, and record the necessary proof obligation. It can then be discharged by the user (possibly with the help of an expert) without holding up the development. Naturally, if such an obligation cannot be discharged, it will still be necessary to revisit the editing step that created it.

Based on the semantics of the model, additional static checks are necessary: for example, there is a certain relation between the trace language implied by the state machine and the sequence diagram of a component protocol. The tool verifies the consistency in the model validation phase. This is a purely static matter, very much like type checking.

In the following, we describe the different main modeling phases, their artifacts, and principal editing steps and validation.

2 Initial Requirements Model

The rCOS methodology considers the development of component-based reactive software systems from use cases. This corresponds to systems that interact with their environment through a set of well-defined methods invocations. Data types are given as either primitive types, or modeled as object-oriented classes with single inheritance. The emphasis clearly lies on the modeling of data, and control flow. Algorithmic concern only plays a subordinate role, as the systems we are aiming at exhibit structured data and behaviour, unlike, for example, a compiler, or an image processing algorithm. Naturally there is also computation in the systems, but it is usually encapsulated within a method, and as such only one artifact besides many others in the entire specification.

Top-down development of use cases

In the first phase, understanding and analysis of the problem domain is necessary. Assuming that the textual problem description is more or less complete, each use case is created separately in the model. Relationships between the participants (actors) in a use case, and between use cases can be documented in the use case diagrams. However, use cases do not directly influence particular aspects of the model. They are mostly used to group data types and functionality, and indicate their origin in the original problem description.

Use cases represent specific work flows within the software application. For example, in a classical library reservation system, borrowing or returning a book, including the necessary data entry, are modeled as separate use cases. In the CoCoME case study for a supermarket cashdesk system, where we previously applied our methodology by hand [2], buying items, paying for the purchase, and reporting are separate use cases.

In our top-down process, each use case will be realized by a component. Therefor, the editor directly allows navigation to the component from the use case. In the initial stage, since there is no component reuse yet, the component is quite trivial and consists only of a single provided interface. The user then defines the signature of the component interface. For this, she may also add the required data types to the model. Primitive types such as natural numbers, integers, or the 'Object' superclass for other classes are provided by the rCOS UML profile.

Reactive behaviour

The user then also encodes the so-called *interaction protocol* of a component into the model: the protocol describes how the component is to be used through synchronized method invocation of the operations in the component's contract interface. This can be achieved by adding a UML interaction and/or state machine to the contract interface. Again, the modeler guides the user through context menus to the corresponding sub-diagram editor, takes care of structuring

the model adequately (and maintaining the necessary rCOS metadata through stereotypes), and provides visual feedback as to the completeness of the model.

In both editors, the editing options of the sequence diagram for the interaction and the state machine are restricted through the modeler to the relevant subset of the UML. The initial sequence diagram is restricted to two lifelines, where one lifeline represents the environment (the actor from the use case), and the second one the component. Message calls are only initiated from the environment to the component. Only in the later stages (see Sec. 3) this diagram will be filled with more detail.

Both diagrams are created with only the operations from the contract interface as message/transition labels. Additionally, we allow a limited amount of data in the form of attributes of primitive type on the contract interface for use in conditionals and guards. As both diagrams talk about the protocol of the same component from two different points of view, some consistency checking of the implied trace languages is in order that we discuss in Sec. 4.

Object-oriented relational calculus

Probably of most interest to learners and practitioners of software engineering is the relational calculus that rCOS uses to specify the functionality, or business logic, of the methods.

Firstly, the user designs the necessary associations between the principal types (classes) in a class diagram. In line with object-oriented analysis and design, attributes of classes are only used for primitive types at this stage. The operations that she created for the contract interface of the component have already been automatically added to the principal designated controller class of the component (the class that realizes the functionality of the component).

Each operation is given as relational rCOS design, which can take the form of statements and pre/postconditions. For this, rCOS builds on Hoare and He's Unifying Theories of Programming (UTP) [7]. In UTP, a *sequential program* (but possibly nondeterministic) is represented by a *design* $D = (\alpha, P)$, where

- α denotes the set of state variables (called observables). Each state variable comes in an unprimed and a primed version, denoting respectively a pre- and a post-state value. The set includes program variables, e.g. x, x', and a designated Boolean variable, ok, ok', that denotes termination or stability of the program.
- P is of the form $p(x) \vdash R(x, x')$, called the functionality specification of the program. Its semantics is defined as $(ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x'))$, meaning that if the program is activated in a stable state, ok, where *precondition* $p(x)$ holds, the execution will terminate, ok', in a state where postcondition R holds.

In UTP, the *refinement partial order* $\sqsubseteq$ among designs is defined such that $D_1 \sqsubseteq D_2$ if they have the same alphabet, say $\{x, x'\}$, and $\forall x, x' \cdot (P_2 \Rightarrow P_1)$ holds, where P_1 and P_2 are the functionality specification of D_1 and D_2, respectively.

It is proven that with this order the set of designs forms a complete lattice, and *true* is the *least* (*worst*) element of the lattice. Furthermore, this lattice is closed under the classical programming constructs:

- *sequential composition*, $D_1; D_2$,
- *conditional choice*, $D_t \lhd g(x) \rhd D_f$, where g is a predicate, and D_t is selected when g evaluates to true, and D_f is selected when g evaluates to false.
- *nondeterministic choice*, $D_1 \vee D_2$, and
- *least fixed point of iterations*, $\mu x.D$.

All these constructs are *monotonic operations* on the lattice of designs. Refinement between designs is naturally defined as logical implications. These fundamental mathematical properties ensure that the domain of designs is a proper semantic domain for sequential programming languages. For a design, we define its *weakest precondition* for a given post condition q: $\mathbf{wp}(p \vdash R, q) \mathrel{\widehat{=}} p \wedge \neg(R; \neg q)$ where the meaning of composing relations by ";" is the same as in UTP, $q_1; q_2 \mathrel{\widehat{=}} \exists v_0 \cdot (p_1[v_0/x'] \wedge p_2[v_0/x])$. A detailed discussion of rCOS designs, and the extension to reactive object-oriented programs can be found in [6].

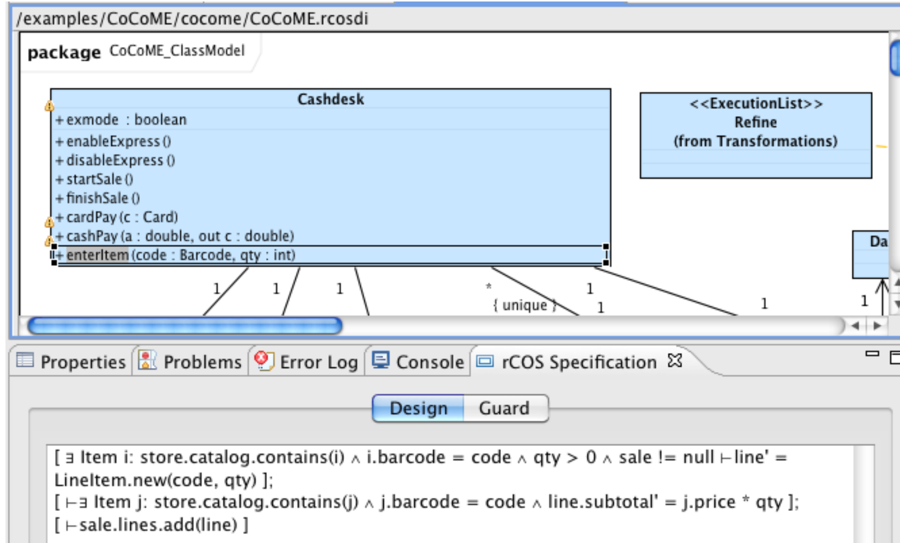

Fig. 2. Relational design in the rCOS modeler

Classes may be given an *invariant*, which is given as a boolean rCOS expression. This invariant is used in code generation, and may be used by reasoning frameworks, such as the one we intend to use to check refinement (see Sec. 3). Fig. 2 shows an rCOS specification for an operation in the UML modeler.

3 Refinement through transformations

The result of the modeling process is obviously a model. It has to be consistent and complete, in the sense that it captures the users intention. But just like neither the structure nor the completeness of a UML model are prescribed, there is no prescribed execution semantics of a particular model. Of course the UML superstructure definition [15] gives a natural language semantics to the concept, and the Executable UML [13] approach focuses on a well-defined subset of the UML. Nevertheless, a concrete implementation language for operations remains beyond its scope.

As our goal is an executable system in the form of code for a modern object-oriented programming language like Java or C++, there is still a gap between the relational specification, and the executable code. Relational designs in rCOS are not executable *per se*. Only the executable fragment of rCOS (no quantification, no pre/post conditions, . . .) has this property. Accordingly, we must somehow get from the relational specification to the executable subset in a semi-automated, correct-by-construction manner.

Furthermore, the structure of the initial requirements model is still too coarse for a concrete object-oriented language: for example, the associations between classes have to mapped to fields in the programming language. Also, from looking at the designs we can see that the relational specification is too dense (only operations in the component controller class), and does not take into account best practices of object-oriented programming, with regard to things like encapsulation.

Of course, this could be seen as a "mistake" in the design of the requirements model. Why not directly define the requirements model in a manner that is closer to object-oriented programming? But that would be missing the point of MDA/MDD: we would then eventually hard-code programming language-specific idioms into the model, making it more difficult to switch to a different target language. More importantly, it would also confuse the artifacts derived from the textual requirements with their implementation. This makes it more difficult to track a particular entity in the model to its source in the requirements document.

Instead, we favour here a transformation approach: a "better" model (in the sense of executability) is derived from the initial requirements model through correct-by-construction refinement steps, which are applied to the input model. We can thus automate the transition from the requirements model to a more programming-oriented model. This also gives us the advantage of revising requirements, and coming up with a semi-automated way of propagating the changes through a chain of refinement steps. In this way, hopefully (depending on the impact of the requirements change) most of the refinement steps still apply and transform the model to its more object-oriented representation, and those steps that no longer apply due to the modification require further editing by the user. With that, we leave the requirements modeling stage, and enter the object-oriented design phase.

In the following, we will discuss certain particular refinement steps that the user applies to the model. For a general description of such a transformation framework, see our work in [17]. Here, we focus on structural refinement to enhance encapsulation, and relational refinements that turn predicates into executable, algorithmic solutions.

Object-oriented refinement

The principal starting point for refinements is an operation. Usually, the functionality specification of an operation makes use of navigation paths through the class model. That is, expressions in the specification traverse the associations (or attributes), starting from the current class. According to object-oriented design laws (see e.g. [5]), a method should not use navigation paths to manipulate elements "owned" by other classes, especially not repeatedly. This makes a class brittle with respect to changes to other classes of the system: a modification in one class may require subsequent changes in other classes. Encapsulating behaviour for example in so-called setters and getters alleviates this, and reduces the number of necessary followup changes through the model.

To apply such a refinement step, the tool offers the Expert Pattern transformation [3, 11], which formalizes this particular change to a program. While we have implemented an automated application of the setters and getters, we envision a more general, interactive usage of this pattern: in the specification of the operation, the user selects a consistent fragment. In this fragment, a number of navigation expressions occur. The tool can calculate the possible classes that the selected fragment can be delegated to, and allows the user to pick the target. The fragment can then be transformed into a single method call, which will be created with the appropriate specification in the selected target class. Any necessary parameters, that is, variables not local to the fragment, are passed as parameters to the new method. For more complex refactorings additional checks will be necessary: just because a navigating expression occurs syntactically multiple times in a fragment, does not mean that at runtime every expression will indeed refer to the same object due to side effects in imperative languages such as ours. This is somewhat alleviated in the tool due to an analysis whether a method is side-effect free, a so-called *query*.

This delegation is then reflected in an updated sequence diagram, where a new lifeline for the target object is created if necessary, and a new message is inserted at the appropriate position.

As the initial requirements model has been created with only methods for the component controller class, we now see that the application of the Expert Pattern is exactly the source of the further methods in the other classes.

Components and their composition

For the refined component model, in the sequence diagram of a use case the *persistent objects* are identified after application of the expert pattern. The objects are then grouped by the user to become part of the desired components:

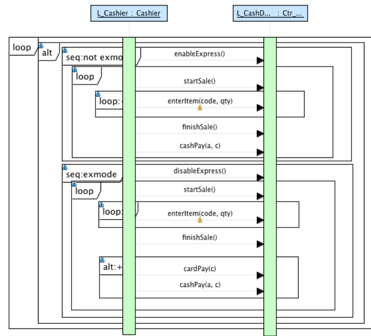

Fig. 3. Initial sequence diagram

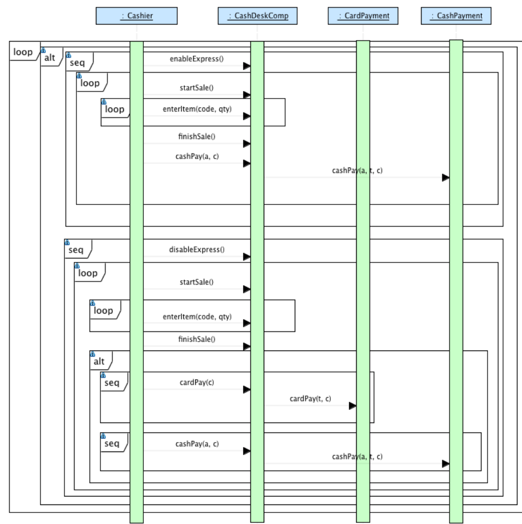

Fig. 4. Refined sequence diagram

we introduce (if the target component does not already exist in the diagram) a new component with its lifeline, collapse the lifelines of objects in the same group onto the new lifeline, and internalize the messages. Only the messages from other components to the newly formed entity remain. Fig. 3 and Fig. 4 show the initial sequence diagram for our case study, and after the refinement (we apologise for the generally poor readability of sequence diagrams in print).

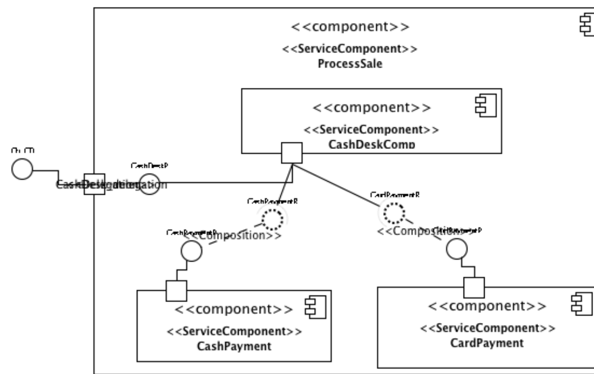

Fig. 5. Component diagram

This process also indirectly influences the component diagram, where the previous component is extended through composition with the newly created

components and their respective interfaces. Fig. 5 shows such a refined component; we elide a graphics showing the trivial diagram of a single component with a single interface for the initial component.

Relational to executable specifications

After applying the Expert Pattern to delegate responsibility, we have transformed the program into an equivalent program adhering to object-oriented design principles. But the model is not executable yet: there may still be relational fragments contained in the specifications. Only some of those can be automatically turned into the executable subset of rCOS. For example, the precondition

$$\exists\ \textsc{Item i: store.catalog.contains(i)} \wedge \textsc{i.barcode} = \textsc{code} \wedge \textsc{qty} > 0 \wedge \textsc{sale} \neq \mathbf{null}$$

can be automatically rewritten into a `for`-loop. We have automated this transformation, and a few other transformations that split a complex design into the sequential composition of simpler designs, so that other transformations can match their input more easily based on a single normal form.

Of course it is not feasible to translate every design into an executable specification: especially when quantification is involved, the user has to give an algorithmic solution to his design. Since the design is refined manually, it is possible that mistakes creep in, where the user actually failed to provide a *correct refinement*. Again this is where tool support should come in, and by tracking refinement steps through our transformation framework, we have the added benefit of the generated proof obligation for this step, which serves as documentation and can be discharged by the user through a theorem prover at a later stage [12].

Code generation

After finally all non-executable constructs in the model have been refined (the tool indicates whether there is still work to be done), the model can then be used for code generation. We have established the principal mapping from rCOS to the Java programming language, and with the help of a few annotations (mostly on the libraries supplied by the tool), we can generate the code automatically. The code generator can easily be adapted to generate code for other object-oriented languages. Currently, the tool can only generate monolithic, not distributed, programs, and does not consider component deployment.

4 Verification

Apart from the static checking of the model on the UML level (completeness and consistency, mostly given in the Object Constraint Language OCL [14, 18]), and type checking of designs, we use two further verification techniques to ensure the correctness of the construction process.

Firstly, as already alluded to in Sec. 2, we ensure consistency of the dynamic behaviour descriptions by transforming both the sequence diagram and the state machine into the process algebra CSP, the input language of the model checker

FDR2, together with appropriate assertions on trace language inclusion. (The actual check has to be run manually in the model checker, our tool integration is not that far yet.) As the sequence diagram specifies the external behaviour as seen from the customer, each possible trace must be accepted by the state machine. We also use the route through CSP to check compatibility of component protocols, and so-called failure-divergence trace refinement. For a full treatment of reactive behaviour and its semantics in the rCOS methodology, see [4].

Secondly, apart from model checker, we need to ensure that all refinement steps are correct. Although it would in principle be possible to require a refinement proof for the entire program after each transformation, clearly this cannot be the solution, since program equivalence is a much too coarse-grained concept. Instead, we expect the user to mostly use transformations that have been proved to produce correct-by-construction refinements. Once such a rule has been proved as correct in a general form, no further verification is necessary. However, in general, rules will have semantic restrictions on their applicability, so it may be that the user has to provide a proof that the enabling condition of a transformation rule indeed holds. So while it will not be possible to avoid theorem proving entirely, we are confident that the required conditions can be much easier proved than program equivalence. The interested reader is invited to refer to [12] for a discussion.

5 Summary

The tool is far from finished. Not all parts of the system have been honed to perfection, as this takes considerable resources: apart from work on the formal side, uncounted hours have been spent on GUI programming, although our adaptation of the TOPCASED platform [16] allows us to work on a higher level of abstraction with the model. But we are confident that our work is a proof of concept of the feasibility of a top-down approach to model-driven or model-based software engineering. Of course it is illusional to expect our tool to compete with the likes of IBM's Rational product, the NetBeans IDE, or other computer-aided software engineering tools.

Our tool allows for a plethora of future extensions in the various directions (transformations, refinement, reasoning support). One such extension is the testing framework for components designed, or reverse-engineered, using our methodology [10]. The next goal will be a coherent set of lecture notes that illustrates software engineering concepts, and guides students to create their own models with the tool. We hope that the additional aspects of *formal* modeling and development, and testing and verification, will increase the perception of the importance of those tasks, that commonly disappear behind the programming tasks.

The tool is available as an Eclipse Rich Client from http://rcos.iist.unu.edu, together with a set of example files.

References

1. F. Arbab and M. Sirjani, editors. *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers*, volume 5961 of *Lecture Notes in Computer Science*. Springer, 2010.
2. Z. Chen, A. H. Hannousse, D. V. Hung, I. Knoll, X. Li, Y. Liu, Z. Liu, Q. Nan, J. C. Okika, A. P. Ravn, V. Stolz, L. Yang, and N. Zhan. Modelling with relational calculus of object and component systems–rCOS. In A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, chapter 3. Springer, Aug. 2008.
3. Z. Chen, Z. Liu, V. Stolz, L. Yang, and A. P. Ravn. A refinement driven component-based design. In *12th Intl. Conf. on Engineering of Complex Computer Systems (ICECCS 2007)*, pages 277–289. IEEE Computer Society, July 2007.
4. Z. Chen, C. Morisset, and V. Stolz. Specification and validation of behavioural protocols in the rCOS modeler. In Arbab and Sirjani [1], pages 387–401.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
6. J. He, X. Li, and Z. Liu. A theory of reactive components. *Electr. Notes Theor. Comput. Sci.*, 160:173–195, 2006.
7. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
8. P. Kruchten. *The Rational Unified Process—An Introduction*. Addison-Wesly, 2000.
9. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, 3rd edition, 2005.
10. B. Lei, X. Li, Z. Liu, C. Morisset, and V. Stolz. Robustness testing for software components. Technical Report 423, UNU-IIST, 2009.
11. Z. Liu, C. Morisset, and V. Stolz. rCOS: theory and tools for component-based model driven development. In Arbab and Sirjani [1], pages 62–80. Keynote, UNU-IIST TR 406.
12. Z. Liu, C. Morisset, and S. Wang. Isabelle proof obligations for graph-based refinement of rCOS programs. Technical Report 430, UNU-IIST, May 2010.
13. S. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture*. Addison Wesley, 2002.
14. Object Management Group. Object constraint language, version 2.0, May 2006.
15. Object Management Group. Unified Modeling Language: Superstructure, version 2.3, May 2010. http://www.omg.org/spec/UML/2.3/Superstructure.
16. N. Pontisso and D. Chemouil. TOPCASED Combining formal methods with model-driven engineering. In *ASE'06: Proc. of the 21st IEEE/ACM Intl. Conf. on Automated Software Engineering*, pages 359–360, Washington, DC, USA, 2006. IEEE Computer Society.
17. V. Stolz. An integrated multi-view model evolution framework. *Innovations in Systems and Software Engineering*, 6(1–2):13–20, Dec. 2009.
18. J. Warmer and A. Kleppe. *The Object Constraint Language: precise modeling with UML*. Addison-Wesley, 1999.

Erweiterung der Containerdarstellung von Listen mittels Relationen
—
Proving Properties About Functions on Lists Involving Element Tests

Daniel Seidel* and Janis Voigtländer

Rheinische Friedrich-Wilhelms-Universität Bonn, Germany
{ds,jv}@iai.uni-bonn.de

Abstract. Bundy and Richardson (1999, LPAR) developed a method for reasoning about functions manipulating lists which is based on separating shape from content, and then exploiting a mathematically convenient representation for expressing shape-only manipulations. Later, Prince et al. (2008, FLOPS) extended the technique to other data structures, and gave it a more formal basis via the theory of containers. All these results are restricted to fully polymorphic functions. For example, functions using equality tests on list elements are out of reach. We remedy this situation by developing new abstractions and representations for less polymorphic functions. In Haskell speak, we extend the earlier approach to be applicable in the presence of (certain) type class constraints.

1 Introduction

Abstraction is a useful strategy to get a clear view on the things that matter. Regarding proofs about program equivalences it is beneficial to have an abstract representation of data structures and functions, holding exactly the information necessary for the intended reasoning, and in an easily accessible form. For lists, Bundy and Richardson (1999) introduced a higher-order formulation in which a list is viewed as a pair (n, f) where n is a natural number representing the length of the list, i.e. its shape, and f is a content function taking each position in the list to the element stored there. Bundy and Richardson's motivation was ease of reasoning about such representations. In a more precise and more general form, the idea later recurred as reasoning via *container representations* (Abbott et al. 2003; Prince et al. 2008).

The usefulness of the abstraction from the actual elements stored in a list is made apparent by the fact that certain *container morphisms*, taking a list (in this case) to another one, do not touch the image of f. An example for such a container morphism is the function $reverse^c$, the container version of the usual

* This author was supported by the DFG under grant VO 1512/1-1.

function reversing a list. The application of this container morphism is given as follows:

$$reverse^c\ (n, f) = (n, \lambda i \to f\ (n - i - 1))$$

In general, a morphism shuffles positions and can alter the length of the list, remove elements, duplicate others. It cannot touch the elements.

The advantage of the container representation, which led Bundy and Richardson to using that representation, is that proofs about programs expressible as the composition of container morphisms become (simple) arithmetic proofs. For example, the proof that reversing a (finite) list twice is the identity is obtained very easily as follows:

$$\begin{aligned} reverse^c\ (reverse^c\ (n, f)) &= reverse^c\ (n, \lambda i \to f\ (n - i - 1)) \\ &= (n, \lambda i \to f\ (n - (n - i - 1) - 1)) \\ &= (n, \lambda i \to f\ i) \\ &= (n, f) \end{aligned}$$

Prince et al. (2008) use, from (Abbott et al. 2003, 2005), that container morphisms correspond to parametrically polymorphic functions (or, *natural transformations*). Such polymorphic functions act independently of the concrete input type and hence, of necessity, independently of concrete elements of a type. For example, a fully polymorphic function g from lists to lists, expressed via the type $\mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$, maps for every type τ input lists of type $\mathcal{L}(\tau)$ to output lists of type $\mathcal{L}(\tau)$ without using any specifics of the type τ.

What both, Bundy and Richardson (1999) and Prince et al. (2008), fail to do is to reason about functions that are not fully polymorphic. In both papers, there is discussion of treating a function *member* that checks whether a given value is an element of a given list. In both cases, the outcome of the discussion is that the proposed reasoning method is not effective for this function. Similarly, reasoning would not work for the function *nub* that eliminates duplicates from a list. While Bundy and Richardson only identified the problematic case, and Prince et al. went a step further by observing that the problem can be explained by a lack of polymorphism, we do provide a solution. In retrospect, at least the basic idea behind our solution may seem obvious: if a function is not polymorphic enough, then start by expressing exactly how it loses its polymorphism, and how much polymorphism is left nevertheless. In Haskell, the appropriate formal concept is available via the type class mechanism (Wadler and Blott 1989). For example, the already mentioned function *nub* can naturally be given the type $Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$. Of course, there is no reason at all to expect that this corresponds to a standard container morphism, because those where shown to be isomorphic to functions of type $\mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$. But we can investigate refined notions of container representations and container morphisms, so that effective reasoning in the spirit of the earlier method becomes possible again.

Our approach can be summarised as follows. We build a container representation where functions of type $Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ can be represented as container morphisms. Clearly, they still have to abstain from inspecting the concrete list elements arbitrarily. But they can use information about equivalence

checks between elements. This requires an adjustment of the container notions, for now an equivalence relation must be accessible in some form. Indeed, instead of checking for equivalence of concrete elements, we use an equivalence relation on the *positions* to represent the relation on the elements. Of course, the approach is not limited to the type class *Eq*. In a similar way, container extensions, container morphisms, and the reasoning method could in principle be extended to handle other type classes.

Our main proof tool are *free theorems* (Wadler 1989). Such theorems are statements about functions only dependent on the function type, relying on a formalisation of parametricity (Reynolds 1983). Typically, they are relational statements that can be specialised to functions and then provide a proof for simple program transformations. For example, the free theorem for the type $\mathcal{L}(\alpha) \to \mathbf{Nat}$ states that for every function $f :: \tau_1 \to \tau_2$ with τ_1 and τ_2 arbitrary, every function $g :: \mathcal{L}(\alpha) \to \mathbf{Nat}$, and every list $xs :: \mathcal{L}(\tau_1)$ we have $g\ (map\ f\ xs) = g\ xs$, where *map* is the usual function that applies its argument function to each element in its input list. Since free theorems are available for free, i.e. can be automatically generated (`http://www-ps.iai.uni-bonn.de/ft/`), we will use them as given, without considering further background here.

2 Repetition of the Container Perspective on Lists

Let us first clarify some notations. The set of natural numbers is denoted by **Nat**. Depending on the context, a natural number n represents either the number $n \in \mathbf{Nat}$ or the set of natural numbers $\{0, \ldots, n-1\}$. Furthermore, the type constructor for lists, already used above, is defined by

$$\mathcal{L}(\tau) = \{[x_0, \ldots, x_{n-1}] \mid n \in \mathbf{Nat}, \forall i \in n.\, x_i :: \tau\}$$

Lists can alternatively be defined as container extensions, meaning by a shape (the length) and a content function (mapping each position to its entry). An appropriate definition (without using container terminology) was already introduced by Bundy and Richardson (1999). We restate it here by defining the set $\mathcal{C}(\tau)$ of list container extensions of type τ as

$$\mathcal{C}(\tau) = \{(n, f) \mid n \in \mathbf{Nat}, f : \mathbf{Nat} \to \tau\}$$

where the f need not be totally defined, i.e., can be a partial function. However, in every container extension (n, f), one requires f to be defined at least for all natural numbers less than n. It would be possible to enforce this by using dependent types, and that is what Prince et al. (2008) do, but we obstain from doing so.

A *container morphism* then is an element of the set

$$\{(s_n, P_n)_{n \in \mathbf{Nat}} \mid s_n :: \mathbf{Nat} \wedge P_n :: \mathbf{Nat} \to \mathbf{Nat}\}$$

For each container morphism and each $n \in \mathbf{Nat}$ we allow P_n in (s_n, P_n) to be a partial function. But for well-definedness, one requires that a family $(s_n, P_n)_{n \in \mathbf{Nat}}$

is a container morphism only if for every n and $i \in s_n$ we have $(P_n\ i) \in n$. We often abbreviate $(s_n, P_n)_{n\in\mathbf{Nat}}$ as (s, P). The application of a container morphism to a container extension is given by

$$(s, P)\ (n, f) = (s_n, f \circ P_n)$$

Here are some container morphisms that intuitively correspond to well-known functions of type $\mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$:

$$\begin{aligned} reverse^c &= (n, \lambda i \to n - i - 1)_{n\in\mathbf{Nat}} \\ init^c &= (n - 1, id)_{n\in\mathbf{Nat}} \\ tail^c &= (n - 1, \lambda i \to i + 1)_{n\in\mathbf{Nat}} \end{aligned}$$

3 Refining the Container-Related Notions

Considering functions like *nub*, removing all duplicates from a list, or *sort*, sorting a list's elements, it is clear that they are not fully polymorphic in their list element type. The functions require the elements of the input list to have an equivalence test or an order defined on. Our aim now is to appropriately adapt the notions of container extension and container morphism to get equally useful results for functions of types $Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ and $Ord\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ as the earlier works (Bundy and Richardson 1999; Prince et al. 2008) provide for functions of type $\mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$. Here, we focus on the type class *Eq*.

To capture what happens if elements in a list are checkable for equivalence, the container notions have to be adjusted. We use $\mathcal{E}(M)$ to denote the class of all equivalence relations over a set M. For simplicity of notation, we regard each equivalence relation $\cong$ on a subset of **Nat** as the equivalence relation $\cong \cup\, id_{\mathbf{Nat}}$ on **Nat**. An *Eq*-container extension is not only dependent on a type, it also depends on an equivalence relation defined on the type.

Definition 1. *Let τ be some type and $\cong_\tau$ an equivalence relation on it. An Eq-*container extension *of type τ with respect to $\cong_\tau$ is a triple $(n, \cong, f)$ with $n \in \mathbf{Nat}$, $\cong\, \in \mathcal{E}(\mathbf{Nat})$, and $f : \mathbf{Nat} \to \tau$ a partial function such that*

$$\forall i, j \in n.\ i \cong j \Leftrightarrow f\ i \cong_\tau f\ j$$

or, equivalently,[1]

$$(\ker_{\cong_\tau} f|_n) = (\cong \cap\, (n \times n)) \tag{1}$$

The set of all such container extensions is denoted by $\mathcal{C}^{Eq}(\tau, \cong_\tau)$.

Note that for every *Eq*-container extension $(n, \cong, f)$, the function f has to be defined on every natural number less than n to satisfy condition (1). Note also that we use f as a function from list positions into the type, where condition (1)

[1] The kernel of a function over an equivalence relation is defined as $(\ker_{\cong} f) = \{(i, j) \mid (f\ i) \cong (f\ j)\}$. The notation $f|_n$ means the restriction of function f to the domain n.

guarantees that the function preserves equivalence. We could have been tempted to instead define f as a function from *equivalence classes* of positions, with respect to $\cong$, into τ, rather than from the positions themselves. While these choices may appear to be interchangeable, there is actually a crucial difference. With our choice we can distinguish elements that are equivalent with respect to $\cong_\tau$, but not equal. For example, consider the list $[1,3] :: \mathcal{L}(\mathbf{Nat})$ and assume that the equivalence relation defined on **Nat** were equivalence modulo 2. Then a container representation working with a function from equivalence classes of positions would, at length two, only be able to represent lists with two equal elements ($[1,1]$, $[2,2]$, $[3,3]$, $\ldots$) and lists with elements of different parity ($[1,2]$, $[2,1]$, $[1,4]$, $\ldots$), but not the list $[1,3]$ as distinguishable from $[1,1]$ and $[3,3]$. One might be willing to accept this limited expressiveness, as indeed when the equivalence provided by the type class instance for **Nat** is equivalence modulo 2, then all of $[1,3]$, $[1,1]$, and $[3,1]$ ought to be considered equivalent with respect to the inferred type class instance for $\mathcal{L}(\mathbf{Nat})$. But after all, equivalent with respect to a type class instance is not the same as semantically equal, and we want to keep that distinction in our reasoning. For example, we want to still be able to observe that applying (the container morphism corresponding to) *reverse*[2] to $[1,3]$ gives $[3,1]$, and not $[1,1]$ or $[3,3]$.

After having made this important decision, we can set up a pair of functions for going back and forth between *Eq*-container extensions and lists.

Lemma 1. *For each type τ and $\cong_\tau$ an equivalence relation on τ, the instantiations of the functions $\Box^{Eq}$ and $(\Box^{Eq})^{-1}$ defined as*

$$
\begin{array}{ll}
\Box^{Eq} & :: Eq\ \alpha \Rightarrow \mathcal{C}^{Eq}(\alpha, \cong_\alpha) \to \mathcal{L}(\alpha) \\
\Box^{Eq}\ (n, _, f) & = map\ f\ [0, \ldots, n-1] \\
(\Box^{Eq})^{-1} & :: Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{C}^{Eq}(\alpha, \cong_\alpha) \\
(\Box^{Eq})^{-1}\ xs & = (length\ xs, \ker_{\cong_\alpha}(xs\ !!), xs\ !!)
\end{array}
$$

satisfy the following three properties:

1. $(\Box^{Eq} \circ (\Box^{Eq})^{-1}) = id_{\mathcal{L}(\tau)}$
2. $((\Box^{Eq})^{-1} \circ \Box^{Eq}) \subseteq \equiv_{\mathcal{C}^{Eq}(\tau,\cong_\tau)}$, *where* $\equiv_{\mathcal{C}^{Eq}(\tau,\cong_\tau)} = \{((n,E,f),(n,E',f')) \mid \forall i \in n.\ f\ i = f'\ i\}$
3. $\forall (n,E,f),(n',E',f') \in \mathcal{C}^{Eq}(\tau,\cong_\tau).$
   $(n,E,f) \equiv_{\mathcal{C}^{Eq}(\tau,\cong_\tau)} (n',E',f')$ *iff* $\Box^{Eq}\ (n,E,f) = \Box^{Eq}\ (n',E',f')$

Now, appropriate morphisms between *Eq*-container extensions, and their application, are defined as follows.

Definition 2. *An Eq-*container morphism *(s,P) is a family of pairs $(s_{n,\cong}, P_{n,\cong})_{n \in \mathbf{Nat}, \cong \in \mathcal{E}(\mathbf{Nat})}$ such that $s_{n,\cong} :: \mathbf{Nat}$ and $P_{n,\cong} :: \mathbf{Nat} \to \mathbf{Nat}$ with $(P_{n,\cong}\ i) \in n$ for every $i \in s_{n,\cong}$.*

[2] Clearly, we can regard every function of type $\mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ as a function of type $Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$.

Definition 3. *Let $(n, \cong, f)$ be an Eq-container extension and (s, P) an Eq-container morphism. The application of (s, P) to $(n, \cong, f)$ is defined as*

$$(s, P)\ (n, \cong, f) = (s_{n,\cong}, \ker_{\cong} P_{n,\cong}, f \circ P_{n,\cong})$$

The following lemma states the well-definedness of these notions.

Lemma 2. *Let τ be a type and $\cong_\tau$ an equivalence relation on it. Let $c \in \mathcal{C}^{Eq}(\tau, \cong_\tau)$ and let m be an Eq-container morphism. Then we have $(m\ c) \in \mathcal{C}^{Eq}(\tau, \cong_\tau)$.*

Comparing the definitions of morphisms on ordinary container extensions and on *Eq*-container extensions, we can easily translate the former ones into the latter ones.

Note 1. Every (ordinary) container morphism $(s_n, P_n)_{n \in \mathbf{Nat}}$ can be viewed as the *Eq*-container morphism $(s_n, P_n)_{n \in \mathbf{Nat}, \cong \in \mathcal{E}(\mathbf{Nat})}$.

To verify that our definitions of *Eq*-container extensions and *Eq*-container morphisms are useful when reasoning about functions of type $Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$, we need the following results.

Theorem 1. *For every function $g :: Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ there exists an Eq-container morphism (s, P) such that*

$$g \circ \Box^{Eq} = \Box^{Eq} \circ (s, P)$$

Proof. Let $g :: Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$. Then the free theorem for g's type tells us that $map\ h\ (g\ l) = g\ (map\ h\ l)$ for every choice of types τ_1, τ_2 that are instances of *Eq* (with $\cong_{\tau_1}, \cong_{\tau_2}$ being the concrete equivalence relations provided), function $h :: \tau_1 \to \tau_2$, and list $l :: \mathcal{L}(\tau_1)$, provided that $(\ker_{\cong_{\tau_2}} h) = \cong_{\tau_1}$.

Now, let $(n, \cong, f) \in \mathcal{C}^{Eq}(\tau, \cong_\tau)$. By the definition of *Eq*-container extensions we know that the function f satisfies $(\ker_{\cong_\tau} f|_n) = (\cong \cap (n \times n))$. So for $h = f|_n$, $\cong_{\tau_1} = (\cong \cap (n \times n))$, and $\cong_{\tau_2} = \cong_\tau$ we can apply the free theorem above and obtain $map\ f|_n\ (g\ l) = g\ (map\ f|_n\ l)$ for every list $l :: \mathcal{L}(\mathbf{Nat})$. Hence, we can reason as follows:[3]

$$\begin{aligned}
(g_{\cong_\tau} \circ \Box^{Eq})\ (n, \cong, f) &= g_{\cong_\tau}\ (map\ f\ [0, \ldots, n-1]) \\
&= g_{\cong_\tau}\ (map\ f|_n\ [0, \ldots, n-1]) \\
&= map\ f|_n\ (g_{\cong \cap (n \times n)}\ [0, \ldots, n-1]) \\
&= (\Box^{Eq} \circ (\Box^{Eq})^{-1})\ (map\ f|_n\ (g_{\cong \cap (n \times n)}\ [0, \ldots, n-1])) \\
&= \Box^{Eq}\ (length\ (g_{\cong \cap (n \times n)}\ [0, \ldots, n-1]), \\
&\qquad \ker_{\cong_\tau} (f|_n \circ ((g_{\cong \cap (n \times n)}\ [0, \ldots, n-1])\ !!)), \\
&\qquad f|_n \circ ((g_{\cong \cap (n \times n)}\ [0, \ldots, n-1])\ !!)) \\
&= (\Box^{Eq} \circ (s, P))\ (n, \cong, f)
\end{aligned}$$

[3] To highlight the changes of the equivalence relation that g uses, we have throughout indexed each instance of g with the equivalence relation it actually works with.

where we set

$$(s,P) = (\mathit{length}\ (g_{\cong\cap(n\times n)}\ [0,\ldots,n{-}1]), (g_{\cong\cap(n\times n)}\ [0,\ldots,n{-}1])\ !!)_{n\in\mathbf{Nat},\cong\in\mathcal{E}(\mathbf{Nat})}$$

and use

$$f \circ ((g_{\cong\cap(n\times n)}\ [0,\ldots,n-1])\ !!) = f|_n \circ ((g_{\cong\cap(n\times n)}\ [0,\ldots,n-1])\ !!)$$

as well as

$$\ker_{\cong} ((g_{\cong\cap(n\times n)}\ [0,\ldots,n-1])\ !!) = \ker_{\cong_\tau} (f|_n \circ ((g_{\cong\cap(n\times n)}\ [0,\ldots,n-1])\ !!))$$

These two statements used here are true since it is easy to see (for example, using another instantiation of g's free theorem) that $g_{\cong\cap(n\times n)}\ [0,\ldots,n-1]$ contains only elements from 0 to $n-1$ and since, for the second statement, we know that $(\ker_{\cong_\tau}\ f|_n) = (\cong\cap(n\times n))$.

Corollary 1. *For every function* $g :: Eq\ \alpha \Rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ *there exists an Eq-container morphism* (s,P) *such that*

$$g = \Box^{Eq} \circ (s,P) \circ (\Box^{Eq})^{-1}$$

Proof. By Theorem 1 and Lemma 1(1).

Lemma 3. *Let* g, g' *be functions of type* $Eq \Rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$*. Let* $(s,P),(s',P')$ *be Eq-container morphisms such that* $g = \Box^{Eq} \circ (s,P) \circ (\Box^{Eq})^{-1}$ *and* $g' = \Box^{Eq} \circ (s',P') \circ (\Box^{Eq})^{-1}$*. Then we have* $g \circ g' = \Box^{Eq} \circ (s,P) \circ (s',P') \circ (\Box^{Eq})^{-1}$*.*

Proof. By Lemma 1(2–3).

We now have all the formal material required to reason about functions of type $Eq \Rightarrow \mathcal{L}(\alpha) \rightarrow \mathcal{L}(\alpha)$ by instead reasoning about *Eq*-container morphisms. To manifest this with some examples, consider first the following container morphism versions of *nub* and *rmSingles*, where the first of these functions removes duplicates from a list and the second one throws away each element that appears only once in a given list (in both cases, ultimately with respect to an equivalence relation provided via a type class instance for *Eq*, of course):

$\mathit{nub}^c = (s_{n,\cong}, P_{n,\cong})_{n\in\mathbf{Nat},\cong\in\mathcal{E}(\mathbf{Nat})}$
with
$s_{n,\cong} = |n_{/\cong}|$ and
$P_{n,\cong} = \lambda i \rightarrow \min\{j\ :\ |\{[k]_\cong\ :\ k \leq j\}| = i+1\}$

$\mathit{rmSingles}^c = (s_{n,\cong}, P_{n,\cong})_{n\in\mathbf{Nat},\cong\in\mathcal{E}(\mathbf{Nat})}$
with
$s_{n,\cong} = \sum_{e\in n_{/\cong},|e|>1} |e|$ and
$P_{n,\cong} = \lambda i \rightarrow \min\{j\ :\ |\{j' \in \bigcup_{e\in n_{/\cong},|e|>1} e\ :\ j' \leq j\}| = i+1\}$

Note that we use standard notations $n_{/\cong}$ for factorisation with respect to an equivalence relation and $[k]_{\cong}$ for building equivalence classes.

As already noticed, we can view "ordinary" container morphisms as *Eq*-container morphisms as well. For an example, we give the application of $init^c$ to an *Eq*-container extension. As we use them in the following examples of proofs using *Eq*-container morphisms, we show the result of applying nub^c and $rmSingles^c$, in general, as well.

$$\begin{aligned}
init^c\ (n, \cong, f) &= (n-1, \cong, f) \\
nub^c\ (n, \cong, f) &= (|n_{/\cong}|, id, \lambda i \to f\ (\min\{j\ :\ |\{[k]_{\cong}\ :\ k \le j\}| = i+1\})) \\
rmSingles^c(n, \cong, f) &= (\textstyle\sum_{e \in n_{/\cong}, |e|>1} |e|, \\
&\qquad \ker_{\cong}\ (\lambda i \to \min\{j\ :\ |\{j' \in \textstyle\bigcup_{e \in n_{/\cong}, |e|>1} e\ :\ j' \le j\}| \\
&\qquad\qquad = i+1\}), \\
&\qquad \lambda i \to f\ (\min\{j\ :\ |\{j' \in \textstyle\bigcup_{e \in n_{/\cong}, |e|>1} e\ :\ j' \le j\}| \\
&\qquad\qquad = i+1\}))
\end{aligned}$$

Note that we used algebraic simplifications like that ($\ker_{\cong}\ id$) is $\cong$ and that the kernel of an (up to $\cong$) injective function is the identity.

Let us now demonstrate the usefulness of reasoning with our extended container notions, based on three examples.

An example proof with *Eq*-container morphisms. We wish to show that $nub \circ init$ always returns a prefix of the result of just nub. Using our new setup, we can do this by showing that for every *Eq*-container extension c,

$$\mathbf{prefix}\ ((nub^c \circ init^c)\ c)\ (nub^c\ c) \tag{2}$$

holds, where **prefix** is defined by

$$\mathbf{prefix}\ (n_1, \cong_1, f_1)\ (n_2, \cong_2, f_2)\ \Leftrightarrow\ n_1 \le n_2\ \wedge\ \forall i \in n_1.\ f_1\ i = f_2\ i$$

To prove the desired statement, we take an arbitrary *Eq*-container extension $c = (n, \cong, f)$ and first calculate both arguments to **prefix** in (2) above. We get

$$\begin{aligned}
(nub^c \circ init^c)\ c &= (nub^c \circ init^c)\ (n, \cong, f) \\
&= nub^c\ (n-1, \cong, f) \\
&= (|(n-1)_{/\cong}|, id, \lambda i \to f\ (\min\{j\ :\ |\{[k]_{\cong}\ :\ k \le j\}| = i+1\}))
\end{aligned}$$

for the first argument and

$$\begin{aligned}
nub^c\ c &= nub^c\ (n, \cong, f) \\
&= (|n_{/\cong}|, id, \lambda i \to f\ (\min\{j\ :\ |\{[k]_{\cong}\ :\ k \le j\}| = i+1\}))
\end{aligned}$$

for the second one.

To verify the **prefix** property we then have to establish the following statements:

1. $|(n-1)_{/\cong}| \leq |n_{/\cong}|$
2. $\forall i \in |(n-1)_{/\cong}|.\ f\ (\min\{j : |\{[k]_{\cong} : k \leq j\}| = i+1\}) = f\ (\min\{j : |\{[k]_{\cong} : k \leq j\}| = i+1\})$

of which the first is a simple property of factorisation (of a subset, with respect to the same equivalence relation), and of which the second is a syntactic tautology.

Another example proof. We wish to show that $\mathit{rmSingles} \circ \mathit{nub}$ always returns an empty list. Using our new setup, we can do this by showing that for every Eq-container extension c, the container extension $(\mathit{rmSingles}^c \circ \mathit{nub}^c)\ c$ has 0 in its length component. So let $c = (n, \cong, f)$ be an Eq-container extension. Then:

$$\begin{aligned}
&(\mathit{rmSingles}^c \circ \mathit{nub}^c)\ c \\
&= \mathit{rmSingles}^c\ (\mathit{nub}^c\ (n, \cong, f)) \\
&= \mathit{rmSingles}^c\ (|n_{/\cong}|, \mathit{id}, \lambda i \to f\ (\min\{j : |\{[k]_{\cong} : k \leq j\}| = i+1\})) \\
&= (\sum_{e \in |n_{/\cong}|_{/\mathit{id}}, |e|>1} |e|, \cdots, \cdots) \\
&= (0, \cdots, \cdots)
\end{aligned}$$

And yet another example proof. We wish to show that nub is idempotent, i.e. $\mathit{nub} \circ \mathit{nub} = \mathit{nub}$. Using our new setup, we can do this by showing that $\mathit{nub}^c \circ \mathit{nub}^c = \mathit{nub}^c$.[4] So let $c = (n, \cong, f)$ be an Eq-container extension. Then:

$$\begin{aligned}
(\mathit{nub}^c \circ \mathit{nub}^c)\ c &= \mathit{nub}^c\ (\mathit{nub}^c\ (n, \cong, f)) \\
&= \mathit{nub}^c\ (|n_{/\cong}|, \mathit{id}, h) \\
&\quad \text{with } h = \lambda i \to f\ (\min\{j : |\{[k]_{\cong} : k \leq j\}| = i+1\}) \\
&= (||n_{/\cong}|_{/\mathit{id}}|, \mathit{id}, \lambda i \to h\ (\min\{j : |\{[k]_{\mathit{id}} : k \leq j\}| = i+1\})) \\
&= (|n_{/\cong}|, \mathit{id}, h) \\
&= \mathit{nub}^c\ c
\end{aligned}$$

where except for the next-to-last one all steps are simply by applying definitions. That one interesting step is valid by $|m_{/\mathit{id}}| = m$ for every $m \in \mathbf{Nat}$,[5] and by the fact that for every $i \in \mathbf{Nat}$,

$$\begin{aligned}
\min\{j : |\{[k]_{\mathit{id}} : k \leq j\}| = i+1\} &= \min\{j : |\{\{k\} : k \leq j\}| = i+1\} \\
&= \min\{j : |\{\{0\}, \{1\}, \ldots, \{j\}\}| = i+1\} \\
&= i
\end{aligned}$$

[4] Actually, it would suffice to show that for every type τ and Eq-container extension $c \in \mathcal{C}^{\mathit{Eq}}(\tau, \cong_\tau)$, it holds that $(\mathit{nub}^c \circ \mathit{nub}^c)\ c \equiv_{\mathcal{C}^{\mathit{Eq}}(\tau, \cong_\tau)} \mathit{nub}^c\ c$.

[5] Note that our notation overloading is at work here, according to which $m \in \mathbf{Nat}$ can represent the actual number m in one context and the set of numbers $\{0, \ldots, m-1\}$ in another context.

4 Conclusion

We have extended the ellipsis (Bundy and Richardson 1999) or container (Prince et al. 2008) technique for reasoning about functions on lists to the case of the presence of element tests. The key insight is to use, as in the proof of Theorem 1, an extension of free theorems (Reynolds 1983; Wadler 1989) to ad-hoc polymorphism à la type classes (Wadler and Blott 1989). This flavour of free theorems, situated in the middle between full polymorphism and arbitrary type-specific behaviour, has been folklore for a while, but has mostly been ignored in the literature. Recently, awareness of the possibility of this specific kind of type-based reasoning has been increasing (Gibbons and Paterson 2009), and applications are appearing (Voigtländer 2008b, 2009a,b). With the current work, we make a case for furthering such investigations, by providing a further useful application. An obvious goal for future work is to see what needs to be done to make reasoning with our refined container-related notions, as we have performed on examples by hand, more effective and mechanisable. Just as the techniques of Bundy and Richardson (1999) and Prince et al. (2008) have to rely on good proof tactics for arithmetics, our method will have to rely on tactics that additionally take properties of equivalence relations and total preorders into account, and that can exploit algebraic notions like the kernel of a function over a relation, etc.

How about further extensions? Moving from lists to a broader range of data structures is largely orthogonal to taking element tests into account. So while Prince et al. (2008) extend the technique of Bundy and Richardson (1999) by replacing lists with the more general concept of a container extension, and while we have here extended the technique of Bundy and Richardson by replacing fully polymorphic functions on lists (only) with ad-hoc polymorphic functions on lists, a combination of the two extensions should be possible. We do not foresee any major obstacles, except maybe for making the best choice in providing a succinct mathematical representation for separating shape (an enhanced notion thereof, taking equivalence and/or order into account) from content in each situation. The guiding principle is that the abstraction should be chosen in such a way that the shape of the output can be determined solely from the shape of the input, as was done here.

The more challenging direction for extension is to treat other type classes than *Eq* and *Ord*. The framework of free theorems is readily available for other type classes as well, but the step of finding the right abstractions and morphism notions may appear to require new insights for each new setting to consider. For example, while both *Eq* and *Ord* mathematically correspond to relations, or to ways of *observing* elements of an unspecified type, what about type classes that provide ways of *constructing* elements via some operations, say class *Monoid*? Interestingly, very recent work by Bernardy et al. (2010) could shed some light here. For the purpose of testing (not verification), they essentially characterise polymorphic functions in terms of monomorphic inputs, such as characterising a function of type $[\alpha] \rightarrow [\alpha]$ in terms of its action on integer lists of the form $[1, \ldots, n]$. For more complicated types, in particular higher-order ones, they

work from a classification of function arguments (typically themselves functions) into observers and constructors, and describe a methodology for finding fixed types and monomorphic inputs that completely determine a function's behaviour. Via the dictionary translation method, type class constraints lead to precisely such different kinds of function arguments, so there is a good chance for mutual leverage here.[6]

References

M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computational Structures, Proceedings*, volume 2620 of *LNCS*, pages 23–38. Springer-Verlag, 2003.

M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *European Symposium on Programming, Proceedings*, volume 6012 of *LNCS*, pages 125–144. Springer-Verlag, 2010.

A. Bundy and J. Richardson. Proofs about lists using ellipsis. In *Logic for Programming, Artificial Intelligence, and Reasoning, Proceedings*, volume 1705 of *LNCS*, pages 1–12. Springer-Verlag, 1999.

J. Gibbons and R. Paterson. Parametric datatype-genericity. In *Workshop on Generic Programming, Proceedings*, pages 85–93. ACM Press, 2009.

R. Prince, N. Ghani, and C. McBride. Proving properties about lists using containers. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 97–112. Springer-Verlag, 2008.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008a.

J. Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008b.

J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009a.

J. Voigtländer. Free theorems involving type constructor classes. In *International Conference on Functional Programming, Proceedings*, pages 173–184. ACM Press, 2009b.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989. doi: 10.1145/99370.99404.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages, Proceedings*, pages 60–76. ACM Press, 1989.

[6] Indeed, for some specific types, the technique of Bernardy et al. (2010) actually leads to characterisations already used by Voigtländer (2008a, 2009a).

A Functional, Successor List Based Version of Warshall's Algorithm

Rudolf Berghammer

Institut für Informatik
Christian-Albrechts-Universität Kiel
Olshausenstraße 40, 24098 Kiel, Germany
rub@informatik.uni-kiel.de

Abstract. We show how systematically to develop a purely functional version of Warshall's algorithm for computing transitive closures. It bases on an implementation of relations by lists of successor lists. The final version immediately can be implemented in Haskell. The resulting Haskell program has the same runtime complexity as the traditional imperative array-based implementation of Warshall's algorithm.

1 Introduction

The computation of the transitive closure R^+ of a (binary) relation R on a set X has many practical applications. This is mainly due to the fact that, if R is the set of edges of a directed graph $G = (X, R)$, then R^+ relates two vertices $x, y \in X$ of G if and only if y is reachable from x via a nonempty path. Usually the task of computing the relation R^+ is solved by Warshall's algorithm [11]. Its traditional implementation in an imperative programming language is based on a representation of the relation R by a 2-dimensional Boolean array. This leads to a simple and efficient program with three nested loops that needs $O(n^3)$ steps, where m is the cardinality of the carrier set X of R. See, for instance, Algorithm 2.3.8 in [7].

However, in certain cases arrays are unfit for representing relations. In particular this holds if both R and R^+ are of "medium density" or even sparse. Here a representation of relations by lists of successor lists is much more economic w.r.t. the required space. But such a representation derogates the simplicity and efficiency of the imperative implementation of Warshall's algorithm. Arrays are also problematic if another programming paradigm is used. Especially, the method of imperatively updating an array representing the relation is alien to the purely functional programming paradigm which restricts the use of side effects.

In the present paper we show how systematically to obtain a purely functional version of Warshall's algorithm that bases on an implementation of relations by lists of successor lists and also has a cubic runtime. In the first step we develop a functional algorithm for computing R^+ that solely bases on relation algebra and the generation of (relational) vectors via disjoint unions of (relational) points. To obtain from it a version that works on lists of successor sets we then represent

relations on X by functions from X in its powerset 2^X and vectors on X by elements of the powerset 2^X and, after that, the functions by lists over 2^X and the sets by lists over X. Going, finally, from lists over 2^X to lists of lists over X we obtain a version in the functional programming language Haskell.

2 Relation-Algebraic Preliminaries

Following the Z specification language [8] we denote the set (or type) of all (binary) relations with source X and target Y by $[X \leftrightarrow Y]$ and write $R : X \leftrightarrow Y$ instead of $R \in [X \leftrightarrow Y]$. If the sets X and Y are finite, we may consider R as a Boolean matrix with $|X|$ rows and $|Y|$ columns. We assume the reader to be familiar with the basic operations on relations, viz. R^T (transposition), $\overline{R}$ (complement), $R \cup S$ (join), $R \cap S$ (meet), $R;S$ (composition), the predicate indicating $R \subseteq S$ (inclusion) and the special relations O (empty relation), L (universal relation) and I (identity relation). Furthermore, we assume the reader to know the most fundamental laws of relation algebra like $\mathsf{I};R = R$, $R^{\mathsf{T}^\mathsf{T}} = R$, $(R;S)^\mathsf{T} = S^\mathsf{T};R^\mathsf{T}$, $R;(S \cup T) = R;S \cup R;T$, and the following one, in [7] called *Schröder equivalences*.

$$Q;R \subseteq S \iff Q^\mathsf{T};\overline{S} \subseteq \overline{R} \iff \overline{S};R^\mathsf{T} \subseteq \overline{Q} \tag{1}$$

We also will use the relation-algebraic specifications of the following properties: *reflexivity* $\mathsf{I} \subseteq R$, *transitivity* $R;R \subseteq R$, *injectivity* $R;R^\mathsf{T} \subseteq \mathsf{I}$ and *surjectivity* $\mathsf{L};R = \mathsf{L}$. For more details concerning the algebraic treatment of relations we refer to the seminal paper [9] and the textbook [7].

A (relational) *vector* is a relation v which satisfies the equation $v = v;\mathsf{L}$ and a (relational) *point* is an injective and surjective vector. For vectors the targets are irrelevant. We, therefore, consider in the following mostly vectors $v : X \leftrightarrow \mathbf{1}$ with a specific singleton set $\mathbf{1} = \{\bot\}$ as target and omit in such cases the second component $\bot$ in a pair, i.e., write $x \in v$ instead of $(x, \bot) \in v$. Then v *describes* the subset $\{x \in X \mid x \in v\}$ of its source X. If X is finite, a vector of $[X \leftrightarrow \mathbf{1}]$ can be considered as a Boolean matrix with $|X|$ rows and exactly one column, i.e., as a Boolean column vector in the usual sense, and the set it describes is given by the components with entry 1.

In the Boolean matrix model a point of $[X \leftrightarrow \mathbf{1}]$ is a Boolean column vector in which exactly one entry is 1. This means that a point $p : X \leftrightarrow \mathbf{1}$ describes a singleton subset of X, or an element of X if we identify a singleton set $\{x\} \subseteq X$ with the only element $x \in X$ it contains. Later we will use that if p describes $x \in X$, then $(y, z) \in p;p^\mathsf{T}$ is equivalent to $y = x$ and $z = x$.

3 Computing Transitive Closures Using Relation Algebra

Given a relation $R : X \leftrightarrow X$, its *reflexive-transitive closure* $R^* : X \leftrightarrow X$ is defined as the least reflexive and transitive relation that contains R and its *transitive closure* $R^+ : X \leftrightarrow X$ is defined as the least transitive relation that contains R.

It is well-known (cf. [7,1]) that R^* and R^+ also can be specified via least fixed point constructions. The least fixed point of $\tau_R : [X \leftrightarrow X] \to [X \leftrightarrow X]$, where $\tau_R(Q) = \mathsf{I} \cup R;Q$, is R^*, and the least fixed point of $\sigma_R : [X \leftrightarrow X] \to [X \leftrightarrow X]$, where $\sigma_R(Q) = R \cup R;Q$, is R^+. From these specifications we obtain by fixed point considerations (see e.g., again [7,1]) the following equations.

$$\mathsf{O}^* = \mathsf{I} \qquad R^* = \mathsf{I} \cup R^+ \qquad R^+ = R;R^* \qquad (R \cup S)^* = R^*;(S;R^*)^* \tag{2}$$

In [1] the rightmost equation of (2) is called the *star-decomposition* rule. A simple fixed point argument also shows that R is transitive if and only if $R = R^+$.

Now, let $R : X \leftrightarrow X$ be the relation of a finite directed graph $G = (X, R)$ with vertex set $X = \{v_1, \ldots, v_m\}$. The idea behind Warshall's algorithm is to consider for $i, 0 \leq i \leq m$, the subset $X_i := \{v_1, \ldots, v_i\}$ of X and the relation $R_i : X \leftrightarrow X$ such that for all $x, y \in X$ it holds

$$(x,y) \in R_i \iff \begin{cases} \text{there exists a path } (z_1, \ldots, z_k) \text{ from } x \text{ to} \\ y \text{ such that } k > 1 \text{ and } z_2, \ldots, z_{k-1} \in X_i. \end{cases} \tag{3}$$

Then, for all $x, y \in X$ and $i, 1 \leq i \leq m$, it can be shown that $(x,y) \in R_i$ if and only if $(x,y) \in R_{i-1}$ or $(x, v_i) \in R_{i-1}$ and $(v_i, y) \in R_{i-1}$. Since, furthermore, for all $x, y \in X$ it holds $(x,y) \in R_0$ if and only if $(x,y) \in R$ and $(x,y) \in R_m$ if and only if $(x,y) \in R^+$, we are able to compute R^+ as limit of the finite chain

$$R = R_0 \subseteq R_1 \subseteq \ldots \subseteq R_{m-1} \subseteq R_m = R^+.$$

This is exactly what the traditional implementation of Warshall's algorithm realizes by means of a Boolean array.

To express (3) in terms of relation algebra we consider for a given relation $R : X \leftrightarrow X$ a vector $v : X \leftrightarrow \mathbf{1}$ and the partial identity relation $\mathsf{I}_v := \mathsf{I} \cap v;v^\mathsf{T}$ induced by v. If we assume that v describes the subset X_v of X, then a little reflection shows for all $x, y \in X$ that $(x,y) \in R;(\mathsf{I}_v;R)^*$ if and only if there exists a path $(z_1, \ldots, z_k)$ from x to y such that $k > 1$ and $z_2, \ldots, z_{k-1} \in X_v$. This motivates the following definition.

$$\mathit{warsh}(R,v) : X \leftrightarrow X \qquad \mathit{warsh}(R,v) = R;(\mathsf{I}_v;R)^* \tag{4}$$

Using the first and third equation of (2), the definition of *warsh* in (4) implies for the empty vector $\mathsf{O} : X \leftrightarrow \mathbf{1}$ that

$$\mathit{warsh}(R,\mathsf{O}) = R;(\mathsf{I}_\mathsf{O};R)^* = R;(\mathsf{O};R)^* = R;\mathsf{O}^* = R;\mathsf{I} = R,$$

which corresponds (since O describes $\emptyset$) to the equivalence of $(x,y) \in R_0$ and $(x,y) \in R$ for all $x, y \in X$, and it implies for the universal vector $\mathsf{L} : X \leftrightarrow \mathbf{1}$ that

$$\mathit{warsh}(R,\mathsf{L}) = R;(\mathsf{I}_\mathsf{L};R)^* = R;(\mathsf{I};R)^* = R;R^* = R^+,$$

which corresponds (since L describes X) to the equivalence of $(x,y) \in R_m$ and $(x,y) \in R^+$ for all $x, y \in X$. Our goal is to obtain an inductive specification of

(4) so that a later implementation in Haskell can base on pattern matching. In respect thereof, the just shown equations correspond to the induction base and the termination case. They, therefore, motivate to consider a vector $v : X \leftrightarrow \mathbf{1}$ such that $v \neq \mathsf{L}$, to take an arbitrary point $p : X \leftrightarrow \mathbf{1}$ for that $p \subseteq \overline{v}$ holds, and to express $\mathit{warsh}(R, v \cup p)$ in terms of $\mathit{warsh}(R, v)$.

Due to the Schröder equivalences (1) the assumption $p \subseteq \overline{v}$ is equivalent to $v;p^\mathsf{T} \subseteq \overline{\mathsf{I}}$ and also to $p;v^\mathsf{T} \subseteq \overline{\mathsf{I}}$. Using these properties we can calculate

$$
\begin{aligned}
\mathsf{I} \cap (v \cup p);(v \cup p)^\mathsf{T} &= \mathsf{I} \cap (v;v^\mathsf{T} \cup v;p^\mathsf{T} \cup p;v^\mathsf{T} \cup p;p^\mathsf{T}) \\
&= \mathsf{I} \cap (v;v^\mathsf{T} \cup p;p^\mathsf{T}) && v;p^\mathsf{T} \cup p;v^\mathsf{T} \subseteq \overline{\mathsf{I}} \\
&= (\mathsf{I} \cap v;v^\mathsf{T}) \cup p;p^\mathsf{T} && p;p^\mathsf{T} \subseteq \mathsf{I}.
\end{aligned}
$$

Now, the inductive specification of (4) we are looking for follows from

$$
\begin{aligned}
\mathit{warsh}(R, v \cup p) &= R;(\mathsf{I}_{v \cup p};R)^* && \text{by (4)} \\
&= R;((\mathsf{I}_v \cup p;p^\mathsf{T});R)^* && \text{see above} \\
&= R;(\mathsf{I}_v;R \cup p;p^\mathsf{T};R)^* \\
&= R;(\mathsf{I}_v;R)^*;(p;p^\mathsf{T};R;(\mathsf{I}_v;R)^*)^* && \text{by (2)} \\
&= \mathit{warsh}(R,v);(p;p^\mathsf{T};\mathit{warsh}(R,v))^* && \text{by (4)} \\
&= \mathit{warsh}(R,v);(\mathsf{I} \cup p;p^\mathsf{T};\mathit{warsh}(R,v))^+ && \text{by (2)} \\
&= \mathit{warsh}(R,v);(\mathsf{I} \cup p;p^\mathsf{T};\mathit{warsh}(R,v)) && \text{see below} \\
&= \mathit{warsh}(R,v) \cup \mathit{warsh}(R,v);p;p^\mathsf{T};\mathit{warsh}(R,v).
\end{aligned}
$$

The correctness of the last but one step follows from

$$
\begin{aligned}
p;p^\mathsf{T};\mathit{warsh}(R,v);p;p^\mathsf{T};\mathit{warsh}(R,v) &\subseteq p;\mathsf{L};p^\mathsf{T};\mathit{warsh}(R,v) \\
&= p;p^\mathsf{T};\mathit{warsh}(R,v) && p \text{ vector}
\end{aligned}
$$

because this is the transitivity of $p;p^\mathsf{T};\mathit{warsh}(R,v)$ and, as a consequence, the relation equals its transitive closure.

If we apply, as customary in functional programming, a `let`-clause to avoid the multiple calls of the function *warsh*, then the just shown three equations lead to the following functional algorithm for computing transitive closures that only uses the constants and operations of relation algebra.

$$
\begin{aligned}
&\mathit{transcl} : [X \leftrightarrow X] \to [X \leftrightarrow X] \\
&\mathit{transcl}(R) = \mathit{warsh}(R, \mathsf{L}) \\[4pt]
&\mathit{warsh} : [X \leftrightarrow X] \times [X \leftrightarrow \mathbf{1}] \to [X \leftrightarrow X] \\
&\mathit{warsh}(R, \mathsf{O}) \quad\;\; = R \\
&\mathit{warsh}(R, v \cup p) = \mathtt{let}\ S = \mathit{warsh}(R,v) \\
&\qquad\qquad\qquad\quad\;\; \mathtt{in}\ \ S \cup S;p;p^\mathsf{T};S
\end{aligned}
\tag{5}
$$

In the second equation of the function *warsh* it is implicitely assumed that $v \neq \mathsf{L}$ and that p is a point of type $[X \leftrightarrow \mathbf{1}]$ with $p \subseteq \overline{v}$. If X is finite and of the form $X = \{v_1, \ldots, v_m\}$, then the universal vector $\mathsf{L} : X \leftrightarrow \mathbf{1}$ can be repesented as union $p_1 \cup \ldots \cup p_m$ of m pairwise disjoint points $p_1, \ldots, p_m : X \leftrightarrow \mathbf{1}$, where p_i describes the element $v_i \in X$, $1 \leq i \leq m$. In such a case the call $\mathit{transcl}(R)$ leads to the total number of $m+1$ calls of the function *warsh*.

At this place it should be mentioned that the implicit assumption on v and p in the algorithm of (5) can be avoided if we suppose a choice function *point* to be at hand (as, for instance, in the relation-algebraic tool RelView [3]) such that a call $point(v)$ yields a point that is contained in the nonempty vector v. For the point $p := point(v)$ then it holds $v = (v \cap \overline{p}) \cup p$. Using this property in combination with a conditional, the inductive specification of the function *warsh* of (5) then can be reformulated as recursive function (that now decreases the vector argument) as follows.

$$\begin{array}{l} warsh : [X \leftrightarrow X] \times [X \leftrightarrow \mathbf{1}] \to [X \leftrightarrow X] \\ warsh(R, v) \;=\; \texttt{if } v = \mathsf{O} \texttt{ then } R \\ \qquad\qquad\qquad\qquad \texttt{else let } p \;=\; point(v) \\ \qquad\qquad\qquad\qquad\qquad\qquad S = warsh(R, v \cap \overline{p}) \\ \qquad\qquad\qquad\qquad\qquad \texttt{in } \; S \cup S; p; p^{\mathsf{T}}; S \end{array} \tag{6}$$

But the version of *warsh* given in (6) does not directly lead to the final Haskell program in the typical inductive functional style we are aiming for. Therefore, in the remainder of the paper we concentrate on the development of the functions of (5).

4 From Relation Algebra to Lists

In this section we show how the functional algorithm (5) can be transformed into a version that bases on a representation of relations by means of lists of successor sets and (as we will demonstrate in Section 5) immediately can be implemented in Haskell using the pre-defined Haskell datatype for lists only.

In the first step we represent relations $F : X \leftrightarrow X$ by functions $f : X \to 2^X$ such that for all $x, y \in X$ it holds $(x, y) \in F$ if and only if $y \in f(x)$. Using graphtheoretic terminology, hence, relations are represented by functions which map vertices to their sets of successors. Furthermore, we represent vectors (points) by the subsets (elements) of X they describe.

Now, suppose that the input relation $R : X \leftrightarrow X$ of *transcl* is represented by the function $r : X \to 2^X$. Since the universal vector $\mathsf{L} : X \leftrightarrow \mathbf{1}$ describes (i.e., is represented by) the set X we obtain the following version of the main function *transcl* of (5) that now works on functions,

$$\begin{array}{l} transcl : (X \to 2^X) \to (X \to 2^X) \\ transcl(r) = warsh(r, X) \end{array} \tag{7}$$

To obtain a corresponding new version of the auxiliary function *warsh* of (5) we again assume that $r : X \to 2^X$ represents $R : X \leftrightarrow X$. The induction base $warsh(r, \emptyset) = r$ is a consequence of the fact that the empty vector $\mathsf{O} : X \leftrightarrow \mathbf{1}$ is represented by the empty set $\emptyset$. For the remaining case, assume that the vector $v : X \leftrightarrow \mathbf{1}$ is represented by the subset V of X, the point $p : X \leftrightarrow \mathbf{1}$ is represented by the element $e \in X \setminus V$ and the relation $S : X \leftrightarrow X$ of the `let`-clause of (5) is represented by the function $s : X \to 2^X$. For all $x, y \in X$ we then can calculate

as follows, where the conditional is introduced to enhance readability and to prepare a later translation into Haskell.

$$\begin{array}{ll}
 & (x,y) \in S \cup S;p;p^{\mathsf{T}};S \\
\iff & (x,y)\in S \;\vee\; (x,y)\in S;p;p^{\mathsf{T}};S \\
\iff & (x,y)\in S \;\vee\; \exists\, i,j \in X : (x,i)\in S \wedge (i,j) \in p;p^{\mathsf{T}} \wedge (j,y) \in S \\
\iff & (x,y)\in S \;\vee\; \exists\, i,j \in X : (x,i)\in S \wedge i = e \wedge j = e \wedge (j,y) \in S \\
\iff & (x,y)\in S \;\vee\; ((x,e)\in S \wedge (e,y) \in S) \\
\iff & y\in s(x) \;\vee\; (e\in s(x) \wedge y\in s(e)) \\
\iff & y\in s(x) \;\vee\; \texttt{if}\ e\in s(x)\ \texttt{then}\ y\in s(e)\ \texttt{else}\ \mathit{false} \\
\iff & y\in s(x) \;\vee\; \texttt{if}\ e\in s(x)\ \texttt{then}\ y\in s(e)\ \texttt{else}\ y\in\emptyset \\
\iff & y\in s(x) \;\vee\; y \in \texttt{if}\ e\in s(x)\ \texttt{then}\, s(e)\ \texttt{else}\ \emptyset \\
\iff & y \in \texttt{if}\ e\in s(x)\ \texttt{then}\ s(x) \cup s(e)\ \texttt{else}\ s(x)
\end{array}$$

If we apply the familar λ-notation to denote anonymous functions, then the just proved relationship in combination with β-conversion shows that the anonymous function $\lambda x.\texttt{if}\ e \in s(x)\ \texttt{then}\ s(x) \cup s(e)\ \texttt{else}\ s(x)$ (where x ranges over X) represents the relation $S \cup S;p;p^{\mathsf{T}};S : X \leftrightarrow X$. Using additionally that the vector $v \cup p : X \leftrightarrow \mathbf{1}$ is represented by the subset $V \cup \{e\}$ of X we, finally, obtain the following new version of the auxiliary function *warsh* of (5).

$$\begin{array}{l}
\mathit{warsh} : (X \to 2^X) \times 2^X \to (X \to 2^X) \\
\mathit{warsh}(r, \emptyset) \qquad\quad = r \\
\mathit{warsh}(r, V \cup \{e\}) = \texttt{let}\ s = \mathit{warsh}(r, V) \\
\qquad\qquad\qquad\qquad\quad\; \texttt{in}\ \ \lambda x.\texttt{if}\ e\in s(x)\ \texttt{then}\ s(x) \cup s(e)\ \texttt{else}\ s(x)
\end{array} \tag{8}$$

In analogy to the version of *warsh* of (5) in the second equation of the version of (8) it is implicitely assumed that $V \neq X$ and that the element e of X is not contained in V.

Having replaced relations by functions, vectors by sets and points by elements, in the second step we now represent the arguments and the results of the function *transcl* of (7) and the function *warsh* of (8) by lists. For the argument and the result of *transcl* and, hence, also for the first argument and the result of *warsh* we take lists over 2^X, i.e., elements of $(2^X)^*$, and for the second argument of *warsh* we take lists over X, i.e., elements of X^*. To simplify the presentation *we assume for the following that the set X consists of the natural numbers* $0, 1, \ldots, m$. The additional number 0 (in Section 3 we assumed $X = \{v_1, \ldots, v_m\}$ as carrier set) is motivated with a view to a later use of Haskell lists. Here 0 is the index of the first list element[1].

The assumption on the carrier set X of all relations we consider allows to represent the function $r : X \to 2^X$ by the list $rs \in (2^X)^*$ of length $m + 1$ such that for all $x \in X$ the x-th component of rs — in the sequel we use the Haskell notation $rs!!x$ for this construction — equals the set $r(x)$. In the same way the function $s : X \to 2^X$ of the `let`-clause of *warsh* is represented by a list

[1] On the contrary, using $X = \{v_0, v_1, \ldots, v_m\}$ in Section 3 would cause problems with the definition of the subsets $X_i := \{v_0, \ldots, v_i\}$, since $\emptyset = X_i$ then requires $i = -1$.

$ss \in (2^X)^*$ of length $m+1$ such that $ss!!x = s(x)$ for all $x \in X$. The second argument of *warsh* is represented by the increasingly sorted list of the elements it contains, where we additionally do not allow *multiple occurrences of elements*.

Because the set X is represented by the increasingly sorted list of the natural numbers from 0 to m and m equals the length of the list rs minus 1, from the just introduced list representation we get the following new version of the main function *transcl*. In this version we apply the Haskell notation $\texttt{[}0..m\texttt{]}$ for the increasingly sorted list of the natural numbers from 0 to m to prepare the later translation into Haskell.

$$\begin{aligned} &\mathit{transcl} : (2^X)^* \to (2^X)^* \\ &\mathit{transcl}(rs) = \mathit{warsh}(rs, \texttt{[}0..|rs|-1\texttt{]}) \end{aligned} \tag{9}$$

To obtain a list-based version of the auxiliary function *warsh*, first, we represent the anonymous function of the `let`-clause of (8) by a list over 2^X. By assumption it holds $ss!!x = s(x)$ for all $x \in X$, that is, all elements x of the list $\texttt{[}0..m\texttt{]}$. Using a notation similar to Haskell's well-known list comprehension we, therefors, get for the anonymous function $\lambda x.\texttt{if } e \in s(x) \texttt{ then } s(x) \cup s(e) \texttt{ else } s(x)$ the following list representation.

$$\texttt{[if } e \in ss!!x \texttt{ then } ss!!x \cup ss!!e \texttt{ else } ss!!x \mid x \in \texttt{[}0..m\texttt{]]} \tag{10}$$

It is obvious that the list comprehension specified in (10) coincides with the list comprehension $\texttt{[if } e \in ms \texttt{ then } ms \cup ss!!e \texttt{ else } ms \mid ms \in ss\texttt{]}$. If we suppose that the list $vs \in X^*$ represents the set V of (8), then the list $e : vs$ with the additional first element $e \in X \setminus V$ represents the set $V \cup \{e\}$. The last two properties in connection with the fact that the empty list represents the empty set show that the function of (8) is correctly implemented by the subsequent list version.

$$\begin{aligned} &\mathit{warsh} : (2^X)^* \times X^* \to (2^X)^* \\ &\mathit{warsh}(rs, \texttt{[]}) \quad = rs \\ &\mathit{warsh}(rs, e : vs) = \texttt{let } ss = \mathit{warsh}(rs, vs) \\ &\qquad\qquad\qquad\quad\; \texttt{in [if } e \in ms \texttt{ then } ms \cup ss!!e \texttt{ else } ms \mid ms \in ss\texttt{]} \end{aligned} \tag{11}$$

By (11) we have reached the desired inductive functional style. Note that in the algorithm no longer an implicit assumption on vs and e is required.

5 Implementations in Haskell

Now, we are in the position to translate the functions of (9) and (11) into the functional programming language Haskell and to show that the resulting Haskell program runs in cubic time, i.e., has the same runtime complexity as the traditional imperative array-based implementation of Warshall's algorithm. We assume the reader to be familiar with Haskell. Otherwise, he may consult one of the well-known textbooks about it, for example [4, 10].

To obtain an implementation of the functions *transcl* and *warsh* of (9) and (11) in Haskell we represent lists over 2^X, i.e., lists of sets of integers, by lists of lists of integers. For the appearing lists of integers we require the same properties as in Section 4. That is, these lists (and, as a consequence, all successor lists of the input of the main function) have to be increasingly sorted and without multiple occurrences of elements. The reason for this precondition on the input will become clear later if we consider the runtime complexity.

If we go from lists of sets to list of lists and formulate the result in Haskell, then version (9) of the main function becomes the following Haskell function.

```
transcl :: [[Int]] -> [[Int]]
transcl rs =                                  (12)
  warsh rs [0..length rs - 1]
```

Subsets of X are implemented by Haskell lists. Hence, set-membership $e \in ms$ can directly be implemented by the pre-defined Haskell operation `elem`. Assuming additionally a Haskell function `cup` to be at hand that implements set union on the list implementations of sets, a straightforward translation of the function of (11) into Haskell code looks as follows.

```
warsh :: [[Int]] -> [Int] -> [[Int]]
warsh rs [] = rs
warsh rs (e:vs) =                                           (13)
  let ss = warsh rs vs
  in  [if elem e ms then cup ms (ss!!e) else ms | ms <- ss]
```

From Section 3 we know already that a call `transcl rs` leads to the total number of $m+1$ calls of the Haskell function `warsh`. The list specified by the list comprehension of `warsh` consists of $m+1$ component lists. Since each of these $m+1$ lists possesses at most $m+1$ elements and the `elem` test requires linear time in the length of the list argument, the entire list comprehension of the Haskell function of (13) can be evaluated in time $O(m^2)$ if the call `cup ms (ss!!e)` only requires time $O(m)$. The list `ss!!e` can be computed in time $O(m)$. A straightforward implementation of set union on a list implementation of sets requires quadratic time. But we can do better using that, because of the precondition, the lists `ms` and `ss!!e` are increasingly sorted and without multiple occurrences of elements. On such lists an obvious linear implementation of set union is given by the following Haskell function that merges two sorted lists into a sorted one and removes at the same time all multiple occurrences of elements.

```
cup :: [Int] -> [Int] -> [Int]
cup [] ys = ys
cup xs [] = xs
cup (x:xs) (y:ys) =                                (14)
  case compare x y of EQ -> x : cup xs ys
                      LT -> x : cup xs (y:ys)
                      GT -> y : cup (x:xs) ys
```

Altogether, the functions of (12), (13) and (14) constitute a Haskell program for computing transitive closures that bases on a representation of relations via lists of successor lists and runs in cubic time.

We conclude the section with two modifications of the derived Haskell program. The first one concerns the precondition. The cubic runtime remains preserved if in the Haskell function of (12) the call `warsh rs [0..length rs - 1]` is replaced by the call `warsh (map (nub.sort) rs) [0..length rs - 1]`, where the Haskell operation `nub` removes duplicate elements from a list, `sort` is the pre-defined sorting function on lists, "`.`" denotes function composition and `map` is the higher-order function that applies a function to each component of a list. The advantage of this modification is that its correctness does no longer depend on the fact that the successor lists of the input are strictly increasing, without sacrificing the runtime complexity. The second modification concerns the element test in the list comprehension of (13). Again the runtime complexity $O(m^3)$ remains preserved if in the Haskell function of (13) the call `elem e ms` is replaced by the call `iselem e ms` of the following Haskell function.

```
iselem :: Int -> [Int] -> Bool
iselem x [] = False
iselem x (y:ys) =
  case compare x y of EQ -> True                    (15)
                      GT -> iselem x ys
                      LT -> False
```

Practical experiments have shown that by this modification the runtime – depending on the input – to a greater or lesser extent is improved. This is because the function of (15) takes advantage of the fact that the successor lists are increasingly sorted.

6 Conclusion

We have shown how systematically to develop a purely functional version of Warshall's algorithm for computing transitive closures by combining relation-algebraic reasoning and the technique of data refinement. The final algorithm bases on an implementation of relations by lists of successor lists and immediately can be implemented in the functional programming language Haskell. The resulting Haskell program has the same runtime complexity as the traditional imperative array-based implementation of Warshall's algorithm.

Directed graphs are nothing else than relations on sets of vertices. Hence, the computation of the transitive closure of a relation can be seen as a problem of graph-theory. In this regard, related to our work are all the approaches to program graph algorithms in a functional language. In the meantime functional graph algorithms have a long tradition. We only want to mention three examples. In [2] transformational programming is applied to derive certain functional reachability algorithms. The paper [6] deals with the specification and functional

computation of the depth-first search forest and presents some classical applications (topological sorting, testing for cycles, strongly connected components) in the functional style. To achieve a linear running time, monads are used to mimic the imperative marking technique. Instead of regarding graphs as monolithic data as the just mentioned papers do, in [5] graphs are inductively generated. This allows to write many graph algorithms in the typical functional style.

References

1. Aarts A. et al.: Fixed point calculus. Information Processing Letters 53, 131-136 (1996).
2. Berghammer R., Ehler H., Zierer H.: Development of graph algorithms by program transformation. In: Göttler H., Schneider H.J. (eds.): Graph-Theoretic Concepts in Computer Science. LNCS 314, Springer, 206-218 (1988).
3. Berghammer R., Neumann F.: RelView — An OBDD-based Computer Algebra system for relations. In: Ganzha V.G., Mayr E.W., Vorozhtsov E. (eds.): Computer Algebra in Scientific Computing, LNCS 3718, Springer, 40-51 (2005).
4. Bird R.: Introduction to functional programming using Haskell. Second edition, Prentice Hall (1998).
5. Erwig M.: Inductive graphs and functional graph algorithms. Journal of Functional Programming 11, 467-492 (2001).
6. King D.J., Launchbury J.: Structuring depth-first search algorithms in Haskell. Proc. ACM Symposium on Principles of Programming. ACM Press, 344-356 (1995).
7. Schmidt G., Ströhlein T.: Relations and graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoretical Computer Science, Springer (1993).
8. Spivey J.M.: The Z notation: A reference manual. Prentice Hall (1992).
9. Tarski A.: On the calculus of relations. Journal of Symbolic Logic 6, 73-89 (1941).
10. Thomson S.: Haskell – The craft of functional programming. Addison-Wesley (1999).
11. Warshall S.: A theorem on Boolean matrices. Journal of the ACM 9, 11-12 (1962).

Transforming Functional Logic Programs into Monadic Functional Programs

Bernd Braßel, Sebastian Fischer, Michael Hanus, Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{bbr|sebf|mh|fre}@informatik.uni-kiel.de

We present a high-level transformation scheme to translate lazy functional logic programs into pure Haskell programs. This transformation is based on a recent proposal to efficiently implement lazy non-deterministic computations in Haskell into monadic style. We build on this work and define a systematic method to transform lazy functional logic programs into monadic programs with explicit sharing. This results in a transformation scheme which produces high-level and flexible target code. For instance, the target code is parametric w.r.t. the concrete evaluation monad. Thus, different monad instances could, for example, define different search strategies (e.g., depth-first, breadth-first, parallel).

Reference

1. Braßel B., Fischer S., Hanus M., Reck F.: Transforming Functional Logic Programs into Monadic Functional Programs In: Proc. of the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2010). Springer. To appear.