INSTITUT FÜR INFORMATIK

Programmiersprachen und Rechenkonzepte

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

Bad Honnef, 4.-6. Mai 2009

Michael Hanus, Bernd Braßel (Hrsg.)

Bericht Nr. 0915

Juli 2009

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

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

Programmiersprachen und Rechenkonzepte

26. Workshop der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“ Bad Honnef, 4.-6. Mai 2009

Michael Hanus, Bernd Braßel (Hrsg.)

Bericht Nr. 0915
Juli 2009

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

Dieser Bericht enthält eine Zusammenstellung der Beiträge des
26. Workshops Programmiersprachen und Rechenkonzepte,
Physikzentrum Bad Honnef, 4.-6. Mai 2009.

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 spannenden Ereignis machen. Ein besonderer Dank gilt den Autoren die mit ihren vielfältigen Beiträgen zu diesem Band beigetragen haben. Ein abschließender Dank gebührt noch den Mitarbeitern des Physikzentrums Bad Honnef, die durch ihre umfassende Betreuung für eine angenehme und anregende Atmosphäre gesorgt haben.

Kiel, im Juli 2009 — Michael Hanus, Bernd Braßel

Inhaltsverzeichnis

Free Theorems Involving Type Constructor Classes*
(Extended Abstract)

Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany

voigt@tcs.inf.tu-dresden.de

Abstract. One of the strengths of functional languages like Haskell is an expressive type system. For example, free theorems [6] allow the derivation of useful statements about programs from their (polymorphic) types alone. And yet, some of the benefits this should hold for reasoning about programs seem not to be realised to full extent. For example, Haskell uses monads [2] to structure programs by separating concerns [7] and to safely mingle pure and impure computations [3]. A lot of code can be kept independent of a concrete choice of monad. This pertains to functions from the Prelude (Haskell's standard library) like

$$\mathit{sequence} :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \alpha] \to \mu\ [\alpha]\,,$$

but also to many user-defined functions. This is certainly a boon for modularity of programs. But what about reasoning?
Type signatures like above signify polymorphism not only over ordinary types (like α), but also over type *constructors* (like μ) restricted by *class constraints* (like Monad). Our aim is to show how to obtain free theorems in such situations, and to discuss interesting applications.

1 Introduction

Let us consider a more specific example, say functions of the type $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$. Here are some:

$$f_1 = \mathit{head}$$

$$f_2\ \mathit{ms} = \mathit{sequence}\ \mathit{ms} \gg\!= \mathit{return} \circ \mathit{sum}$$

$$f_3 = f_2 \circ \mathit{reverse}$$

$$\begin{aligned} f_4\ [\,] &= \mathit{return}\ 0 \\ f_4\ (m : \mathit{ms}) &= \mathbf{do}\ \ i \leftarrow m \\ &\qquad\quad \mathbf{let}\ l = \mathit{length}\ \mathit{ms} \\ &\qquad\quad \mathbf{if}\ i > l\ \mathbf{then}\ \mathit{return}\ (i + l)\ \mathbf{else}\ f_4\ (\mathit{drop}\ i\ \mathit{ms}) \end{aligned}$$

* The full version of this paper appears at ICFP'09.

As we see, there is quite a variety of such functions. There can be simple selection of one of the monadic computations from the input list (as in f_1), there can be sequencing of these monadic computations (in any order) and some action on the encapsulated values (as in f_2 and f_3), and the behaviour, in particular the choice which of the computations from the input list are actually performed, can even depend on the encapsulated values themselves (as in f_4, made a bit artificial here). Further possibilities are that some of the monadic computations from the input list are performed repeatedly, and so forth. But still, all these functions also have something in common. They can only *combine* whatever monadic computations, and associated effects, they encounter in their input lists, but they cannot *introduce* new effects of any concrete monad, not even of the one they are actually operating on in a particular application instance. This limitation is determined by the function type. For if an f were, on and of its own, to cause any additional effect to happen, be it by writing to the output, by introducing additional branching in the nondeterminism monad, or whatever, then it would immediately fail to get the above type parametric over μ. In a language like Haskell, should not we be able to profit from this kind of abstraction for reasoning purposes?

If so, what kind of insights can we hope for? One thing to expect is that in the special case when the concrete computations in an input list passed to an $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ correspond to pure values (e.g., are values of type $\mathsf{IO}\ \mathsf{Int}$ that do not perform any actual input or output), then the same should hold of f's result for that input list. This statement is quite intuitive from the above observation about f being unable to cause new effects on its own. But what about more interesting statements, for example the preservation of certain invariants? Say we pass to f a list of stateful computations and we happen to know that they do depend on, but do not alter (a certain part of) the state. Is this property preserved throughout the evaluation of a given f? Or say the effect encapsulated in f's input list is nondeterminism but we would like to simplify the program by restricting the computation to a deterministically chosen representative from each nondeterministic manifold. Under what conditions, and for which kind of representative-selection functions, is this simplification safe and does not lead to problems like a collapse of an erstwhile nonempty manifold to an empty one from which no representative can be chosen at all?

One could go and study these questions for particular functions like the f_1 to f_4 given further above. But instead we would like to answer them for any function of type $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ in general, without consulting particular function definitions. And we would not like to restrict to the two or three scenarios depicted in the previous paragraph. Rather, we want to explore more abstract settings of which statements like the ones in question above can be seen, and dealt with, as particular instances. And, of course, we prefer a generic methodology that applies equally well to other types than the specific one of f considered so far in this introduction. These aims are not arbitrary or far-fetched. Precedent has been set with the theorems obtained for free by Wadler [6] from relational parametricity [4]. Derivation of such free theorems,

too, is a methodology that applies not only to a single type, works independently of particular function definitions, and applies to a diverse range of scenarios.

Unsurprisingly then, we do build on Reynolds' and Wadler's work. Of course, the framework that is usually considered when free theorems are derived needs to be extended to deal with types like $\mathsf{Monad}\ \mu \Rightarrow \ldots$. But the ideas needed to do so are there for the taking. Indeed, both relational parametricity extended for polymorphism over type constructors rather than over ordinary types only, as well as relational parametricity extended to take class constraints into account, are in the folklore. However, these two strands of possible extension have not been combined before, and not been used as we do.

2 Free Theorems

Consider the type signature $f :: [\alpha] \to [\alpha]$. What does it tell us about the function f? For sure that it takes lists as input and produces lists as output. But we also see that f is polymorphic, due to the type variable α, and so must work for lists over arbitrary element types. How, then, can elements for the output list come into existence? The answer is that the output list can only ever contain elements from the input list. For the function, not knowing the element type of the lists it operates over, cannot possibly make up new elements of any concrete type to put into the output, such as 42 or True, or even id, because then f would immediately fail to have the general type $[\alpha] \to [\alpha]$.

> So for any input list l (over any element type) the output list $f\ l$ consists solely of elements from l.

But how can f decide which elements from l to propagate to the output list, and in which order and multiplicity? The answer is that such decisions can only be made based on the input list l. For in a pure functional language f has no access to any global state or other context based on which to decide. It cannot, for example, consult the user in any way about what to do. And the means by which to make decisions based on l are limited as well. In particular, decisions cannot possibly depend on any specifics of the elements of l. For the function is ignorant of the element type, and so is prevented from analysing list elements in any way (be it by pattern-matching, comparison operations, or whatever). In fact, the only means for f to drive its decision-making is to inspect the *length* of l, because that is the only element-independent "information content" of a list.

> So for any pair of lists l and l' of same length (but possibly over different element types) the lists $f\ l$ and $f\ l'$ are formed by making the same position-wise selections of elements from l and l', respectively.

Now consider the following standard Haskell function:

$$\begin{aligned} &map :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\ &map\ g\ [\,] \qquad\ = [\,] \\ &map\ g\ (a : as) = (g\ a) : (map\ g\ as) \end{aligned}$$

Clearly, *map g* for any g preserves the lengths of lists. So if $l' = map\ g\ l$, then $f\ l$ and $f\ l'$ are of the same length and contain, at each position, position-wise exactly corresponding elements from l and l', respectively. Since, moreover, any two position-wise corresponding elements, one from l and one from $l' = map\ g\ l$, are related by the latter being the g-image of the former, we have that at each position $f\ l'$ contains the g-image of the element at the same position in $f\ l$.

> So for any list l and (type-appropriate) function g, we have $f\ (map\ g\ l) = map\ g\ (f\ l)$.

Note that during the reasoning leading up to that statement we did not (need to) consider the actual definition of f at all. The methodology of deriving free theorems à la [6] is a way to obtain statements of this flavour for arbitrary function types, and in a more disciplined (and provably sound) manner than the mere handwaving performed above.

The key to doing so is to interpret types as relations. For example, given the type signature $f :: [\alpha] \to [\alpha]$, we take the type and replace every quantification over type variables, including implicit quantification (note that the type $[\alpha] \to [\alpha]$, by Haskell convention, really means $\forall\alpha.\ [\alpha] \to [\alpha]$), by quantification over relation variables: $\forall\mathcal{R}.\ [\mathcal{R}] \to [\mathcal{R}]$. Then, there is a systematic way of reading such expressions over relations as relations themselves. In particular,

- base types like Int are read as identity relations,
- for relations $\mathcal{R}$ and $\mathcal{S}$, we have

$$\mathcal{R} \to \mathcal{S} = \{(f, g) \mid \forall(a, b) \in \mathcal{R}.\ (f\ a, g\ b) \in \mathcal{S}\}\text{, and}$$

- for types τ and τ' with at most one free variable, say α, and a function $\mathcal{F}$ on relations such that every relation $\mathcal{R}$ between closed types τ_1 and τ_2, denoted $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$, is mapped to a relation $\mathcal{F}\ \mathcal{R} : \tau[\tau_1/\alpha] \Leftrightarrow \tau'[\tau_2/\alpha]$, we have

$$\forall\mathcal{R}.\ \mathcal{F}\ \mathcal{R} = \{(u, v) \mid \forall\tau_1, \tau_2, \mathcal{R} : \tau_1 \Leftrightarrow \tau_2.\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}\ \mathcal{R}\}$$

  (Here, $u_{\tau_1} :: \tau[\tau_1/\alpha]$ is the instantiation of $u :: \forall\alpha.\ \tau$ to the type τ_1, and similarly for v_{τ_2}. In what follows, we will always leave type instantiation implicit.)

Also, every fixed type constructor is read as an appropriate construction on relations. For example, the list type constructor maps every relation $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ to the relation $[\mathcal{R}] : [\tau_1] \Leftrightarrow [\tau_2]$ defined by (the least fixpoint of)

$$[\mathcal{R}] = \{([\,], [\,])\} \cup \{(a : as, b : bs) \mid (a, b) \in \mathcal{R},\ (as, bs) \in [\mathcal{R}]\}\,,$$

the Maybe type constructor maps every relation $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ to the relation $\mathsf{Maybe}\ \mathcal{R} : \mathsf{Maybe}\ \tau_1 \Leftrightarrow \mathsf{Maybe}\ \tau_2$ defined by

$$\mathsf{Maybe}\ \mathcal{R} = \{(\mathsf{Nothing}, \mathsf{Nothing})\} \cup \{(\mathsf{Just}\ a, \mathsf{Just}\ b) \mid (a, b) \in \mathcal{R}\}\,,$$

and similarly for other user-definable types.

The key insight of relational parametricity à la [4] now is that any expression over relations that can be built as above, by interpreting a closed type, denotes the identity relation on that type.

For the above example, this insight means that any $f :: \forall\alpha.\ [\alpha] \to [\alpha]$ satisfies $(f, f) \in \forall\mathcal{R}.\ [\mathcal{R}] \to [\mathcal{R}]$, which by unfolding some of the above definitions is equivalent to having for every τ_1, τ_2, $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$, $l :: [\tau_1]$, and $l' :: [\tau_2]$ that $(l, l') \in [\mathcal{R}]$ implies $(f\ l, f\ l') \in [\mathcal{R}]$, or, specialised to the function level ($\mathcal{R} \mapsto g$, and thus $[\mathcal{R}] \mapsto \mathit{map}\ g$), for every $g :: \tau_1 \to \tau_2$ and $l :: [\tau_1]$ that $f\ (\mathit{map}\ g\ l) = \mathit{map}\ g\ (f\ l)$. This proof finally provides the formal counterpart to the intuitive reasoning earlier in this section.

3 The Extension to Type Constructor Classes

We now want to deal with two new aspects: with quantification over type *constructor* variables (rather than just over type variables) and with *class constraints* [8]. For both aspects, the required extensions to the interpretation of types as relations appear to be folklore, but have seldom been spelled out and have not been put to use before as we do here.

Regarding quantification over type *constructor* variables, the necessary adaptation is as follows. Just as free type variables are interpreted as relations between arbitrarily chosen closed types (and then quantified over via relation variables), free type constructor variables are interpreted as functions on such relations tied to arbitrarily chosen type constructors. Formally, let κ_1 and κ_2 be type constructors (of kind $* \to *$). A *relational action* for them, denoted $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$, is a function $\mathcal{F}$ on relations between closed types such that every $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ (for arbitrary τ_1 and τ_2) is mapped to an $\mathcal{F}\ \mathcal{R} : \kappa_1\ \tau_1 \Leftrightarrow \kappa_2\ \tau_2$. For example, the function $\mathcal{F}$ that maps every $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ to

$$\mathcal{F}\ \mathcal{R} = \{(\mathsf{Nothing}, [\,])\} \cup \{(\mathsf{Just}\ a, [b]) \mid (a, b) \in \mathcal{R}\}$$

is a relational action $\mathcal{F} : \mathsf{Maybe} \Leftrightarrow []$. The relational interpretation of a type quantifying over a type constructor variable is now performed in an analogous way as explained for quantification over type (and then, relation) variables above.

Regarding *class constraints*, [6, Section 3.4] directs the way by explaining how to treat the type class Eq in the context of deriving free theorems. The idea is to simply restrict the relations chosen as interpretation for type variables that are subject to a class constraint. Clearly, only relations between types that are instances of the class under consideration are allowed. Further restrictions are obtained from the respective class declaration. Namely, the restrictions must precisely ensure that every class method (seen as a new constant in the language) is related to itself by the relational interpretation of its type. This relatedness then guarantees that the overall result (i.e., that the relational interpretation of every closed type is an identity relation) stays intact [1]. The same approach immediately applies to type constructor classes as well. Consider, for example,

the Monad class declaration:

$$\begin{array}{l}\textbf{class}\ \mathsf{Monad}\ \mu\ \textbf{where}\\ \quad \mathit{return} :: \alpha \to \mu\ \alpha\\ \quad (\gg\!=) :: \mu\ \alpha \to (\alpha \to \mu\ \beta) \to \mu\ \beta\end{array}$$

Since the type of *return* is $\forall\mu.\ \mathsf{Monad}\ \mu \Rightarrow (\forall\alpha.\ \alpha \to \mu\ \alpha)$, we expect that $(\mathit{return}, \mathit{return}) \in \forall\mathcal{F}.\ \mathsf{Monad}\ \mathcal{F} \Rightarrow (\forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\ \mathcal{R})$, and similarly for $\gg\!=$. The constraint "$\mathsf{Monad}\ \mathcal{F}$" on a relational action is now defined in precisely such a way that both conditions will be fulfilled.

Definition 1. *Let κ_1 and κ_2 be type constructors that are instances of* Monad *and let $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ be a relational action. If*

- $(\mathit{return}_{\kappa_1}, \mathit{return}_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\ \mathcal{R}$ *and*
- $((\gg\!=_{\kappa_1}), (\gg\!=_{\kappa_2})) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \mathcal{F}\ \mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S})$,

then $\mathcal{F}$ is called a Monad-action.[1] *(While we have decided to generally leave type instantiation implicit, we explicitly retain instantiation of type constructors in what follows, except for some examples.)*

For example, given the following standard Monad instance definitions:

$$\begin{array}{l}\textbf{instance}\ \mathsf{Monad}\ \mathsf{Maybe}\ \textbf{where}\\ \quad \mathit{return}\ a = \mathsf{Just}\ a\\ \quad \mathsf{Nothing} \gg\!= k = \mathsf{Nothing}\\ \quad \mathsf{Just}\ a \quad \gg\!= k = k\ a\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathsf{Monad}\ []\ \textbf{where}\\ \quad \mathit{return}\ a = [a]\\ \quad \mathit{as} \gg\!= k = \mathit{concat}\ (\mathit{map}\ k\ \mathit{as})\end{array}$$

the relational action $\mathcal{F} : \mathsf{Maybe} \Leftrightarrow []$ given above is a Monad-action.

We are now ready to derive free theorems involving (polymorphism over) type constructor classes. For example, functions $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ as considered in the introduction will necessarily always satisfy $(f, f) \in \forall\mathcal{F}.\ \mathsf{Monad}\ \mathcal{F} \Rightarrow [\mathcal{F}\ \mathit{id}_{\mathsf{Int}}] \to \mathcal{F}\ \mathit{id}_{\mathsf{Int}}$, i.e., for every choice of type constructors κ_1 and κ_2 that are instances of Monad, and every Monad-action $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$, we have $(f_{\kappa_1}, f_{\kappa_2}) \in [\mathcal{F}\ \mathit{id}_{\mathsf{Int}}] \to \mathcal{F}\ \mathit{id}_{\mathsf{Int}}$. Based on this, we can prove several theorems by instantiating the $\mathcal{F}$ here, and provide plenty of examples of interesting results obtained for concrete monads. An important role will be played by a notion connecting different Monad instances on a functional level.

Definition 2. *Let κ_1 and κ_2 be instances of* Monad *and let $h :: \kappa_1\ \alpha \to \kappa_2\ \alpha$. If*

- $h \circ \mathit{return}_{\kappa_1} = \mathit{return}_{\kappa_2}$ *and*
- *for every choice of closed types τ and τ', $m :: \kappa_1\ \tau$, and $k :: \tau \to \kappa_1\ \tau'$,*

$$h\ (m \gg\!=_{\kappa_1} k) = h\ m \gg\!=_{\kappa_2} h \circ k\,,$$

then h is called a Monad-morphism.

[1] It is worth noting that "dictionary translation" [8] would be an alternative way of motivating this definition.

As a final preparation, we mention two laws that Monad instances κ are often expected to satisfy:

$$return_\kappa\ a \gg\!=_\kappa\ k = k\ a \tag{1}$$

$$(m \gg\!=_\kappa\ k) \gg\!=_\kappa\ q = m \gg\!=_\kappa\ (\lambda a \to k\ a \gg\!=_\kappa\ q) \tag{2}$$

4 Reasoning about Monadic Programs

We continue to focus on functions $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$. However, it should be emphasised that results of the same spirit can also be systematically obtained for other function types involving quantification over Monad-restricted type constructor variables (and for type constructor classes other than Monad).

4.1 Purity Preservation

As mentioned in the introduction, one first intuitive statement we naturally expect to hold of any $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ is that when all the monadic values supplied to f in the input list are actually pure (not associated with any proper monadic effect), then f's result value, though of some monadic type, should also be pure. After all, f itself, being polymorphic over μ, cannot introduce effects from any specific monad. This statement is expected to hold no matter what monad the input values live in. For example, if the input list consists of computations in the list monad, defined in the previous section and modelling nondeterminism, but all the concretely passed values actually correspond to deterministic computations, then we expect that f's result value also corresponds to a deterministic computation. Similarly, if the input list consists of IO computations, but we only pass ones that happen to have no side-effect at all, then f's result, though living in the IO monad, should also be side-effect-free. To capture the notion of "purity" independently of any concrete monad, we use the convention that the pure computations in any monad are those that may be the result of a call to *return*. Note that this does not mean that the values in the input list must *syntactically* be *return*-calls. Rather, each of them only needs to be *semantically equivalent* to some such call. The desired statement is now formalised as follows, and can then be used for reasoning about specific monads.

Theorem 1. *Let* $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$*, let* κ *be an instance of* Monad *satisfying law (1), and let* $l :: [\kappa\ \mathsf{Int}]$*. If every element in* l *is a* $return_\kappa$*-image, then so is* $f_\kappa\ l$*.*

Example 1. Let $l :: [[\mathsf{Int}]]$, i.e., $l :: [\kappa\ \mathsf{Int}]$ for $\kappa = []$. We might be interested in establishing that when every element in l is (evaluated to) a singleton list, then the result of applying any $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to l will be a singleton list as well. While this propagation is easy to see for f_1, f_2, and f_3 from the introduction, it is maybe not so immediately obvious for the f_4 given there. However, Theorem 1 tells us without any further effort that the statement in question does indeed hold for f_4, and for any other f of the same type.

4.2 Safe Value Extraction

A second general statement we are interested in is to deal with the case that the monadic computations provided as input are not necessarily pure, but we have a way of discarding the monadic layer and recovering underlying values. Somewhat in the spirit of *unsafePerformIO* :: IO $\alpha \to \alpha$, but for other monads and hopefully safe. Then, if we are interested only in a thus projected result value of f, can we show that it only depends on likewise projected input values, i.e., that we can discard any effects from the monadic computations in f's input list when we are not interested in the effectful part of the output computation? Clearly, it would be too much to expect this reduction to work for arbitrary "projections", or even arbitrary monads. Rather, we need to devise appropriate restrictions and prove that they suffice. The formal statement is as follows.

Theorem 2. *Let* $f ::$ Monad $\mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$*, let* κ *be an instance of* Monad*, and let* $p :: \kappa\ \alpha \to \alpha$*. If*

- $p \circ return_\kappa = id$ *and*
- *for every choice of closed types* τ *and* τ'*,* $m :: \kappa\ \tau$*, and* $k :: \tau \to \kappa\ \tau'$*,*

$$p\ (m \gg\!=_\kappa k) = p\ (k\ (p\ m))\,,$$

then $p \circ f_\kappa$ *gives the same result for any two lists of same length whose corresponding elements have the same* p*-images, i.e.,* $p \circ f_\kappa$ *can be "factored" as* $g \circ (map\ p)$ *for some suitable* $g ::$ [Int] $\to$ Int*.*

Note that no monad laws at all are needed in Theorem 2 (and its proof). The same will be true for the other theorems we are going to provide. But first, we consider several example applications of Theorem 2.

Example 2. Consider the well-known writer, or logging, monad (specialised here to the String monoid):

```
newtype Writer α = Writer (α, String)

instance Monad Writer where
  return a = Writer (a, "")
  Writer (a, s) >>= k = Writer (case k a of Writer (a', s') → (a', s ++ s'))
```

Assume we are interested in applying an $f ::$ Monad $\mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to an $l ::$ [Writer Int], yielding a monadic result of type Writer Int. Assume further that for some particular purpose during reasoning about the overall program, we are only interested in the actual integer value encapsulated in that result, as extracted by the following function:

$$\begin{aligned} &p :: \mathsf{Writer}\ \alpha \to \alpha \\ &p\ (\mathsf{Writer}\ (a, s)) = a \end{aligned}$$

Intuition suggests that then the value of $p\ (f\ l)$ should not depend on any logging activity of elements in l. That is, if l were replaced by another $l' ::$ [Writer Int]

encapsulating the same integer values, but potentially attached with different logging information, then $p\ (f\ l')$ should give exactly the same value. Since the given p fulfils the required conditions, Theorem 2 confirms this intuition.

Example 3. Recall the list monad defined in Section 3. It is tempting to use $head :: [\alpha] \to \alpha$ as an extraction function and expect that for every $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, we can factor $head \circ f$ as $g \circ (map\ head)$ for some suitable $g :: [\mathsf{Int}] \to \mathsf{Int}$. But actually this factorisation fails in a subtle way. Consider, for example, the (for the sake of simplicity, artificial) function

$$\begin{aligned} f_5 &:: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int} \\ f_5\ [\,] &= return\ 0 \\ f_5\ (m : ms) &= \mathbf{do}\ \ i \leftarrow m \\ &\qquad\quad f_5\ (\mathbf{if}\ i > 0\ \mathbf{then}\ ms\ \mathbf{else}\ tail\ ms) \end{aligned}$$

Then for $l = [[1], [\,]]$ and $l' = [[1, 0], [\,]]$, both of type $[[\mathsf{Int}]]$, we have $map\ head\ l = map\ head\ l'$, but $head\ (f_5\ l) \neq head\ (f_5\ l')$. In fact, the left-hand side of this inequation leads to an "head of empty list"-error, whereas the right-hand side delivers the value 0. Clearly, this means that the supposed g cannot exist for f_5 and $head$. An explanation for the observed failure is provided by the conditions imposed on p in Theorem 2. It is simply not true that for every m and k, $head\ (m \gg\!= k) = head\ (k\ (head\ m))$. More concretely, the failure for f_5 observed above arises from this equation being violated for $m = [1, 0]$ and $k = \lambda i \to \mathbf{if}\ i > 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ [0]$.

Example 4. Assume, just for the scope of this example, that the type constructor [] yields (the types of) nonempty lists only. Clearly, it becomes an instance of Monad by just the same definition as given in Section 3. There are now several choices for a never failing extraction function $p :: [\alpha] \to \alpha$. For example, p could be *head*, could be *last*, or could be the function that always returns the element in the middle position of its input list (and, say, the left one of the two middle elements in the case of a list of even length). But which of these candidates are "good" in the sense of providing, for every $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, a factorisation of $p \circ f$ into $g \circ (map\ p)$?

The answer is provided by the two conditions on p in Theorem 2, which specialised to the (nonempty) list monad require that

- for every a, $p\ [a] = a$, and
- for every choice of closed types τ and τ', $m :: [\tau]$, and $k :: \tau \to [\tau']$, $p\ (concat\ (map\ k\ m)) = p\ (k\ (p\ m))$.

From these conditions it is easy to see that now $p = head$ is good (in contrast to the situation in Example 3), and so is $p = last$, while the proposed "middle extractor" is not. It does not fulfil the second condition above, roughly because k does not necessarily map all its inputs to equally long lists.

4.3 Monad Subspacing

Next, we would like to tackle reasoning not about the complete absence of (à la Theorem 1), or disregard for (à la Theorem 2), monadic effects, but about finer nuances. Often, we know certain computations to realise only some of the potential effects to which they would be entitled according to the monad they live in. If, for example, the effect under consideration is nondeterminism à la the standard list monad, then we might know of some computations in that monad that they realise only none-or-one-nondeterminism, i.e., never produce more than one answer, but may produce none at all. Or we might know that they realise only non-failing-nondeterminism, i.e., always produce at least one answer, but may produce more than one. Then, we might want to argue that the respective nature of nondeterminism is preserved when combining such computations using, say, a function $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$. This preservation would mean that applying any such f to any list of empty-or-singleton lists always gives an empty-or-singleton list as result, and that applying any such f to any list of nonempty lists only gives a nonempty list as result for sure. Or, in the case of an exception monad ($\mathsf{Either\ String}$), we might want to establish that an application of f cannot possibly lead to any exceptional value (error description string) other than those already present somewhere in its input list. Such "invariants" can often be captured by identifying a certain "subspace" of the monadic type in question that forms itself a monad, or, indeed, by "embedding" another, "smaller", monad into the one of interest. Formal counterparts of the intuition behind the previous sentence and the vague phrases occurring therein can be found in Definition 2 and the following theorem, as well as in the subsequent examples.

Theorem 3. *Let* $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$*, let* $h :: \kappa_1\ \alpha \to \kappa_2\ \alpha$ *be a* Monad*-morphism, and let* $l :: [\kappa_2\ \mathsf{Int}]$*. If every element in* l *is an* h*-image, then so is* $f_{\kappa_2}\ l$*.*

Example 5. Consider the well-known reader monad:

$$\begin{aligned}
&\textbf{newtype}\ \mathsf{Reader}\ \rho\ \alpha = \mathsf{Reader}\ (\rho \to \alpha)\\
&\\
&\textbf{instance}\ \mathsf{Monad}\ (\mathsf{Reader}\ \rho)\ \textbf{where}\\
&\quad \mathit{return}\ a = \mathsf{Reader}\ (\lambda r \to a)\\
&\quad \mathsf{Reader}\ g \gg\!= k = \mathsf{Reader}\ (\lambda r \to \textbf{case}\ k\ (g\ r)\ \textbf{of}\ \mathsf{Reader}\ g' \to g'\ r)
\end{aligned}$$

Assume we are given a list of computations in a Reader monad, but it happens that all present computations depend only on a certain part of the environment type. For example, for some closed types τ_1 and τ_2, $l :: [\mathsf{Reader}\ (\tau_1, \tau_2)\ \mathsf{Int}]$, and for every element $\mathsf{Reader}\ g$ in l, $g\ (x, y)$ never depends on y. We come to expect that the same kind of independence should then hold for the result of applying any $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to l. And indeed it does hold by Theorem 3 with the following Monad-morphism:

$$\begin{aligned}
&h :: \mathsf{Reader}\ \tau_1\ \alpha \to \mathsf{Reader}\ (\tau_1, \tau_2)\ \alpha\\
&h\ (\mathsf{Reader}\ g) = \mathsf{Reader}\ (g \circ \mathit{fst})
\end{aligned}$$

Example 6. Let $l :: [\mathsf{IO}\ \mathsf{Int}]$ and assume that the only side-effects that elements in l have consist of writing strings to the output. We would like to use Theorem 3 to argue that the same is then true for the result of applying any $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to l. To this end, we need to somehow capture the concept of "writing (potentially empty) strings to the output as only side-effect of an IO computation" via an embedding from another monad. Quite naturally, we reuse the Writer monad from Example 2. The embedding function is as follows:

$$\begin{aligned} &h :: \mathsf{Writer}\ \alpha \to \mathsf{IO}\ \alpha \\ &h\ (\mathsf{Writer}\ (a, s)) = \mathit{putStr}\ s \gg \mathit{return}\ a \end{aligned}$$

What is left to do is to show that h is a Monad-morphism. But this property follows from $\mathit{putStr}\ \text{“”} = \mathit{return}\ ()$, $\mathit{putStr}\ (s +\!\!+ s') = \mathit{putStr}\ s \gg \mathit{putStr}\ s'$, and monad laws (1) and (2) for the IO monad.

Example 7. Consider the well-known state monad:

$$\begin{aligned} &\mathbf{newtype}\ \mathsf{State}\ \sigma\ \alpha = \mathsf{State}\ (\sigma \to (\alpha, \sigma)) \\ \\ &\mathbf{instance}\ \mathsf{Monad}\ (\mathsf{State}\ \sigma)\ \mathbf{where} \\ &\quad \mathit{return}\ a = \mathsf{State}\ (\lambda s \to (a, s)) \\ &\quad \mathsf{State}\ g \gg\!= k = \mathsf{State}\ (\lambda s \to \mathbf{let}\ (a, s') = g\ s\ \mathbf{in} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{case}\ k\ a\ \mathbf{of}\ \mathsf{State}\ g' \to g'\ s') \end{aligned}$$

Intuitively, this monad extends the reader monad by not only allowing a computation to depend on an input state, but also to transform the state to be passed to a subsequent computation. A natural question now is whether being a specific state transformer that actually corresponds to a read-only computation is an invariant that is preserved when computations are combined. That is, given some closed type τ and $l :: [\mathsf{State}\ \tau\ \mathsf{Int}]$ such that for every element $\mathsf{State}\ g$ in l, $\mathit{snd} \circ g = \mathit{id}$, is it the case that for every $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, also $f\ l$ is of the form $\mathsf{State}\ g$ for some g with $\mathit{snd} \circ g = \mathit{id}$?

The positive answer is provided by Theorem 3 with the following Monad-morphism:

$$\begin{aligned} &h :: \mathsf{Reader}\ \tau\ \alpha \to \mathsf{State}\ \tau\ \alpha \\ &h\ (\mathsf{Reader}\ g) = \mathsf{State}\ (\lambda s \to (g\ s, s)) \end{aligned}$$

4.4 Effect Abstraction

As a final statement about the type $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, we would like to show that we can abstract from some aspects of the effectful computations in the input list if we are interested in the effects of the final result only up to the same abstraction. For conveying between the full effect space and its abstraction, we again use Monad-morphisms.

Theorem 4. *Let* $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ *and let* $h :: \kappa_1\ \alpha \to \kappa_2\ \alpha$ *be a* Monad*-morphism. Then* $h \circ f_{\kappa_1}$ *gives the same result for any two lists of same length whose corresponding elements have the same* h*-images.*

Example 8. Consider the well-known exception monad:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Monad}\ (\mathsf{Either\ String})\ \textbf{where}\\&\quad \mathit{return}\ a = \mathsf{Right}\ a\\&\quad \mathsf{Left}\ \mathit{err} \gg\!= k = \mathsf{Left}\ \mathit{err}\\&\quad \mathsf{Right}\ a \gg\!= k = k\ a\end{aligned}$$

We would like to argue that if we are only interested in whether the result of f for some input list over the type Either String Int is an exceptional value or not (and which ordinary value is encapsulated in the latter case), but do not care what the concrete error description string is in the former case, then the answer is independent of the concrete error description strings potentially appearing in the input list. Formally, let $l_1, l_2 :: [\mathsf{Either\ String\ Int}]$ be of same length, and let corresponding elements either be both tagged with Left (but not necessarily containing the same strings) or be identical Right-tagged values. Then for every $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, $f\ l_1$ and $f\ l_2$ either are both tagged with Left or are identical Right-tagged values. This statement holds by Theorem 4 with the following Monad-morphism:

$$\begin{aligned}&h :: \mathsf{Either\ String}\ \alpha \to \mathsf{Maybe}\ \alpha\\&h\ (\mathsf{Left}\ \mathit{err}) = \mathsf{Nothing}\\&h\ (\mathsf{Right}\ a)\ = \mathsf{Just}\ a\end{aligned}$$

5 The Full Paper

The full version of this paper, containing more details, related work, and to demonstrate the broader scope beyond dealing with monads, an application of the proposed approach to transparently introducing difference lists into a program (endowed with a neat and general correctness proof), appears as [5].

References

1. J.C. Mitchell and A.R. Meyer. Second-order logical relations (Extended abstract). In *Logic of Programs, Proceedings*, volume 193 of *LNCS*, pages 225–236. Springer-Verlag, 1985.
2. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
3. S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Principles of Programming Languages, Proceedings*, pages 71–84. ACM Press, 1993.
4. J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
5. J. Voigtländer. Free theorems involving type constructor classes. In *International Conference on Functional Programming, Proceedings*. ACM Press, 2009.
6. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
7. P. Wadler. The essence of functional programming (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 1992.
8. 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.

Strict Observations

Jan Christiansen
Department of Computer Science
CAU Kiel, Germany
jac@informatik.uni-kiel.de

Abstract

Call-by-name is known to be an optimal evaluation strategy with respect to termination. In practice you can only benefit from this result if functions are not unnecessarily strict. Often it is not trivial to implement a function in a least strict way. In the following we use the lazy functional programming language Haskell.

Consider a data type for Peano Numbers. A Peano Number is either zero or the successor of another Peano Number.

$$\textbf{data}\ \mathit{Peano} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Peano}$$

We define the multiplication of two numbers by means of the addition *plus*.

$$\mathit{mult} :: \mathit{Peano} \to \mathit{Peano} \to \mathit{Peano}$$
$$\mathit{mult}\ \mathit{Zero}\ y = \mathit{Zero}$$
$$\mathit{mult}\ (\mathit{Succ}\ x)\ y = \mathit{plus}\ y\ (\mathit{mult}\ x\ y)$$

Furthermore we define an infinite Peano number which is an infinite sequence of successors.

$$\mathit{infinity} :: \mathit{Peano}$$
$$\mathit{infinity} = \mathit{Succ}\ \mathit{infinity}$$

Thanks to non-strictness the evaluation of *mult Zero infinity* yields *Zero*. But the evaluation of *mult infinity Zero* does not terminate. We can improve *mult* with respect to strictness and therefore termination. Consider the following implementation of the multiplication.

$$\mathit{mult}' :: \mathit{Peano} \to \mathit{Peano} \to \mathit{Peano}$$
$$\mathit{mult}'\ \mathit{Zero}\ y = \mathit{Zero}$$
$$\mathit{mult}'\ (\mathit{Succ}\ \_)\ \mathit{Zero} = \mathit{Zero}$$
$$\mathit{mult}'\ (\mathit{Succ}\ x)\ y = \mathit{plus}\ y\ (\mathit{mult}'\ x\ y)$$

For all total arguments *mult* and mult' yield the same results. But the evaluation of mult' *infinity Zero* yields *Zero*. Note that it is not trivial to check whether mult' is less strict than *mult* as the two functions can still be incomparable.

This is not an artificial example. The paper "Declaring Numbers" [1] introduces lazy natural numbers by a binary representation. As a motivating example they present a multiplication of Peano Numbers which is implemented in the exact same manner as *mult*. Furthermore the library "Numbers"[1] provides an implementation of Peano Numbers which implements the multiplication in the same manner as *mult* . Both paper and library care about least strict implementations. This demonstrates that it is not even trivial to implement a simple function like the multiplication of Peano Numbers in a least strict way. On the other hand you can only benefit from non-strict programming languages if functions are least strict.

Olaf Chitil first presented the idea of least-strictness and implemented a tool called StrictCheck [2]. The tool was supposed to check whether a function is least strict. But it has some shortcomings. In particular it does not consider sequentiality. That is, it suggests implementations that are not implementable in a sequential language. In the long term we aim at a lightweight tool for checking whether a function is least strict that does consider sequentiality.

References

[1] Bernd Brassel, Sebastian Fischer, and Frank Huch. Declaring numbers. In *Proc. of the 16th International Workshop on Functional and (Constraint) Logic Programming WFLP 2007*, 2007.

[2] Olaf Chitil. Promoting non-strict programming. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL 2006*, pages 512–516, Budapest, Hungary, September 2006. Eotvos Lorand University.

[1] available via hackage (`http://hackage.haskell.org`)

Automatische Testfallerzeugung auf Basis der Überdeckungsanalyse*

Tim A. Majchrzak und Herbert Kuchen

Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster
Münster, Germany
{tima, herbert}@wi.uni-muenster.de

Zusammenfassung Wir stellen ein Werkzeug für die automatische Generierung von Testfällen vor. Es führt Java Bytecode symbolisch aus, um Ausführungspfade durch ein Programm zu finden. Dazu nutzt es Constraint Solving, die Erzeugung von Entscheidungspunkten und Backtracking. Da die Zahl der gefundenen Testfälle sehr hoch werden kann und die meisten von ihnen redundant sind, regen wir an, Testfälle anhand ihres Beitrags zur globalen Kontroll- und Datenflussüberdeckung zu eliminieren. Neben den dazu benötigten Techniken zeigen wir experimentelle Ergebnisse, um die Machbarkeit des Ansatzes zu demonstrieren.

1 Einführung

Software zu testen ist mühevoll und teuer. Komponententests (*unit tests*) haben im Rahmen des Testprozesses eine hohe Bedeutung. Sie bieten die Chance, bereits in frühen Entwicklungsphasen eine wesentliche Zahl von Fehlern zu finden. Entwickler und Tester schreiben sie in der Regel von Hand. Dies ist eine anspruchsvolle, häufig aber auch als langweilig angesehene Tätigkeit. Besonders schwierig ist das Schreiben von Komponententests, die eine bestimmte Überdeckung des Quelltexts garantieren. Idealerweise sollten Komponententests ohne oder nur mit geringem Eingriff durch Tester erstellt werden. Daher beschäftigen wir uns mit Möglichkeiten zur automatischen Generierung.

Unser Werkzeug Muggl (**Mu**enster **g**enerator of **gl**ass-box test cases) basiert auf GlassTT [2], das an unserem Institut entwickelt wurde. Aufgrund fundamentaler Änderung wurde Muggl von Grund auf neu geschrieben. Es nutzt nur den Constraint Solver von GlassTT, der sich als sehr leistungsfähig erwiesen hat. Durch die Änderungen gehen wir auf die Probleme ein, die im Zusammenhang mit der symbolischen Ausführung aufgetreten sind. Gleichzeitig nutzen wir die Vorzüge. Neben dem eigentlichen Beitrag diskutieren wir in diesem Artikel weitere Vorzüge gegenüber dem alten Werkzeug.

Muggl führt Bytecode aus class-Dateien, die z. B. vom Java Compiler `javac` erzeugt wurden, symbolisch aus. Dazu nutzt es eine symbolische Implementierung der Java Virtual Machine (*JVM*) [3], die Eingabeparameter von Methoden als logische Variablen behandelt. Bedingte Sprünge und andere Instruktionen, die Einfluss auf den Kontrollfluss haben, führen bei Ausführung zur Erzeugung von Entscheidungspunkten (*Choice Points*). Muggl versucht Parametersätze für die getesteten Methoden zu finden,

* Diese Arbeit ist eine angepasste Übersetzung des unter [1] erscheinenden Artikels.

indem es mithilfe eines Suchalgorithmus mögliche Pfade durch das Programm ermittelt. Jeder gefundene Parametersatz und das zugehörige Ergebnis stellen einen Testfall dar. Da die Zahl der Pfade durch ein Programm typischerweise unbegrenzt ist, darf nur ein kleiner endlicher Satz repräsentativer Testfällen ausgewählt werden. Wir schlagen vor, die Auswahl aufgrund der *Kontroll-* und *Datenflussüberdeckung* vorzunehmen.

Dieser Artikel ist wie folgt strukturiert. Abschnitt 2 stellt Muggl vor. In Abschnitt 3 diskutieren wir die Erzeugung von Kontroll- und Datenflussinformationen und ihre Überdeckung. Der Algorithmus zur Elimination wird in Abschnitt 4 vorgestellt, gefolgt von experimentellen Ergebnissen in Abschnitt 5. In Abschnitt 6 zeigen wir verwandte Arbeiten und schließen mit Abschnitt 7.

2 Muggl

2.1 Grundidee und Architektur

Anstelle von Quelltexten verarbeitet Muggl class-Dateien mit Java Bytecode [3]. Dies hat zwei wesentliche Vorteile. Erstens werden Optimierungen durch den Compiler einbezogen. Zweitens können zahlreiche Sprachen zu Java Bytecode kompiliert werden, die sich dann mit *einem* Werkzeug testen lassen. Java Bytecode ist konzeptionell ähnlich zur Microsoft Common Intermediate Language (CIL) [4] und – mit Einschränkungen – zu Assemblersprache. Eine Erweiterung von Muggl auf andere Sprachen ist daher eine Option (für Einschränkungen vgl. [5]). Fehlende Quelltexte sind unproblematisch, da sie mit dem `LineNumberTable`-Attribut mit Bytecode verknüpft werden können [3].

Die dynamische Analyse eines Programms erfordert die Ausführung des Codes in einer Laufzeitumgebung. Aufgrund der anspruchsvollen Arbeitsweise von Muggl, die symbolische Ausführung und Manipulationstechniken beinhaltet, fiel die Entscheidung gegen eine existierende VM. Zwar hätte auf die JVM von SUN zurückgegriffen werden können, Muggl nutzt aber seine eigene virtuelle Maschine. Sie implementiert die JVM-Spezifikationen für Java 1.4 mit Ausnahme von Threading. Die Änderungen von Java 1.5 und 1.6 werden derzeit eingearbeitet.

Muggls Architektur teilt sich in GUI und Ausführungskern (Abb. 1). Die GUI dient der einfachen Bedienung und bietet unter anderem Einstellungsdialoge sowie einen Klassenbrowser. Wenn die Ausführung gestartet wird, zeigt sie Laufzeitinformationen an. Alternativ steht eine *Schritt-für-Schritt*-Umgebung zur Verfügung, mit der etwa der Status des *Frame-Stacks* oder die Überdeckung nachvollzogen werden können. Zur Testfallerzeugung ist diese Umgebung nicht erforderlich. Sie kann aber sehr dabei helfen, Einstellungen zu optimieren und ein Programm besser zu verstehen.

Der Kern von Muggl besteht aus einer symbolischen virtuellen Maschine und zusätzlichen Komponenten. Neben der symbolischen ist auch die „normale“ Ausführung möglich. Ein eigener Bytecode-Parser wird zum Lesen und Manipulieren von class-Dateien genutzt. Anders als sein Vorgänger und ähnliche Werkzeuge (siehe Abschnitt 6) nutzt Muggl einen *nativen Wrapper*. Zahlreiche Klassen des Java Runtime Environment (JRE) nutzen Methoden, die `native` sind, also eine plattformspezifische Funktionalität bieten. Muggl fängt Zugriffe auf solche Methoden ab und leiten sie an das JRE der Plattform weiter. Dabei werden Daten aus Muggls interner Darstellung umgewandelt.

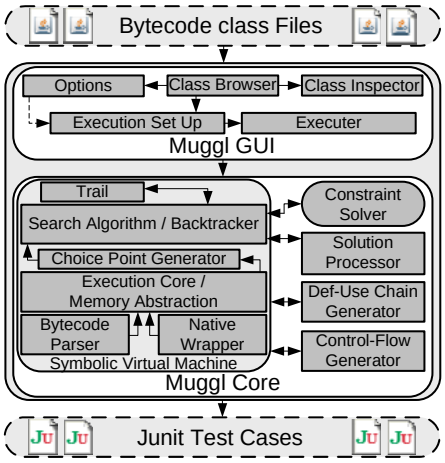


Abbildung 1. Architektur von Muggl

Obwohl nur nicht-symbolische Daten umgewandelt werden können, ist somit die Ausführung der meisten Programme möglich. Alternativ können native Methoden durch Java-Methoden simuliert werden. Dies erfolgt z. B. für Methoden von `System.out`.

Ebenfalls wichtig sind der Lösungsprozessor, der aus Testfällen Lösungen im Sinne logischer Programmierung erstellt, sowie die Kontroll- und Datenflussgeneratoren.

2.2 Symbolische Ausführung und Constraint Solving

Wenn Muggls VM Bytecode symbolisch ausführt, werden Eingabeparameter als logische Variablen behandelt. Sie werden an Terme gebunden, die aus Konstanten, Operatoren und Variablen bestehen. Ein Beispiel für eine statische Methode zur Berechnung des größten gemeinsamen Teilers ist in Abb. 2 gegeben. Die Zahlen sind Instruktionsnummern (*Offsets*). Die dritten Spalte zeigt betroffenen Variablen bzw. Quelltext-Anweisungen. Zwei lokale Variablen werden auf den Stack gelegt und verglichen. Ist die erste kleiner, wird die Ausführung mit der nächsten Instruktion fortgesetzt. Ansonsten erfolgt ein Sprung zur dritten `iload_0`-Instruktion (Offset 11). Das Beispiel sollte größtenteils selbsterklärend sein. `istore_x` nimmt Elemente vom Stack um sie in lokale Variablen zu speichern; die Instruktion an Offset 23 ruft die Methode rekursiv auf.

Um zu entscheiden, welcher von zwei Ausdrücken kleiner ist, kann das Werkzeug *Constraints* lösen. Am bedingten Sprung `if_icmpge` stehen zwei Wege durch den Suchbaum aller möglicher Pfade durch den Code zur Wahl. Muggl prüft alle Constraints, um den zu durchsuchenden Ast zu bestimmen. In Abb. 2 könnten wir bei Erreichen von Offset 02 z. B. wissen, dass der Methodenaufruf $ggt(x, x * 2)$ war. Folglich ist die erste lokale Variable kleiner als die zweite und nur ein Ast muss durchsucht werden.

Constraint Solving wird für echte Probleme sehr kompliziert. An die Stelle einzelner linearer treten beliebige nicht-lineare Constraints. Muggl nutzt zu deren Lösungen einen eigens für die Aufgabe entwickelten Constraint Solver [6]. Er wurde bereits in GlassTT genutzt und hat sich als leistungsfähig erwiesen. Die Lösung linearer und nicht-linearer Constraint ist möglich und alle von der JVM unterstützen primitive Da-

tentypen werden abgebildet. Die Lösung erfolgt durch diverse Solver, z. B. mithilfe des *Fourier-Motzkin-Algorithmus* oder des *Bisektionsverfahrens*. Durch den Solver wird der Suchraum im Vergleich zu Ansätzen ohne einen solchen dramatisch verkleinert [7].

Jeder gefundene Pfad entspricht einem Testfall. Die Ableitung von Testfälle aus der symbolischen Ausführung ist sehr ähnlich wie in [2] erläutert. Da die Breitensuche zu viel Speicher benötigt, nutzt Muggl die Tiefensuche. Immer, wenn eine `return`-Instruktion erreicht wird und keine Frames mehr auf dem Stack der VM zur Ausführung bereitliegen, wird eine Lösung erzeugt. Dafür werden logische Variablen auf konkrete Werte abgebildet. Danach erfolgt das *Backtracking*, wie es aus logischen Programmiersprachen wie *Prolog* bekannt ist. Es versetzt die VM in den Zustand, der an dem letzten Punkt, an dem eine Entscheidung getroffen wurde, herrschte. Solche *Choice Points* werden etwa bei bedingten Sprüngen erzeugt. Jeder verfügt über einen *Trail* und speichert alle Änderungen am Zustand seit seiner Initialisierung. Zum Zurücksetzen wird der Trail rückwärts (*last-in-first-out*) bearbeitet. Für den erreichten Entscheidungspunkt prüft Muggl, ob unbesuchte Äste vorhanden sind und ob diese das Constraint-System verletzten. Ist zumindest ein Ast erreichbar, wird die Ausführung dort fortgesetzt. Ansonsten erfolgt rekursives Backtracking. Sobald keine Entscheidungen mehr übrig sind, stoppt die Ausführung. Alle möglichen Testfälle wurden gefunden.

Muggl verfügt über umfangreiche Funktionen, um Komplexität zu reduzieren und zu kontrollieren. Es bietet vielzählige Einstellungsmöglichkeiten, etwa Abbruchkriterien. Eine diesbezügliche Vertiefung geht aber über den Rahmen dieses Artikels hinaus.

3 Flussanalyse

3.1 Grundlagen und Statische Generierung

Es gibt verschiedene Ansätze, den Kontroll- (*KF*) und insbesondere den Datenfluss von Programmen zu analysieren. Anstatt zu erörtern, welcher dieser Ansätze für welche Art von Problem geeignet ist, stellen wir unsere Definition vor. Eine Diskussion für Softwaretest nützlicher Überdeckungskriterien findet sich z. B. in [8].

Wenn Berechnungen mit mehreren Threads ausgeschlossen werden, stellt ein Ausführungspfad eine sequentielle Liste von Instruktionen dar. Jede Instruktion bildet einen Knoten im Kontrollflussgraphen. Übergänge zwischen Instruktionen bilden Kanten. Verzweigende Instruktionen haben dabei mehr als eine Kante. Graphen können auch Zyklen enthalten. Während der symbolischen Ausführung bestimmt Muggl die Kontrollflussüberdeckung des betrachteten Pfades. Jede Kante des Graphen wird durch zwei Integer-Werte beschrieben, welche die Instruktions-Offsets der durch die Kante verbundenen Knoten wiedergeben. Instruktions-Offsets sind Indizes des Byte-Arrays, der für jede Methode einer class-Datei als Teil des *Code*-Attributs bereitgestellt wird [3].

Die Überdeckung des KF ist ein mögliches Ziel von Komponententests. Allerdings ist sie in der Praxis nicht hinreichend, da viele Fehler nicht gefunden werden. Aus diesem Grund regen wir an, auch den Datenfluss zu betrachten. Es gibt dazu diverse Vorgehensweisen. Wir schlagen die Nutzung kompletter Definitions-Nutzungs-Ketten (*DU*, def-use chains) vor. Sie sind wie folgt definiert:

$$X\text{: Variable}, S, S', S''\text{: Instruktionen}$$
$$def(S) := \{X | S \text{ schreibt } X\},\ use(S) := \{X | S \text{ liest } X\}$$

Falls $X \in def(S) \bigcup use(S')$ und $X \notin def(S'')$ für alle S'' auf einem Pfad von S zu S', formt (X, S, S') eine def-use-Kette. Die Überdeckung wird nur für volle DU-Paare erfasst. Jede Kette wird durch zwei Instruktions-Offsets sowie eine Variable repräsentiert.

Es ist wichtig, ob KF-Graphen und DU-Ketten statisch vor der Ausführung oder dynamisch während dieser erzeugt werden. Zur Erfassung der Überdeckung sind zwei Schritte nötig. Zunächst werden existierende Kanten und mögliche Ketten bestimmt: *Vorbereitung*. Dann findet die Ausführung statt, bei der Kanten und Ketten überdeckt werden: *Überdeckung*. Zu bedenken sind daher Berechnungszeit und Speicherverbrauch für die Vorbereitung und die Überdeckung sowie Auswirkungen auf die Elimination.

Die offensichtliche Idee ist die Kombination der Vorbereitung mit der Ausführung (*on-the-fly*). Die benötigte Rechenzeit ist gering und die Überdeckungserfassung günstig, da jede gefundene Kante bzw. Kette überdeckt ist. Der Speicherbedarf für die Vorbereitung ist vernachlässigbar. Die Speicherung gefundener Kanten und Ketten kann aber sehr umfangreich werden und für hunderttausende Testfälle unpraktikabel sein.

Im statischen Fall hingegen ist zusätzliche Rechenzeit für die Vorbereitung nötig. Auch die Erfassung von gefundenen Kanten und Ketten während der Ausführung erfordert etwas mehr Rechenzeit. Sie kann allerdings durch Nutzung einer Hash-Tabelle begrenzt werden. Die Speichernutzung ist bedeutend geringer. Da alle Kanten und Ketten vorher bekannt sind, müssen diese nicht für jeden gefundenen Testfall gespeichert werden. Vielmehr reicht ein Array boolscher Werte, das alle Kanten und Ketten abbildet. Die Überdeckungsanalyse erfolgt mithilfe der Hash-Tabelle. Für jede gefundene Kante bzw. Kette wird der Wert des zugehörigen Arrays-Index auf *wahr* gesetzt. Ein weiterer Unterschied zeigt sich bei der Elimination: Im dynamischen Fall muss jede Kante und jede Kette jedes Testfalls verglichen werden, um redundante Testfälle zu löschen. Anders ist nicht festzustellen, ob die gleichen Elemente überdeckt wurden. Daher ist eine aufwendige Vorverarbeitung nötig, ohne die das Eliminieren extrem langsam wäre. Im statischen Fall ist dieser Vorgang deutlich einfacher und günstiger; es reicht der Vergleich der Arrays. Dieselben Indizes bedeuten dieselbe Überdeckung. Durch Optimierungen kann die Geschwindigkeit zusätzlich erhöht werden (siehe Abschnitt 4).

Wir haben unsere Arbeit mit der in [7] verglichen, welche dynamische Generierung nutzt. Die Ausführungszeit steigt dort rapide mit der Zahl der gefundenen Testfälle. Wir haben uns daher gegen die dynamische Generierung entschieden.

3.2 Kontrollflussüberdeckung

Für jede ausgeführte Methode erzeugt Muggl den Kontrollflussgraphen. Interprozedurale Graphen müssen dabei nicht erstellt werden, da Methodenaufrufe unbedingte Sprünge darstellen. Die Ausführung kommt immer zu ihnen zurück, entweder durch einen Rücksprung oder durch eine von einer aufgerufenen Methode geworfene Exception. Der gesamte Graph kann durch Kombination der Graphen aller Methoden konstruiert werden. Dies erfordert allerdings eine zusätzliche Analyse, da nicht für alle virtuellen Methodenaufrufe die Zielmethoden statisch ermittelt werden können.

Bytecode-Instruktionen werden nach ihrem Übergangsverhalten kategorisiert:

- Bedingte Sprünge führen entweder zu einem Sprung, oder die Ausführung wird mit der nächsten Instruktion fortgesetzt. Dies trifft auf `if_xcmp<cond>`, `if<cond>`, `ifnonnull` und `ifnull` zu. Die Zahl der Kanten ist zwei.

- Unbedingte Sprünge haben genau eine Nachfolgeinstruktion, allerdings nicht notwendigerweise die direkt folgende. Dies betrifft `goto`, `goto_w`, `jsr` und `jsr_w`. Sie bilden eine Kante im Graphen.
- Die Switches `lookupswitch` und `tableswitch` haben diverse Sprungziele und daher mehrere Kanten.
- Die fünf Instruktionen für Methodenaufrufe (`invokez`) unterbrechen den Kontrollfluss. Nach dem erfolgreichen Aufruf wird die ihnen direkt folgende Instruktion ausgeführt. Sie bilden eine Kante.
- Einige Instruktionen lösen nie Sprünge aus. Als nächstes wird die direkt folgende Instruktion ausgeführt. Beispiele sind arithmetische Instruktionen wie `iadd` und `imul`, logische Instruktionen wie `iand` und `ior`, `push` und `pop`, Instruktionen wie `iconst_1`, die Konstanten auf den Stack legen, und Instruktionen zum Laden und Speichern von Variablen wie `iload` und `istore`.

Viele Instruktionen können Exceptions werfen. Ausnahmen sind ein integraler Bestandteil von Java und Java Bytecode und müssen im Kontrollflussgraphen abgebildet werden. Während die geworfene Exception die einzige Kante der `athrow`-Instruktion darstellt, hat jede Instruktion, die Exceptions auslösen kann, zugehörige Kanten. Falls Exceptions lokal gefangen werden, stellt dies eine Kante zwischen der werfenden und der behandelnden Instruktion dar. Ansonsten wird der Methodenaufruf beendet.

Im Kontrollflussgraphen sind auch Schnittstellen abgebildet. Aufrufinstruktionen führen zur Ausführung einer Methode. Sobald ein Rücksprung ausgeführt oder eine Exception nicht behandelt wird, setzt sich die Ausführung bei der Methode fort, die zu dem obersten *Frame* des VM-Stacks gehört (oder endet, falls dieser leer ist). Um das *Exception-Handling* korrekt im Graphen abzubilden, versuchen wir, die Typen der geworfenen Exceptions zu bestimmen. Für die meisten Instruktionen ist dies statisch bekannt und muss nur für wenige während der Vorbereitung ermittelt werden. Bei Aufrufen ermitteln wir die aufzurufende Methode und prüfen, welche Exceptions sie deklariert. Jede Methode kann zudem Exceptions vom Typ `RuntimeException` bzw. eines ihrer Untertypen werfen. Danach werden alle Exceptions gesammelt, die innerhalb der Methoden geworfen werden können. Nachdem die direkt gefangenen entfernt wurden, stehen die zusätzlich von der Methode geworfenen Exceptions fest. Somit wird ein vollständiger Graph selbst für inkorrekt kompilierte Klassen erstellt.

Da die symbolische Ausführung ein Backtracking erfordert, erstellt Muggl Informationen, mit denen sich auch die Überdeckungsinformationen zurücksetzen lassen.

3.3 Datenflussüberdeckung

Def-use-Ketten auf Ebene des Bytecode zu ermitteln ist deutlich schwieriger als auf Ebene des Quelltexts. Das Laden einer lokalen Variable ist notwendig, aber nicht hinreichend für eine Nutzung, da geladene Variablen nicht in jedem Fall genutzt werden. Daher muss das Laden, Definieren und Benutzen von Variablen überwacht werden. Interprozedurale Graphen müssen explizit erstellt werden.

Sei $PC = \{0, \ldots, 65535\}$ die Menge der Instruktions-Offsets, $M = \{m_0, m_1, \ldots, m_n\}$ die Menge der Methodenbezeichner, $V = \{v_0, v_1, \ldots, v_o\}$ die Menge lokaler Variablen, $H = \{h_0, h_1, \ldots, h_p\}$ die Menge der Objektbezeichner, $S = \{s_0, s_1, \ldots, s_q\}$ die Menge

der Bezeichner statisch initialisierter Klassen, $A = \{a_0, a_1, \ldots, a_r\}$ die Menge der Attribute, und $I = \{i_0, i_1, \ldots, i_t\}$ die Menge der Positionen auf dem Methoden-Stack $\sigma \in I^*$. Seien $Addr = (M \times V) \cup (H \times A) \cup (S \times A) \cup (M \times I)$ Adressen von Variablen und sei $Instr = M \times PC$ die Position von Instruktionen. Jede DU-Kette δ kann als $\delta = (\alpha, \beta, \gamma)$ mit $\alpha \in Addr$ und $\beta, \gamma \in Instr$ beschrieben werden.

Bereite die DU-Erzeugung vor, indem eine initial leere Menge $C = \emptyset$ aus DU-Ketten und ein Stack *st* von Erzeugungszweigen angelegt wird. Jeder Zweig π besteht aus einem Offset *pc* und einer Menge D gefundener Definitionen, also $\pi = (pc, D)$ mit $D = \{(\alpha_0, \beta_0), \ldots, (\alpha_u, \beta_u)\}$, $\alpha_j \in Addr$ und $\beta_j \in Instr$ für $j = 0, \ldots, u$, $u \in \mathbb{N}$. Lege den ersten Zweig $\pi = (0, \emptyset)$ auf *st*. Erzeuge eine Menge geladener Variablen $\lambda \subseteq Addr$. Der Algorithmus zum Finden der DU-Ketten stellt sich wie folgt dar:

Während *st* nicht leer ist:

1. Hole den obersten Zweig $\pi = (pc, D)$ von *st*. Betrachte die Instruktion an Offset *pc* in Methode *m*. Falls sie eine Variable $\alpha \in Addr$ lädt, füge α zu λ hinzu.
2. Falls eine Definition (α, β) gefunden wird (z. B. Abb. 2, Offset 06, `istore_2`), füge sie zu D hinzu.
3. Falls eine Benutzung (α, γ) gefunden wird (z. B. Abb. 2, Offset 02, `if_ifcmpg`), eine dazugehörige Definition $(\alpha, \beta) \in D$ existiert und $\alpha \in \lambda$, erzeuge die DU-Kette $\delta = (\alpha, \beta, \gamma)$ und füge sie zu C hinzu.
4. Entferne jeden zu einer genutzten Variable gehörenden Eintrag α von λ.
5. Verzweige bei bedingten Sprüngen. Dupliziere dazu den bearbeiteten Zweig π und setze sein *pc* auf das Sprungziel. Folglich werden beide Zweige einzeln behandelt. Bei Methodenaufrufen erzeuge einen Zweig π mit $D = \{(\rho_0, \tau_0), \ldots, (\rho_w, \tau_w)\}$, $\rho_j \in M \times V$ und $\tau_j \in M \times PC$, $j = 0, \ldots, w$ so dass alle Argumente als definiert gesetzt sind. Deren Anzahl w wird durch die Signatur der Methode *m* bestimmt.
6. Wenn eine bekannte DU-Kette δ gefunden wird, prüfe (mithilfe einer boolschen Variable), ob Definition gefunden wurden, seit δ das letzte mal gefunden wurde. In diesem Fall, oder falls keine DU-Ketten gefunden wurden, aktualisiere das *pc* von π und lege es wieder auf *st*. Somit werden Endlosschleifen verhindert; der Algorithmus terminiert, sobald alle DU-Ketten gefunden wurden. *st* und alle Hilfsstrukturen werden dann entfernt, während nur die Menge C der DU-Ketten erhalten bleibt.

Die einzigen Ladeinstruktionen sind `xload` und `gety`. Definierende Instruktionen sind z. B. `xstore`, `puty` und `invokez`. Zahlreiche Instruktionen nutzen Variablen, so etwa arithmetische und logische Operatoren, bedingte Sprünge, Methodenaufrufe, Switches und Rücksprünge. Alle Instruktionen, die Werte auf den lokalen Stack legen, definieren Zwischenergebnisse. Zur Optimierung können *solche* DU-Ketten, die bei Erreichen der zugehörigen Definition sicher erreicht werden, weggelassen werden. Ein Beispiel ist die Kette $((ggt, i_0), (ggt, 13), (ggt, 14))$ in Abb. 2.

Aliasing ist unproblematisch, da Referenzen einschließlich Arrays bei Zugriff aufgelöst werden. Sie haben eine eindeutige, fortlaufend vergebene Identifikationsnummer.

Überdeckung werden überwacht, indem für jede ausgeführte Instruktion der *pc* geprüft wird. Gefundene Definitionen werden markiert. Eine DU-Kette wird gefunden, wenn eine Benutzung festgestellt wurde und die zugehörige Definition markiert ist. Analog zum Kontrollfluss ist das Zurücksetzen des Überdeckungsstatus möglich.

off	Bytecode	Java	load	def	use	off	Bytecode	Java	load	def	use
	Methodenaufruf			v_0, v_1		11	**iload_0**	m	v_0		
00	**iload_0**	m	v_0			12	**iload_1**	n	v_1		
01	**iload_1**	n	v_1			13	**isub**	$m - n$	~~v_0~~, ~~v_0~~, i_0	i_0	v_0, v_1
02	**if_ifcmpg 09**	if $(m < n)$	~~v_0~~, ~~v_1~~		v_0, v_1	14	**istore_2**	$r = m - n$	~~i_0~~	v_2	i_0
						15	**iload_2**	r	v_2		
05	**iload_0**	m	v_0			16	**ifne** 17	if $(r == 0)$	~~v_2~~		v_2
06	**istore_2**	$t = m$	~~v_0~~	v_2	v_0	19	**iload_1**	n	v_1		
07	**iload_1**	n	v_1			20	**ireturn**	return n	~~v_1~~		v_1
08	**istore_0**	$m = n$	~~v_1~~	v_0	v_1						
09	**iload_2**	t	v_2			21	**iload_1**	n	v_1		
10	**istore_1**	$n = t$	~~v_2~~	v_1	v_2	22	**iload_2**	r	v_2		
						23	**invokestatic**	ggt(n, r)	~~v_1~~, ~~v_2~~, i_1	i_1	v_1, v_2
						26	**ireturn**	return	~~i_1~~		i_1

Abbildung 2. Erzeugung der Def-use-Ketten für GgT. Zur Einfachheit wurden die Adressen von Definitionen und Benutzungen weggelassen.

4 Elimination der Testfälle

Die symbolische Ausführung in Muggl führt zur Erzeugung von Testfällen. Deren Zahl kann – abhängig von den gewählten Einstellungen – sehr gering, aber auch extrem hoch sein. Auch der Grad an Redundanz kann stark variieren. Es wird daher eine effektive und praktikable Strategie zur Elimination benötigt.

Beim Ende der Ausführung ist die Kontroll- und Datenflussüberdeckung für alle Testfälle bekannt, da Muggl die individuelle Überdeckung eines jeden Testfalls speichert. Gesucht wird *die* (möglichst) minimale Menge der Testfälle, die alle global überdeckten KF-Kante und DU-Ketten überdeckt. Jeder Testfall, der nicht zu dieser Menge gehört, wird als *redundant* angesehen. Folglich handelt es sich um ein *NP-vollständiges* Mengenüberdeckungsproblem (*set-covering*), das eine approximative Lösung erfordert. Der übliche Ansatz ist die Nutzung eines gierigen (*greedy*) Algorithmus, der eine Menge liefert, die nicht wesentlich größer als die optimale Menge ist [9]. Er lässt sich so implementieren, dass er in polynomieller Zeit läuft. Die Überdeckungsdaten werden als Arrays boolscher Werte abgebildet. Durch adäquate Datenstrukturen, Zwischenspeicherungen und die dynamische Entfernung von Testfällen, die keine global noch nicht überdeckten Kanten und Ketten mehr überdecken, ist eine Beschleunigung möglich.

Es ist vorteilhaft, nur eine geringe Zahl von Testfällen übrig zu behalten. Die Ausführungszeit des Satzes von Testfällen sinkt erheblich, was Regressionstests sehr erleichtert. Zudem sollten Testfälle für Menschen nachvollziehbar sein. Während die Parameter nur einen geringen Einfluss auf die Verständlichkeit haben, wächst die Zeit zum Durchdringen eines ganzen Satzes fast linear mit der Zahl der enthaltenen Testfälle.

5 Experimentelle Ergebnisse

Um Muggl zu erproben haben wir diverse experimentelle Läufe vorgenommen. Bitte beachten Sie, dass die Zahl der gefundenen Testfälle stark von den gewählten Einstellungen abhängt. Größere Anzahlen wurden mitunter bewusst erzeugt und entsprechen

Tabelle 1. Experimentelle Ergebnisse

Algorithmus	Laufzeit	Abbruch	Testfälle gefunden	eliminiert	übrig	Eliminations-zeit
Binäre Suche	0.14s	*Überdeckung*	60	57	3	<0.05s
Größter gemeinsamer Teiler	10.0s	Zeit	62	59	3	<0.05s
Größter gemeinsamer Teiler	30.0s	Zeit	85	82	3	<0.05s
Fibonacci	10.0s	Zeit	2	1	1	<0.05s
Mergesort	10.0s	Zeit	201	199	2	<0.05s
Quicksort	10.0s	Zeit	139	136	3	<0.05s
Quicksort	180.0s	Zeit	488	485	3	0.18s
Newton-Verfahren	0.19s	*Überdeckung*	29	24	5	<0.05s
Deutsche Bahn Tickets	10.0s	Zeit	28	24	4	<0.05s
Deutsche Bahn Tickets	300.0s	Zeit	28	24	4	<0.05s

nicht der Zahl an Testfällen, die *nötig* wäre. Die Algorithmen wurden ausgewählt, um nicht Muggl im Allgemeinen, sondern die Elimination zu prüfen. Das Testsystem war mit einer Intel Core 2 Duo E6600 CPU mit 2,4 GHz und 3 Gigabyte PC2-6400 Arbeitsspeicher ausgestattet. Als Betriebssystem kam MS Windows XP mit SP 3 und als Java SDK die Version 1.6.0_07 zum Einsatz. Die Heapgröße der VM betrug 512 Megabyte maximal und 256 Megabyte initial. Die Ergebnisse sind in Tab. 1 dargestellt.

Die folgenden Einstellungen wurden genutzt:

- Aufrufe nativer Methoden wurden durch den Wrappers an das JDK weitergereicht.
- Instruktionen wurden dynamisch optimiert. Muggl kann Instruktionen durch schnellere Instruktionen ersetzen, sobald sie das erste Mal ausgeführt wurden. Dies spart teure Berechnungen, etwa den Zugriff auf den *Konstantenpool*.
- Die Starttiefe der iterativen Tiefensuche war 5, der Erhöhungsschritt betrug 3.
- Die maximale Zahl an Schleifendurchläufen betrug 200. Bei Erreichen dieser Zahl wurde die Bearbeitung des aktuell durchsuchten Asts abgebrochen und das Backtracking gestartet. Auf diese Weise wird verhindert, dass Schleifen mit ungünstigen Grenzen lange durchsucht werden, ohne das dies zu Testfällen führt.
- Die Überdeckung wurde für die initiale und alle aufgerufenen Methoden desselben Pakets überwacht. Abgebrochen wurde bei kompletter globale Überdeckung.
- Eliminiert wurde im Hinblick auf die Kontroll- *und* Datenflussüberdeckung.

Neben kleinerer Algorithmen wie der binären Suche, GgT, Fibonacci, Mergesort, Quicksort und dem Newtonschen Näherungsverfahren haben wir ein größeren Verfahren getestet: Die Ticketberechnung der Deutschen Bahn (DB) ist nach einigen Vereinfachungen nur noch von 16 Parametern abhängig.

Da Arrays nicht symbolisch dargestellt werden können, nutzen wir einen Array-Generator. Dieser liefert zunächst `null` und ein leeres Array. Danach erzeugt er Arrays aufsteigender Länge, deren Elemente als logische Variablen initialisiert werden. Um sehr zahlreich Testfällen zu erhalten, haben wir den Generator auf die Erzeugung extrem vieler Arrays (`Integer.MAX_VALUE`, also $2^{31} - 1$) eingestellt und alle Abbruchkriterien außer der Laufzeit ausgeschaltet. Die Ergebnisse sind in Tab. 2 dargestellt.

Es lassen sich einige Schlüsse ziehen. Unser Ansatz ist praktikabel, da geeignete Testfälle für eine Vielzahl von Beispielen gefunden werden. Ihre Zahl ist nach der Elimination nahezu optimal. So verbleiben z. B. drei Testfälle, also die minimale Menge,

Tabelle 2. Zusätzliche experimentelle Ergebnisse

		Testfälle			
Algorithmus	Laufzeit	gefunden	eliminiert	übrig	Eliminationszeit
Binäre Suche	10.0s	13.530	13.526	4	0.13s
Binäre Suche	60.0s	79.676	79.672	4	0.5s
Binäre Suche	600.0s	374.883	374.879	4	3.56s
Quicksort	180.0s	1.158	1.154	4	<0.05s

für die binäre Suche. Der erste führt zu einer Durchsuchung der linken und rechten Hälfte des Such-Arrays und liefert dann die Position eines Elements. Der zweite Testfall ist eine nicht erfolgreiche Suche. Beim dritten wird die einzige mögliche Exception, eine `NullPointerException`, geworfen, da `null` anstelle eines Arrays übergeben wird.

Als Problem stellt sich der Abbruch dar. Nur bei wenigen Beispielen führt die Überdeckung zum Abbruch. Denn statisch gefundene, *mögliche* Kontrollflusskanten und DU-Ketten können aufgrund semantischer Einschränkungen häufig nicht überdeckt werden, selbst bei Überdeckung jeder *erreichbaren* Kante und Kette. Daher werden künstliche Kriterien wie die maximale Laufzeit bzw. die Rekursionstiefe genutzt. Nichts desto trotz werden in kurzer Zeit geeignete Testfälle gefunden. Daher steigt die Zahl der Testfälle mit einer Erhöhung der Laufzeit bei den meisten Beispielen nicht.

Die Elimination von Testfällen anhand ihrer Beitrags zur globalen Überdeckung hat sich als erfolgreich erwiesen. Unser Algorithmus approximiert erfolgreich das Minimum, die Effizienz ist dabei hoch. Auch wenn noch komplexere Beispiele getestet werden müssen, sind die Ergebnisse sehr gut. So wurden beispielsweise weniger als 4 Sekunden benötigt, um fast alle von 374.883 Testfällen zu eliminieren.

6 Ähnliche Arbeiten

Zur Kontroll- und Datenflussanalyse wurden zahlreiche Arbeiten veröffentlicht. Da die zugrundeliegenden Techniken bekannt sind, geben wir nur Hinweise auf Arbeiten im Zusammenhang mit Bytecode. Grundlagen zu Flussgraphen finden sich etwa in [10]. Eine detaillierte Einführung in die Erzeugung von Kontrollflussgraphen aus Java Bytecode wird von [11] gegeben. Mit dem *Eclipse*-Plugin *Control Flow Graph Factory* [12] lassen sich Graphen für Java-Quelltexte und Bytecode visualisieren. Die Datenflussanalyse von Bytecode wird vor allem im Rahmen der Verifikation [13] und Schwachstellenanalyse [14] behandelt. Während sogar eine US-Patent für eine Methode zur Erzeugung von Datenflussgraphen aus Java Bytecode existiert (#6233733), ist uns keine Methoden bekannt, die DU-Ketten statisch erzeugt und ihre Überdeckung dynamisch überwacht.

Zur symbolische Ausführung und der Erzeugung von Testdaten wurden diverse Arbeiten mit jeweils speziellen Schwerpunkt publiziert. Wir beschränken uns daher hier auf neuere Arbeiten; eine Gegenüberstellung mit älteren findet sich in [2]. Wie bereits angeführt, ist Muggl der – wesentlich erweiterte – Nachfolger von [2]. CyTest [7] setzt denselben Ansatzes für die funktional-logische Sprache *Curry* um. Es hat allerdings keinen Constraint Solver und erzeugt Überdeckungsdaten dynamisch. *IBIS* [15] führt Java Bytecode symbolisch aus und bindet einen externen Constraint Solver ein. Intern nutzt es allerdings eine *Prolog*-Repräsentation von class-Dateien. *Pex* [16] erzeugt Test-

fälle für Microsoft .NET-Applikationen und nutzt dazu symbolische Ausführung. Auch wenn es auf pfadgebundenem *Model Checking* basiert, ist es ähnlich zu Muggl. Allerdings nutzt es keine eigene VM, sondern die .NET Profiling API. Auch die Suchstrategie ist unterschiedlich. Mit *Z3* [17] nutzt Pex einen *Satisfiability Modulo Theory* (SMT) Solver, der sich wesentlich von dem in Muggl unterscheidet und Datenstrukturen wie Arrays unterstützt. Eine Elimination von Testfällen findet nicht statt.

Neben den eng verwandten Arbeiten gibt es viele Ansätze, die Überschneidungen mit Muggl in Teilen aufweisen. So lassen sich Komponententests auch mithilfe formaler Beweise erzeugen [18]. *Symstra* [19] und *ATGen* [20] sind Muggl in einigen Punkten ähnlich. Symstras Techniken entsprechen allerdings symbolischem Modell Checking, während ATGen sich auf eine ADA-Teilmenge bezieht. *Osmose* [5] wendet konzeptuell ähnliche Techniken auf Maschinencode an. Aufgrund der dabei herrschenden Komplexität gibt es allerdings zahlreiche Einschränkungen. In [21] wird die heuristische Testfallerzeugung durch evolutionäre Algorithmen beschrieben. Dabei wird die Überdeckung maximiert und vor allem auf DU-Ketten geachtet. Auch Arbeiten zu (Constraint) logischer Programmierung [22] weisen Ähnlichkeiten mit unserer Arbeit auf. Im Gegensatz zum systematischen Ansatz stehen zufallsbasierte Werkzeuge wie *QuickCheck*, die allerdings mitunter beachtliche Resultate erzielen [23]. Symbolische Ausführung kommt auch bei Model Checking zum Einsatz, vgl. z. B. [24]. Aufgrund des stark abweichenden Paradigmas ist ein Vergleich aber nicht zielführend. Für einen generellen Überblick zu automatisierter Testdatenerzeugung werden [25,26] empfohlen.

7 Fazit und zukünftige Arbeit

Wir haben Methoden zur Erzeugung von Kontroll- und Datenflussinformationen aus Java Bytecode vorgestellt. Wir empfehlen, Überdeckungsdaten zur Elimination automatisch erzeugter redundanter Testfälle zu nutzen. Dieser Ansatz ist immer dann geeignet, wenn zahlreiche Testfälle erzeugt werden (etwa zufällig) und alle nicht benötigten aussortiert werden sollen. Durch experimentelle Ergebnisse konnte die Praktikabilität des Ansatzes nachgewiesen werden. Die entwickelten Algorithmen sind robust und schnell. Zukünftige Arbeiten werden die Suche nach Abbruchkriterien sowie den Versuch, direkt weniger überflüssige Testfälle zu erzeugen, umfassen. Dazu gehören gezielte Strategien. So reicht es für eine Methode zur Bearbeitung von AVL-Bäumen etwa, diese mit AVL-Bäumen zu testen, anstatt dazu beliebige Bäume zu nehmen.

Literatur

1. Majchrzak, T.A., Kuchen, H.: Automated test case generation based on coverage analysis. In: TASE '09: Proc. of the 2009 3nd IEEE Int. Symp. on Theoretical Aspects of Software Engineering, IEEE Computer Society (2009) To appear.
2. Mueller, R.A., Lembeck, C., Kuchen, H.: Generating glass-box test cases using a symbolic virtual machine. In: Proc. of the IASTED Int. Conf. on Software Engineering. (2004)
3. Lindholm, T., Yellin, F.: The Java(TM) Virtual Machine Specification (2nd Edition). Prentice Hall PTR (April 1999)
4. o. A.: Standard ECMA-335 - Common Language Infrastructure (CLI). URL: `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

5. Bardin, S., Herrmann, P.: Structural testing of executables. In: ICST '08: Proc. of the 2008 Int. Conf. on Software Testing, Verification, and Validation, IEEE CS (2008) 22–31
6. Lembeck, C., Caballero, R., Mueller, R., Kuchen, H.: Constraint solving for generating glass-box test cases. In: In Proc. of Int. Workshop on Functional and (Constraint) Logic Programming (WFLP). (2004)
7. Fischer, S., Kuchen, H.: Data-flow testing of declarative programs. In: ICFP '08: Proc. of the 13th ACM SIGPLAN int. conf. on Functional programming, ACM (2008) 201–212
8. Pezze, M., Young, M.: Software Testing and Analysis: Process, Principles and Techniques. Wiley (April 2007)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd Ed. The MIT Press (2001)
10. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (August 2006)
11. Zhao, J.: Analyzing control flow in java bytecode. In: Proc. 16th Conf. of Japan Society for Software Science and Technology. (1999) 313–316
12. o. A.: Control Flow Graph Factory 3.3, URL: `http://www.drgarbage.com/control-flow-graph-factory-3-3.html`.
13. Klein, G.: Verified java bytecode verification. it - Information Technology **47**(2) (2005) 107–110
14. Chen, H., Zou, T., Wang, D.: Data-flow based vulnerability analysis and java bytecode. In: ACS'07: Proc. of the 7th Conf. on Applied Computer Science, Stevens Point, Wisconsin, USA (2007) 201–207
15. Doyle, J., Meudec, C.: Ibis: an interactive bytecode inspection system, using symbolic execution and constraint logic programming. In: PPPJ '03: Proc., New York (2003) 55–58
16. Tillmann, N., de Halleux, J.: Pex–white box test generation for .net. In: 2nd Int. Conf. on Tests and Proofs. (April 2008) 134–153
17. de Moura, L., Bjoerner, N. In: Z3: An Efficient SMT Solver. Volume 4963/2008 of Lecture Notes in Computer Science. Springer Berlin (April 2008) 337–340
18. Engel, C., Haehnle, R.: Generating unit tests from formal proofs. In: Proceedings, Testing and Proofs, Springer (2007)
19. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: 11th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer (2005) 365–381
20. Meudec, C.: Atgen: automatic test data generation using constraint logic programming and symbolic execution. Software Testing Verification and Reliability **11**(2) (2001) 81–96
21. Pinte, F., Oster, N., Saglietti, F.: Techniques and tools for the automatic generation of optimal test data at code, model and interface level. In: Companion of the 30th intl. conf. on Software engineering, New York (2008) 927–928
22. Albert, E., Gomez-zamalloa, M., Hubert, L., Puebla, G.: Verification of java bytecode using analysis and transformation of logic programs. In: 9th Intl. Symp. on PADL, number 4354 in LNCS, Springer (2007) 124–139
23. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: ACM SIGPLAN Notices, ACM Press (2000) 268–279
24. Khurshid, S., Pasareanu, C.S.: Generalized symbolic execution for model checking and testing. TACAS 2003, Proc. (2003) 553–568
25. Prasanna, M., Sivanandam, S., Venkatesan, R., Sundarrajan, R.: A survey on automatic test case generation. Academic Open Internet Journal **15** (2005)
26. Edvardsson, J.: A survey on automatic test data generation. In: Proc. of the Second Conf. on Computer Science and Engineering in Linköping, ECSEL (1999) 21–28

Assertion Support for Manipulating Constrained Data-Centric XML

Patrick Michel
Software Technology Group
University of Kaiserslautern

14.04.2009

XML is used for different purposes. We are interested in data-centric applications of XML where it is used to handle structured data in loosely coupled, distributed systems. In many such scenarios, it is important that the XML data complies to structural and integrity constraints, in particular to value-based constraints. The constraints should remain invariant under operations manipulating the data.

In [1], we presented a technique to define and maintain invariants. For a core procedural manipulation language, we showed how to automatically derive the weakest precondition of procedures for the constraints. We presented a path-based formalization of a data-centric XML abstraction and an assertion language that enables weakest precondition generation.

By focusing on paths it is possible to significantly simplify preconditions. This is done by isolating the aspects that are manipulated by the considered procedure from the aspects that remain unchanged. As the overall invariant can be assumed for the input data, only the isolated aspects have to be checked in prestates of procedures.

Building on these theoretical foundations, we are developing higher-level specification languages, which are much more similar to widely used ones, yet can be completely translated into the present formalization. For structural constraints we propose a grammar-based approach in the form of a dialect of Relax NG. For integrity and value-based constraints, we propose a rule-based approach using a specification language resembling XPath. This language has a liberal syntax and is embedded as annotations into the structural grammar.

Although such combinations have been proposed before, we are able to define the semantics of the resulting specification within the path-based formalization and are so able to automatically generate and simplify weakest preconditions for local manipulations. The combination of structural patterns and XPath-like constraints allows to dramatically shorten specifications, providing a very convenient syntax usable by domain experts.

Building on such a specification, the domain expert can write local, atomic manipulation procedures, which represent domain-specific tools necessary to evolve the represented data. Again, we are offering a convenient syntax to define such procedures, which are then translated into the core language presented in [1].

All these techniques are targeted at practical application, ultimately an integration into a general purpose programming language. For this reason traceability of constraints and adequate error messages are necessary to be able to react properly to the failure of a precondition. Every precondition can therefore be traced back to a specific source location, is categorized and has access to concrete values and paths which contribute to it. This information can be used on failure to create a proper response and eventually even proceed by invoking other manipulation procedures.

In this talk I will give a short overview of the general approach and demonstrate our prototype system with practical examples.

References

[1] Patrick Michel and Arnd Poetzsch-Heffter. Assertion support for manipulating constrained data-centric xml. In *International Workshop on Programming Language Techniques for XML (PLAN-X 2009)*, January 2009.

Äquivalenzanalysen - exakt oder nicht - im Vergleich

Dirk Richter

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
06099 Halle/Saale, Germany, richterd@informatik.uni-halle.de

Zusammenfassung Symbolische Modelle definieren Zustandssysteme, welche zur Software-Modell-Prüfung, zum Modell basierten Testen und zur Codegenerierung genutzt werden können. Die Größe und Komplexität dieser Modelle sind dabei entscheidende Einflussfaktoren und sollten möglichst vereinfacht werden. Aus Quellcode gewonnene Modelle in Form von symbolischen Kellersystemen (SPDS) erlauben nicht nur präzisere Ergebnisse, sondern führen auch ohne Modellexplosion bei **exakter** Nachbildung von Rekursion zu weniger Fehlalarmen. Für diese SPDS wurden zwei verschiedene Ansätze verfolgt, die die innere Struktur der Zustände ausnutzen, um den Zustandsraum der Modelle weiter zu verkleinern. Eigene Experimente zeigen, dass diese die Modellprüfung erheblich beschleunigen bzw. die Modellprüfung erst ermöglichen.

Key words: Kellersystem, Remopla, Moped, Software-Modell-Prüfung, Variablenelimination, Äquivalenzanalyse

1 Einleitung

Im Gegensatz zu vergleichbaren Arbeiten bei 'Finite-State' Modellprüfern wie BLAST, SPIN, NuSMV/SMV, F-Soft oder Bogor (Bandera Projekt) untersuche ich unbeschränkte unendliche Modelle in Form von sog. symbolischen Kellsersystemen (SPDS). Viele Modellprüfer beschränken die Rekursionstiefe oder verbieten Methodenaufrufe. Durch diese Unter- bzw. Überapproximation von Methodenaufrufen enstehen Fehlalarme (False Negatives sowie Fehlabstraktionen), die durch korrekte Abbildung von Methodenaufrufen und Rekursion auf SPDS vermieden werden können. Die Beschränkung auf eine konstante maximale Anzahl an Methodenaufrufen ist dann nicht nötig und vermeidet eine exponentielle Modellvergrößerung z.B. durch Inlining. Solche SPDS können mittels JMoped aus Java Bytecode gewonnen und mittels des Modellprüfers Moped überprüft werden und ermöglichen mächtigere interprozedurale und kontextsensitive Modell-Analysen, -Tests und -Prüfungen. Bei Verwendung des Cross-Compilers Grasshopper kann nicht nur Java 1.6 Bytecode verwendet werden, sondern auch Microsoft Intermediate Language. Es ist auch möglich, die Gültigkeit von Java Modeling Language (JML) Annotationen zu überprüfen, wenngleich dies in der Praxis noch unhandlich ist.

Listing 1.1. Java Bispiel zur Berechnung der Fakultät

```
int fac(int n) {
    if (n<=1) return 1;
    return n*fac(n-1);
}
```

Um SPDS Modelle zu optimieren, sind Informationen über das Modellverhalten wichtig, die durch im Programmiersprachenumfeld gängige Programmanalysen gewonnen werden können. Bei einer sog. Äquivalenzanalyse werden z.B. durch Verwendung von Konstanten-Propagation und -Faltung sowie Copy-Propagation Gleichheit und Konstanz von Variablen erkannt. Liang und Harrold zeigen in [1, 2], dass bessere Äquivalenzanalysen zu besserem Slicing sowie besseren Zeiger- und Datenflussanalysen führen. In [3] und [4] wurde gezeigt, dass verschiedene Modellanalysen im Gegensatz zur Anwendung bei herkömmlichen Programmiersprachen **entscheidbar** werden, was prinzipiell die Existenz exakter Modellanalyseverfahren für SPDS nachweist. Dabei heißt eine Modellanalyse **exakt**, wenn das Analyseergebnis weder eine Über- noch eine Unterapproximation darstellt, also präzise das Verhalten des Modells berücksichtigt. Z.B. führt die Verknüpfung von Konstanten-Propagation und -Faltung sowie Copy-Propagation zu einer konservativen Äquivalenzanalyse. Bei einer solchen konservativen Analyse kann es Variablen geben, die zwar gleich oder gar konstant sind, dies aber nicht durch die Analyse entdeckt wird.

Hier soll nun eine modifizierte Version der konservativen Äquivalenzanalyse für SPDS aus [4] kurz erläutert und mit einem neuen **exakten** Verfahren verglichen werden.

2 Symbolische Kellersysteme

$M = (S, \rightarrow, L_A)$ heißt **Kripkestruktur**, falls S und A (nicht notwendigerweise endliche) Mengen sind, $\rightarrow \subseteq S \times S$ und $L_A : S \rightarrow 2^A$. Bei gegebener Kripkestruktur M ist das **Erreichbarkeitsproblem** die Frage, ob es in M einen Pfad von einem Zustand $s \in S$ zu einem anderen Zustand $z \in S$ gibt ($s \xrightarrow{*} z$?). Statt des Erreichbarkeitsproblems können im Rahmen von Modellprüfung auch LTL- oder CTL*-Formeln überprüft werden [5]. Dabei ist eine LTL- oder CTL*-Formel [6] für diese Kripkestruktur oder deren symbolische Beschreibung gegeben. Gefragt wird dann nach der Gültigkeit dieser Formel für einen Anfangszustand bzw. einer Menge von Anfangszuständen dieser Kripkestruktur. Zur Beschreibung von (unendlich) großen Kripkestrukturen können Kellersysteme (Pushdown Systems) verwendet werden. $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ heißt **Kellersystem**, falls P eine Menge von Zuständen, Γ eine endliche Menge (Kelleralphabet) und $\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine Menge von Transitionen sind. Informell ist ein Kellersystem ein Kellerautomat ohne Eingabe. (p, v) heißt **Konfiguration**, falls $p \in P$ und $v \in \Gamma^*$. (p, a) heißt **Kopf** der Konfiguration (p, aw), falls $a \in \Gamma$ und $w \in \Gamma^*$. Auf Konfigu-

Listing 1.2. Generiertes Remopla-Modell (Auszug) zu Listing 1.1

```
1  module int fac(int v0(16)) {
2  int s0(16);  int s1(16);  int s2(16);
3  fac_I:   s0=v0, s1=s0, s2=s1;
4  fac_I1:  s0=1, s1=s0, s2=s1;
5  fac_I2:  if
6           :: s1>s0 -> goto fac_I7, s0=s2;
7           :: else -> s0=s2;
8           fi;
9  fac_I5:  s0=1, s1=s0, s2=s1;
10 fac_I6:  return s0;
11 fac_I7:  s0=v0, s1=s0, s2=s1;
12 fac_I8:  s0=v0, s1=s0, s2=s1;
13 fac_I9:  s0=1, s1=s0, s2=s1;
14 fac_I10: if
15          :: s1>=s0 -> s0=s1-s0, s1=s2;
16          :: else -> s0=(65536-(s0-s1)%65536)%65536, s1=s2;
17          fi;
18 fac_I11: s0=fac_I(s0);
19 fac_I14: s0=(s1*s0)%65536, s1=s2;
20 fac_I15: return s0; }
```

rationen wird die Transitionsrelation $\hookrightarrow$ erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$. Mit $\xrightarrow{*}$ wird die reflexive und transitive Hülle von $\rightarrow$ bezeichnet. Bei einem **Symbolischen Kellersystem** (SPDS) werden die Transitionen nur indirekt (symbolisch) mittels Relationen beschrieben, was die Angabe des vollständigen Kellersystems vereinfacht [7].

Solche SPDS können mit Hilfe der Modellsprache Remopla [8] modelliert werden. Remopla ist zwar syntaktisch ähnlich zu Promela (Eingabesprache für den SPIN Modellprüfer [9]), unterstützt aber keine parallelen Prozesse, sondern synchron parallele Konfigurationenübergänge und explizite Rekursion. In Listing 1.1 ist ein Java-Beispiel zu Berechnung der Fakultätsfunktion abgebildet. Dieses kann mittels des Werkzeuges JMoped automatisch in das Remopla-Modell aus Listing 1.2 überführt werden. Dabei wurden 16 Bit (angehängt an eine Variablendeklaration) zur Modellierung von Integern verwendet, um das Verhalten des Java Programms nachzubilden. Wie dabei nicht zu sehen ist, können in Remopla neben lokalen Variablen loc_q und Parametern $pars_q \subseteq loc_q$ eines Moduls q (auch Prozedur genannt) auch globale Variablen *globs* sowie Arrays oder Verbunde (Struct) deklariert werden. Die lokalen Variablen und Methodenparameter werden in SPDS als Bitvektoren über dem Kelleralphabet zusammen mit der Aufrufhierarchie im Keller repräsentiert. Globale Variablen (in Java Klassenvariablen), die Halde (Heap) sowie Ausnahmen (Exceptions) werden mit Hilfe der Zustände eines Kellersystems beschrieben. Für weitere Details zur Konstruktion von Remopla-Modellen für C und Java sei auf [10–12, 3] verwiesen.

So konstruierte Modelle in Form eines Remopla-Modells können dann durch mein Werkzeug **HalSPSI** verbessert und anschließend durch den Modellprüfer Moped [10] geprüft oder anders weitergenutzt werden. Ziel der im Folgenden erklärten Äquivalenzanalysen ist es, das dazu nötige Kelleralphabet und die benötigten Kellerzustände bereits symbolisch a priori so stark wie möglich zu verringern. Desto genauer hierzu das Analyseergebnis ist, desto bessere Transformationen können später erfolgen. Exakte Äquivalenzanalysen können daher zu einer angestrebten optimalen Transformation führen.

3 Variablenelimination mittels Äquivalenzanalysen

Mittels so genannter Äquivalenzanalysen können Transformationen eingesetzt werden, welche den Zustands- und Konfigurationenraum verkleinern, indem gewisse Variablen als „überflüssig“ erkannt werden. Eine solche Transformation ist die Variablenelimination, welche Äquivalenzanalysen nutzt und hier kurz als Motivation erläutert wird.

Sei dazu angenommen, dass $Vars_q$ die Menge der an der SPDS Marke q verfügbaren Variablen (globale und lokale sowie Parameter-Variablen) ist. Ein Kopf (p, a) bzw. dessen Variablenbelegung $f^q : Vars_q \rightarrow \mathbb{N}$ an der Marke q heißt **realisierbar**, falls ein Pfad[1] aus einer gegebenen Anfangskonfiguration s zu einer Konfiguration (p, aw) existiert (in Zeichen $s \xrightarrow{*} (p, aw)$). Zwei Variablen x und y eines SPDS vor einer Marke q heißen **äquivalent** (in Zeichen $x \equiv_q y$), wenn für alle realisierbaren Variablenbelegungen f^q gilt: $f^q(x) = f^q(y)$. Analog heißt eine Variable x konstant ($x \equiv_q c$), falls $f^q(x) = c$ mit $c \in \mathbb{N}$. D.h. für alle k, welche $s \xrightarrow{*} k$ mit $q = Label(k)$ erfüllen, gilt: $f^q(x) = f^q(y)$ bzw. $f^q(x) = c$. Äquivalente Variablen und Konstanten vor jeder Marke werden zu **Äquivalenzklassen** zusammengefasst. Diese ergeben für die Marke q die **Äquivalenzinformation** $E_q = \{\{x_{11}, x_{12}, ...x_{1n_1}\}, \{x_{21}, x_{22}, ...x_{2n_2}\}, ...\{x_{m1}, x_{m2}, ...x_{mn_m}\}\}$ mit $\forall i, j, l : x_{li} \equiv_q x_{lj}$. In Remopla-Modellen wird dafür abkürzend geschrieben $x_{11}{=}x_{12}...{=}x_{1n_1}$ ... $x_{m1}{=}x_{m2}...{=}x_{mn_m}$.

Wird z.B. in Listing 1.2 zum Moduleintritt von fac keine Äquivalenz angenommen, gilt nach den parallen Zuweisungen in Zeile 3 $v0 \equiv_{fac\_I1} s0$ und nach den Zuweisungen aus Zeile 4 $s0 \equiv_{fac\_I2} 1$ sowie $s1 \equiv_{fac\_I2} v0$. Mit Hilfe dieser Äquivalenzinformationen kann z.B. der boolsche Ausdruck $s1 > s0$ in Zeile 6 durch den äquivalenten Ausdruck $v0 > 1$ ersetzt werden und erlaubt auf diese Weise eine mögliche Einsparung der beiden lokalen Variablen $s0$ und $s1$, was den Zustands- und Konfigurationenraum verkleinert.

In [4] wurde gezeigt, dass das Finden einer Ersetzung von Variablen durch äquivalente Variablen/Konstanten mit minimalem SPDS-Konfigurationenraum NP-schwer ist. In der Praxis ist dies allerdings dank effizienter ILP-Solver (Integer Linear Program) unkritisch. Daher wurde in den Experimenten das Verfahren aus [4] um eine optimale Repräsentantenwahl (Auswahl von Variablendeklarationen) erweitert, welche durch Konstruktion eines ILP ein Überdeckungsproblem

[1] nicht notwendigerweise endlich

löst. Dabei muss jede Variablenverwendung durch eine Variablendeklaration einer äquivalenten Variable oder Konstante „überdeckt“ werden. Dadurch ändert sich das durch das SPDS beschriebene Kellersystem nicht. Auch Aussagen von LTL/CTL*-Formeln bleiben unverändert, solange lediglich lesende Verwendungen von Variablen durch andere ersetzt werden. Wird allerdings eine Variable überflüssig und somit in einer späteren Phase aus dem Modell entfernt, so geht dies i.A. nur, sofern die gegebene LTL/CTL*-Formel nicht über diese 'wegoptimierte' Variable spricht. Falls doch, so sind in jeder symbolischen Konfiguration des SPDS ggf. indirekte Abhängigkeiten einzuführen, um die durch die LTL/CTL*-Formel erzeugten indirekten Verwendungen (über Prädikate) ebenfalls abzudecken und damit die an der LTL/CTL*-Formel beteiligten Variablen in die Lösung der Repräsentantenwahl zu zwingen[2].

4 Konservative Äquivalenzanalyse

Für gängige Hochsprachen ist es **unentscheidbar** (Reduktion Postsches Korrespondenzproblem) fest zu stellen, ob zwei Variablen x und y an einem Programmpunkt q äquivalent sind [13]. Wie in [4] gezeigt, ist dieses Problem für SPDS **entscheidbar**, wenngleich zu aufwänding um damit Modellprüfungen zu beschleunigen. Daher wurde zunächst eine konservative Äquivalenzanalyse untersucht. Eine Äquivalenzinformation $E_q = \{\{x_{11}, x_{12}, ...x_{1n_1}\}, \{x_{21}, x_{22}, ...x_{2n_2}\}, ...\{x_{m1}, x_{m2}, ...x_{mn_m}\}\}$ an einer Marke q heißt **konservativ**, falls $\forall i, j, l : x_{li} \equiv_q x_{lj}$. Anders als bei exakten Äquivalenzinformationen kann es hier Variablenäquivalenzen $a \equiv_q b$ geben, so dass $\nexists g \in E_q : a, b \in g$.

Wird zu der in [4] beschriebenen Konstanten-Propagation und -Faltung noch eine „Copy-Propagation-Analysis“ verwendet, so können damit „Copy-Chains“[3] verfolgt werden. Ein Ziel ist das Aufbrechen dieser „Copy-Chains“, damit letztendlich weniger Variablen im Modell benötigt werden und sich der Zustands- und Konfigurationenraum des Modells verringert. Konkret werden bei der implementierten interprozeduralen Äquivalenzanalyse Äquivalenzen an Konfigurationen für Variablen und zu Konstanten auswertbaren Ausdrücken verfolgt. Z.B. wird nicht durch jede Äquivalenzanalyse die Variablenäquivalenz $x \equiv_q y$ in Zeile 9 von Listing 1.3 erkannt. Es wird in einem Lauf des Modells aus Listing 1.3 eine Anweisungen aus den Zeilen 5 bis 7 zufällig ausgewählt, da alle If-Bedingungen erfüllt sind. Auch meine konservative Äquivalenzanalyse erkennt die frühzeitige Terminierung aller Modellpfade in Zeile 7 durch einen arithmetischen Überlauf nicht und kann damit die Äquivalenz $x \equiv_q y$ in Zeile 9 nicht erkennen.

5 Exakte Äquivalenzanalyse

In [7] wird beschrieben, wie zu einem gegebenen SPDS P symbolisch ein endlicher Automat $Post^*(P)$ konstruiert werden kann, welcher die Menge aller er-

[2] gleichgültig, ob optimale Repräsentantenwahl oder nicht

[3] Copy-Chains sind Zuweisungen mit Wertweiterreichung, wie sie typischerweise bei einem simulierten Operadenstack bei Push- und Pop-Operationen auftreten.

Listing 1.3. Variableninäquivalenz von x und y an q trotz partieller Symmetrie

```
1     int x(1);
2     int y(1);
3     x=1, y=0;
4     if
5         :: true -> x=y, y=x;     # gleichbedeutend zu x=0, y=1;
6         :: true -> skip;
7         :: true -> y=x+1;        # Pfad terminiert wegen Überlauf
8     fi;
9     # {x=y} <= unerkennbare Äquivalenz für konservative Analyse
10 q: goto q;
```

reichbaren Konfigurationen der gegebenen Initialkonfigurationen charakterisiert. Mittels $Post^*(P)$ kann aus den Initialkonfigurationen zu jeder SPDS Marke q eine charakteristische Funktion f^q für die exakten Variablenbelegungen für lokale und globale Variablen bestimmt werden mittels Beschränkung der Transitionen aus $Post^*(P)$ auf symbolische Köpfe. Es wird nun gezeigt, wie aus dieser charakteristischen Funktion f^q die exakten Äquivalenzinformationen effizient bestimmt werden und dabei sog. Symmetrien ausgenutzt werden können. Nach [14] heißt dazu eine Funktion $f : \{0,1\}^n \to \{0,1\}$ partiell **symmetisch in den Variablen** x_k **und** x_j, falls für jede Belegung der x_i gilt: $f(x_1, ...x_{k-1}, x_k, x_{k+1}, ...x_{j-1}, x_j, x_{j+1}, ...x_n) = f(x_1, ...x_{k-1}, x_j, x_{k+1}, ...x_{j-1}, x_k, x_{j+1}, ...x_n)$. Die Funktion f heißt **symmetrisch in einer Menge** von disjunkten Variablenmengen[4] R, wenn f partiell symmetrisch in je zwei Variablen $a, b \in m$ jeder Menge $m \in R$ ist. Die Funktion $f_{x_i} := f(x_1, ...x_{i-1}, 1, x_{i+1}, ...x_n)$ heißt **positiver Kofaktor** und $f_{x_i'} := f(x_1, ...x_{i-1}, 0, x_{i+1}, ...x_n)$ **negativer Kofaktor** von f. Man bezeichnet $f_{ab} := (f_a)_b$ als iterierten Kofaktor. Weitere Details finden sich in [14].

Symmetrieen und die zuvor beschriebenen Äquivalenzklassen weisen einen interessanten Zusammenhang auf: jede Äquivalenz zweier Variablen in einem SPDS führt zu partiellen Symmetrieen. Wie folgendes Theorem zeigt, gilt dies allerdings nicht für die Umkehrung, weswegen Symmetrieberechnungen nicht ausschließlich zur Bestimmung der Äquivalenzklassen genutzt werden können.

Theorem 1. ***Partielle Symmetrie der Variablenäquivalenz***
Sind zwei Variablen $x = (x_1, x_2, ..., x_n)$ und $y = (y_1, y_2, ..., y_n)$ eines SPDS an einer Marke q äquivalent (also $x \equiv_q y$), so ist die charakteristische Funktion $f^q : \{0,1\}^{2n} \to \{0,1\}$ der Variablenbelegungen[5] für q partiell symmetrisch in der Menge von Variablenmengen $\{\{x_1, y_1\}, \{x_2, y_2\}, ...\{x_n, y_n\}\}$. D.h. für jedes i ist f^q in den Variablen x_i und y_i partiell symmetrisch. Die Umkehrung des Theorems gilt nicht.

[4] $R \subset 2^{\{x_1, x_2, ...x_n\}}$ bzw. $\cup R \subseteq \{x_1, x_2, ...x_n\}$ wobei $\forall a, b \in R : a \cap b = \emptyset$
[5] z.B. als BDD repräsentiert

Proof. „⇒“ Sei $x \equiv_q y$, so gilt für jede Belegung der Variablen x und y: $x_1 = y_1, x_2 = y_2, \ldots x_n = y_n$ und damit $\forall i : f^q$ ist partiell symmetrisch in den Variablen x_i und $y_i \Rightarrow f^q$ ist partiell symmetrisch in der Partition $\{\{x_1, y_1\}\}, \{x_2, y_2\}, \ldots, \{x_n, y_n\}\}\}$.

„⇐“ Gegeben sei das SPDS aus Listing 1.3. Darin sind alle Variablenbelegungen an der Marke q als $\{\{x = 0, y = 1\}, \{x = 1, y = 0\}\}$ erkennbar. x und y sind hiernach nicht äquivalent. Wegen $f^q(x, y) = f(0, 1) = f(1, 0) = 1$ ist f^q aber trotzdem partiell symmetrisch in den Variablen x und y. ■

Für einen Äquivalenztest zweier Variablen genügt es, Kofaktoren zu bilden:

Theorem 2. ***Äquivalenzbestimmung zweier SPDS-Variablen*** x ***und*** y
Sei f^q die charakteristische Funktion aller Köpfe von Konfigurationen an der Marke q. Dann gilt für $x, y \in Vars_q$: $x \equiv_q y \Leftrightarrow \forall i : f^q_{x_i y'_i} = \emptyset \wedge f^q_{x'_i y_i} = \emptyset$

Proof. O.B.d.A. sei $f := f^q$ mit $f \neq \emptyset$.

„⇒“ Sei $x \equiv_q y$. Dann gilt für jede mögliche Belegung der Variablen x bzw. y durch $a = (a_1 a_2 \ldots a_n) \in \{0,1\}^n$ bzw. $b = (b_1 b_2 \ldots b_n) \in \{0,1\}^n$: $(f[x := a, y := b] \neq \emptyset) \Rightarrow a = b$. Gäbe es nun ein i, so dass $f_{x_i y'_i} \neq \emptyset$, so gäbe es Belegungen α, β mit $\alpha_i = 1$ und $\beta_i = 0$, so dass $f[x := \alpha, y := \beta] \neq \emptyset$. Die Voraussetzung liefert dann aber wegen $\alpha = \beta$ einen Widerspruch zu $\alpha_i = 1 \neq 0 = \beta_i$. D.h. es kann kein solches i geben, also ist $\forall i : f_{x_i y'_i} = \emptyset$. Analoges gilt für die Annahme $f_{x'_i y_i} \neq \emptyset$.

„⇐“ Sei $\forall i : f_{x_i y'_i} = \emptyset \wedge f_{x'_i y_i} = \emptyset$ und angenommen $x \not\equiv_q y$. Nach Annahme $\exists j, \exists a, b \in \{0,1\}^n : a_j \neq b_j$ und $f[x := a, y := b] \neq \emptyset$.
1. Fall: $a_j = 1$. Wegen $a_j \neq b_j$ ist dann $b_j = 0$ und damit $\emptyset \neq f[x := a, y := b] = f[x := a, y := b]_{x_j} = f[x := a, y := b]_{x_j y'_j}$, und damit auch $f_{x_j y'_j} \neq \emptyset$. Ein Widerspruch zur Voraussetzung. Damit gibt es kein solches j und es ist $x \equiv_q y$.
2. Fall: $a_j = 0$. Analog. ■

Der Aufwand zur Bestimmung aller Äquivalenzklassen an einer Marke q beträgt damit $O(m^2 \cdot n \cdot |G_f|)$, wobei m die Anzahl der SPDS-Variablen[6] ist und $|G_f|$ die Größe eines BDDs für f. Die Symmetrieeigenschaft bzw. vielmehr die Asymmetrieeigenschaft in Theorem 1 kann als hinreichende Bedingung zur Nicht-Äquivalenz von SPDS-Variablen genutzt werden. So kann die Berechnung fast aller Kofaktoren aus Theorem 2 entfallen, denn für BDDs gibt es effiziente Asymmetrietests in Form von Signaturen [14, 15], welche zu zwei gegebenen Variablen x_i und x_j einer Funktion f die Asymmetrie in diesen beiden Variablen entdecken. Formal: $\tau(x_i, y_i) \Rightarrow f$ nicht partiell symmetrisch in x_i und y_i.

Lemma 1. ***Einsparung von Kofaktorbildungen aus Theorem 2***
Sei τ ein Asymmetrietest. Dann gilt: $(\exists i : \tau(x_i, y_i)) \Rightarrow x \not\equiv_q y$.

Proof. Ang. $\exists i : \tau(x_i, y_i)$. Dann ist per Definition f in den Variablen x_i und y_i nicht partiell symmetrisch. Wegen Theorem 1 kann somit nicht $x \equiv_q y$ gelten. Es brauchen somit keine Kofaktoren aus Theorem 2 berechnet werden. ■

[6] Summe aus lokal, global und Parameter-Variablen mit Vielfachheit bei Arrays

Wie Experimente zeigen, sind die Äquivalenzklassen sehr klein[7] mit maximaler Größe $k << m$. Werden nur Signatur basierte Asymmetrietests (alle aus [14, 15]) mit Berechnungszeit $O(|G_f|)$ verwendet, so reduziert sich der Aufwand zur Bestimmung aller Äquivalenzklassen auf $O(k^2 \cdot c^2 \cdot n \cdot |G_f| + m \cdot n \cdot |G_f| + m^2 \cdot n)$ mit sehr kleinem c, welches die Anzahl der binären Variableninäquivalenzen misst, die durch keinen Asymmetrietest entdeckt wurden[8].

Theorem 3. ***Identifkation von Konstanten***
Eine Variable $x = (x_1 x_2 \dots x_n)$ an einer Marke q ist konstant gdw.

$$\forall i : f_{x_i} = \emptyset \vee f_{x'_i} = \emptyset.$$

Ihr Wert entspricht dabei derjenigen Belegung $\overline{x_i}$ der x_i, so dass $f_{\overline{x_1 x_2} \dots \overline{x_n}} \neq \emptyset$.

Die Funktion *Cudd_FindEssential* des BDD-Pakets Cudd findet genau derartige Variablenbelegungen. Für konstante Variablen müssen ebenso keine Kofaktoren aus Theorem 2 bestimmt werden.

6 Experimente und Vergleich der Äquivalenzanalysen

Die konservative Äquivalenzanalyse liefert eine Teilmenge der exakten Äquivalenzinformationen. Die Überapproximation meiner Implementation entsteht zum Einen durch Abstraktion von Arrays und Verbunden – eine Erweiterung ist möglich, wenngleich nicht nötig, wie die Tabellen 1 und 2 zeigen, da so bereits die meisten Äquivalenzen entdeckt werden und sich die Kellersystemgröße nicht mehr wesentlich verringert. Zum Anderen entsteht Ungenauigkeit durch die Abstraktion nicht realisierbarer Pfade im Modell, da alle potentiell möglichen interprozeduralen Kontrollflüsse anhand des Kontrollflussgraphen betrachtet werden (eine Überapproximation der Pfade, siehe Listing 1.3).

Als Grundlage für die Experimente dienten 191 Remopla-Modelle, zu denen exakte Äquivalenzinformationen berechnet werden konnten. Die Modelle wurden mittels JMoped aus Java Beispielen gewonnen, wobei zur Modellgenerierung jeweils unterschiedliche Bitbreiten (bis zu 8 Bit) zur Modellierung von Integern verwendet wurden. Aufgabe für den Modellprüfer Moped war es, zu diesen Modellen jeweils vor und nach der Variablenelimination das Fehlschlagen von Java-Zusicherungen (Assertions) zu prüfen. In gleicher Weise können auch nicht (korrekt) behandelte Ausnahmen, Speicherüberläufe oder andere Fehler automatisch geprüft werden. Durchgeführt wurden die Experimente mit einem AMD 64 X2 4200+ mit 2 GB Hauptspeicher unter Linux (Kernel 2.6.27).

In den Experimenten (Tabelle 1) werden durch die konservative Äquivalenzanalyse im Durchschnitt 76% und in Extremfällen über 96% der auftretenden Äquivalenzen unter den Variablen erkannt. In Tabelle 2 ist die Modellgröße (gemessen in der Anzahl der Köpfe) miteinander verglichen. Wobei $^-$ für die unoptimierte Variante ohne **HalSPSI** , $^+$ für die konservative und * für die exakte

[7] Durchschnitt k=2,45

[8] erster Summand ergibt sich aus den äquivalenten sowie den nicht durch Asymmetrietests erkennbaren Inäquivalenzen (k und c sehr klein); der zweite aus Berechnung der Signaturen (jedes Bit jeder Variablen); der dritte aus Vergleich der Signaturen

Code-Beispiel	
ParameterRestrictions.java (7Bits)	96%
ConcreteFieldClass.java (8Bit)	92%
While.java(8Bit)	85%
Durchschnitt (alle 191 Instanzen)	76%

Tabelle 1. Gleichheit der approximativen und exakten Äquivalenzklassen

Code-Beispiel	hd^-	hd^+	hd^*
ParameterRestrictions(7Bits)	$3,0 \cdot 10^{51}$	$2,3 \cdot 10^{49}$	$2,3 \cdot 10^{49}$
ConcreteFieldClass(8Bit)	$3,7 \cdot 10^{48}$	$1,4 \cdot 10^{40}$	$1,3 \cdot 10^{40}$
While(8Bit)	$4,3 \cdot 10^{45}$	$1,0 \cdot 10^{42}$	$1,0 \cdot 10^{42}$
Durchschnitt (alle 191)	$8,6 \cdot 10^{89}$	$5,2 \cdot 10^{52}$	$2,0 \cdot 10^{52}$

Tabelle 2. Vergleich der Modellgröße (Anzahl Köpfe) vor der Reduktion (hd^-) und nach der Reduktion für approximative (hd^+) und exakte Variante (hd^*)

Modellanalyse steht. Es zeigt sich, dass die Transformationen die Modellgröße signifikant um viele Größenordnungen verringern, was sich auf eine wesentlich geringere Modellprüfzeit auswirkt. Die konservative Analyse verbessert das Modell bereits erheblich, welches durch ein exaktes Verfahren nur noch relativ wenig optimiert werden kann. Die Laufzeit für ein exaktes Verfahren ist jedoch größer als eine Modellprüfung selbst, wo hingegen die Laufzeit für das konservative Verfahren vernachlässigbar klein gegenüber der Modellprüfzeit ist. In 5% der Modellprüfungen konnten dank vorheriger Äquivalenzanalyse und Transformation Speicherüberläufe verhindert und das Modell nun erfolgreich geprüft werden.

7 Zusammenfassung und Ausblick

Es wurden zwei Äquivalenzanalysen vorgestellt, mit denen die innere Zustandsstruktur komprimiert und Modelle bereits symbolisch vereinfacht werden können. Die in den Experimenten verwendete Variablenelimination ermöglicht damit u.a. effizienteres Testen, effizientere Testdatengenerierung mit kompakteren Testdaten und einfachere Simulationen. Exakte Äquivalenzinformationen führen dabei aber auch zu präziseren Analysen (z.B. Points-To-Analysen) für die Ausgangssprache wie C oder Java. Die Experimente zeigen, dass durch **HalSPSI** die Modellprüfung erheblich beschleunigt bzw. die Modellprüfung erst ermöglicht wird. Im Zusammenspiel mit anderen Transformationen wie z.B. Slicing, Wertebereichsreduktion oder Stotterreduktion kann die Modellprüfung in Einzelfällen sogar ganz entfallen. Es ergeben sich dann nochmals beträchtliche Verbesserungen durch Synergieeffekte. Z.B. wird bereits die Kombination der konservativen und der exakten Äquivalenzanalyse[9] die Insgesamtlaufzeit für die exakte Äquivalenzanalyse reduzieren und immer noch zu exakten Ergebnissen führen. Unabhängig von der Reduktion durch Variablenelimination aus Abschnitt 3 kann in **HalSPSI** der SMT-Solver Yices aktiviert werden, um boolsche (Teil-)Ausdrücke

[9] erst die schnelle konservative, dann die exakte durchführen

zu vereinfachen. Analog zum Extrahieren der exakten Äquivalenzinformationen können auf ähnliche Weise sämtliche boolsche Ausdrücke exakt auf Konstanz geprüft und vereinfacht werden, um somit Kontrollflüsse genauer vorher zu sagen.

Literatur

1. D. Liang and M. J. Harrold. *Equivalence analysis: a general technique to improve the efficiency of data-flow analyses in the presence of pointers.* ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 1999.
2. D. Liang and M. J. Harrold. *Equivalence analysis and its application in improving the efficiency of program slicing.* Transactions on Software Engineering and Methodology (TOSEM), Volume 11 Issue 3, 2003.
3. D. Richter and W. Zimmermann. *Slicing zur Modellreduktion von symbolischen Kellersystemen.* Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2007.
4. D. Richter. *Modellreduktionstechniken für symbolische Kellersysteme.* Proc. of the 25. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2008.
5. R. Mayr. *Process Rewrite Systems.* In Electronic Notes in Theoretical Computer Science, Volume 7, pp. 185-205, 1997.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Massachusetts Institute of Technology, Cambridge, 2000.
7. S. Schwoon. *Model-Checking Pushdown Systems.* Technische Universität München, 2002.
8. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. *Introduction to Remopla.* Institute of Formal Methods in Computer Science, University of Stuttgart, 2006.
9. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.
10. J. Esparza and S. Schwoon. *A BDD-based model checker for recursive programs.* LNCS Volume 2102, pp. 324-336, Springer, 2001.
11. J. Obdrzalek. *Formal verification of sequential systems with infinitely many states.* Master's Thesis, FI MU Brno, Masaryk University, 2001.
12. J. Obdrzalek. *Model Checking Java Using Pushdown Systems.* LFCS, University of Edinburgh, 2002.
13. O. Rüthing, J. Knoop, and B. Steffen. *Detecting Equalities of Variables: Combining Efficiency with Precision.* Proc. of the 6th International Symposium on Static Analysis, LNCS 1694, 1999.
14. P. Molitor and C. Scholl. *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen.* Teubner Verlag, 1999.
15. P. Molitor and J. Mohnke. *Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods.* Springer Netherlands, 2004.

Sichere Produktlinien: Herausforderungen für Syntax- und Typ-Prüfungen

Christian Kästner[1], Sven Apel[2], and Gunter Saake[1]

[1] School of Computer Science, University of Magdeburg, ckaestne/saake@ovgu.de
[2] Dept. of Informatics and Math., University of Passau, apel@uni-passau.de

Softwareproduktlinien stellen neue Herausforderungen an Entwicklungswerkzeuge und Programmiersprachen. Während in der klassischen Anwendungsentwicklung genau ein Programm entwickelt wird, entwickelt man in einer Softwareproduktlinie gleich eine Vielzahl verwandter Programme in einer Domäne aus gemeinsamen, wiederverwendbaren Artefakten [15, 3, 5]. Die Gemeinsamkeiten und Unterschiede der Programme einer Software-Produktlinie werden durch Features modelliert [10, 5]. Durch Kombination von Features können (potentiell tausende) maßgeschneiderte Programme spezifiziert und entsprechend generiert werden. Typische Implementierungsansätze die eine automatisierte Generierung von Programmen aus einer Produktlinie erlauben sind Frameworks [9], Präprozessoren (#ifdef) [15, 7], Generatoren [8] oder spezielle Sprachen der Aspekt- oder Feature-Orientierten Programmierung [14, 16, 4, 2].

Software-Produktlinien erlauben die Maßschneiderung von Programmen auf bestimmte Anwendungsszenarien bei hoher Wiederverwendung, und versprechen somit schnellere Entwicklung von maßgeschneiderten Lösungen bei geringeren Kosten [15, 3, 5]. Diesen Vorteilen stehen aber auch neue Herausforderungen gegenüber: eines der wichtigsten Probleme bei der Implementierung von Softwareproduktlinien ist, dass Fehler sehr schwierig zu finden sind wenn sie nur in einzelnen generierten Programmen mit bestimmten Features oder Feature-Kombinationen auftauchen [6, 17, 11, 13]. So schleichen sich Syntax-Fehler (z. B. vergessene schließende Klammer), Typ-Fehler (z. B. ins leere zeigende Methodenaufrufe oder fehlende Klassen), und auch Semantik-Fehler (z. B. fehlerhaftes Verhalten bei bestimmten Featurekombinationen) bei vielen Implementierungsansätzen leicht in einem generierten Programm ein. Oft werden diese Fehler aber erst bemerkt wenn ein Kunde tatsächlich – ggf. Monate nach der ursprünglichen Entwicklung – eine problematische Feature-Kombination anfragt. Das Generieren und isolierte Prüfen aller potentiellen Programme skaliert für Produktlinien nicht.

In dem Vortrag geben wir einen Überblick zu aktuellen Forschungen zum Erkennen von Syntax- und Typ-Fehlern in Produktlinien, und zwar für die gesamte Produktlinie anstatt nur für einzelne Programme:

- Syntax-Fehler können etwa beim Präprozessoreinsatz vermieden werden, wenn man gewisse, leicht überprüfbare Regeln einhält (und dass sogar sprachübergreifend) [12, 13].
- Typ-Fehler können mit einem speziellen Typsystem für Softwareproduktlinien die gesamte Produktlinie statt einzelner Programme prüfen. Dazu haben wir mit FFJ und CFJ zwei auf Featherweight Java basierende Calculi entwickelt (und entsprechende Tools für Java implementiert), die zum Quelltext auch Informationen zu Features der Produktlinie berücksichtigen und somit garantieren können dass aus einer wohl-

getypten Produktlinie nur wohlgetypte Programme generiert werden können [11, 1].

Zu diesen Konzepten stellen wir konkrete Tools und Sprachen zur Produktlinienentwicklung vor, welche diese Prüfungen unterstützen, sowie Erfahrungen aus verschiedenen Fallstudien.

Literatur

1. S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 101–112. 2008.
2. S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
4. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.
5. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.
6. K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. 2006.
7. M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, 2002.
8. S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 810–811. 2003.
9. R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.
10. K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
11. C. Kästner and S. Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 258–267. 2008.
12. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. 2008.
13. C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*. 2009.
14. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242. 1997.
15. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.
16. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443. 1997.
17. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. 2007.

Informationsflussanalyse für Java (Erweitertes Abstract)

Christian Hammer

Universität Karlsruhe
hammer@ipd.info.uni-karlsruhe.de

1 Problemstellung

Eine Reihe von Mechanismen, die Software-bedingten Sicherheitsproblemen vorbeugen oder wenigstens seine negativen Auswirkungen verhindern sollen, sind heute fester Bestandteil des täglichen Lebens: Zugangskontrolle, Codesignierung, Kryptographie, Virenscanner, Firewalls, automatische Updates, etc.

All diesen Ansätzen ist jedoch gemein, dass sie nur punktuell auf Probleme reagieren, aber nie Sicherheit auf dem ganzen Lebensweg von Daten (end-to-end security) garantieren können. Mit klassischer Sicherheitstechnik lassen sich lediglich Identität und Herkunft einer Software überprüfen, semantische Eigenschaften werden nicht berücksichtigt. So kann beispielsweise die Zugangskontrolle des Betriebssystems nur festlegen, ob bestimmte Personen Zugang zu vertraulichen Daten bekommen dürfen. Nach Freigabe dieser Daten entzieht sich ihre weitere Verwendung (d.h. die eigentliche Programmsemantik) weitgehend der Kontrolle des Benutzers. Dieser kann i.A. nur anhand der Identität des Programmherstellers auf dessen Kompetenz und Gutartigkeit hoffen. Allerdings wurden in der Vergangenheit einige Beispiele bekannt, bei denen selbst namhafte Hersteller Daten über die Benutzer gesammelt und an sich weitergeleitet haben. Meist hat der Benutzer auf diese Aktivitäten keinen Einfluss.

2 Zielsetzung

Um end-to-end Security zu erreichen, wurde die Informationsflusskontrolle (information flow control, IFC) erfunden. Diese überprüft ein gegebenes Programm auf die Einhaltung einer bestimmten Sicherheitspolitik, normalerweise eine Variante von Nichtinterferenz. Dazu wird jedem Informationskanal ein gewisses Sicherheitslevel wie z.B. “geheim” oder “öffentlich” zugewiesen. Nichtinterferenz ist dann gegeben, wenn geheime Eingaben die öffentlichen Ausgaben nicht beeinflussen können. Eine Kombination von Nichtinterferenz und Zugangskontrolle garantiert die Sicherheit der Daten auf dem ganzen Verarbeitungspfad. Die klassische Nichtinterferenz ist jedoch für die Praxis zu restriktiv, weil geheime Daten in nichttrivialen Programmen immer auch die öffentlichen Ausgaben in gewissem Maß beeinflussen. Man denke nur an geheime Passwörter zum Login: Das Ergebnis der Überprüfung des eingegebenen Passworts muss veröffentlicht werden. Dazu wurde die sog. Deklassifikation eingeführt. Diese erlaubt unter geeigneten Umständen die Veröffentlichung von Daten, die mit Hilfe von geheimen

Daten berechnet wurden. Deklassifikationen und Annotationen bilden die sog. Sicherheitspolitik.

Bisherige Ansätze zur Informationsflusskontrolle verwenden nur ein sehr eingeschränktes Repertoire der Programmanalyse: In der Literatur fanden sich bis vor Kurzem fast ausschließlich spezielle Typsysteme, die die klassischen Variablentypen um Sicherheitsannotationen wie "geheim", "öffentlich" usw. erweitern. Typsysteme sind zwar elegant und effizient zu implementieren, allerdings benötigen sie dazu eine große Zahl von Benutzerannotationen. Dadurch ist der Aufwand, eine Sicherheitspolitik für ein vorhandenes Programm zu definieren, sehr hoch.

3 Lösungsansatz

Die hier vorgestellte Arbeit beschreitet einen neuen Ansatz zur effektiven Informationsflusskontrolle, welcher auf Programmabhängigkeitsgraphen und sog. Program Slicing basiert. Program Slicing ist in der Literatur schon seit längerem als ein geeignetes Mittel zur Informationsflusskontrolle bekannt. Allerdings wurde dieses Ergebnis bisher nur für eine Variante des Lamda-Kalküls gezeigt und noch nie implementiert. Ein vor kurzem präsentiertes Theorem über den Zusammenhang von IFC und Program Slicing mit Abhängigkeitsgraphen lässt erstmals auch präzise Analysen von echten Sprachen zu.

Abhängigkeitsgraphen und Program Slicing sind grundsätzlich präziser als Typsysteme, da sie den Kontrollfluss durch das Programm berücksichtigen (Fluss-Sensitivität), verschiedene Aufrufkontexte einer Prozedur unterscheiden (Kontext-Sensitivität) und bei objektorientierten Programmen sogar zwischen den Feldern unterschiedlicher Objekte differenzieren (Objekt-Sensitivität). In dieser Arbeit untersuchen wir realistische Programme bzw. deren Sicherheitskerne, die in Java Bytecode vorliegen. Anhand der Bytecode-Semantik wird ein präziser Programmabhängigkeitsgraph erzeugt, der alle zur Laufzeit möglichen Informationsflüsse abbildet. Ausgehend von dem generierten Graphen wird eine auf Program Slicing basierende Analyse beschrieben, die klassische Nichtinterferenz garantiert. Außerdem wurde eine Möglichkeit zur Deklassifikation geschaffen, um an bestimmten Stellen im Programm die Sicherheitsstufe von Informationen herabzustufen. Eine weitere Analyse erlaubt kontext-sensitive Informationsflusskontrolle unter Berücksichtigung der Deklassifikationen. Program Slicing mit Fluss-, Kontext- und Objekt-Sensitivität sowie die darauf aufbauende Informationsflusskontrolle wurden in ein Plugin der bekannten Entwicklungsumgebung Eclipse integriert, wodurch sich die Bedienbarkeit der Tools deutlich verbessert. Die Präzision und Skalierbarkeit der vorgestellten Verfahren wird in Fallstudien verdeutlicht.

Der vollständige Artikel zu dieser Arbeit erscheint im International Journal of Information Security [1].

Literatur

1. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security, vol. 8 (2009) doi: 10.1007/s10207-009-0086-1.

Inkrementelle, zweistufige Deadlock-Analyse für unvollständige Java Card 3.0 Programme*

Rebekka Neumann

Software Quality Lab, Universität Paderborn,
Fürstenallee 11, 33102 Paderborn, Deutschland
rneumann@s-lab.upb.de

Zusammenfassung Wir stellen eine Methode zur Verifikation von Software auf Smart Cards vor, die statische Programmanalyse und Model Checking miteinander kombiniert. Der Java Card 3.0 Standard (JC 3.0) führt Nebenläufigkeit als neue Eigenschaft von Java auf Smart Cards ein. Daher haben wir die Ermittlung von Deadlocks in JC 3.0 Programmen als unser erstes Verifikationsziel gewählt. Der auf JC 3.0 Smart Cards gebrannte Webserver stellt ein realistisches Analyse- und Evaluationsobjekt dar. Aus Sicht unserer Deadlock-Analyse betrachten wir den JC 3.0 Webserver als unvollständiges Programm, das durch JC 3.0 Webanwendungen vervollständigt werden kann, die auf Smart Cards geladen werden. Der Aufwand unserer Deadlock-Aanlyse wird sich durch das separate Analysieren des Webservers und das Wiederverwenden der Webserver-Analyseergebnisse beim Analysieren des vervollständigten Programms reduzieren. Da die Analyseergebnisse einer statischen Programmanalyse konservativ sind, handelt es sich bei manchen von unserer Deadlock-Analyse identifizierten Deadlocks nur um potentielle. Diese werden wir mit Hilfe eines Model Checkers weiter überprüfen, um endgültige, präzise Analyseergebnisse zu erhalten. Die vorgestellte Methode kann das Testen von JC 3.0 Programmen im Qualitätssicherungsprozess nicht vollständig ersetzen, aber die Testaktivitäten zur Aufdeckung von Deadlocks lenken und vereinfachen. Als ein erstes Ergebnis zeigen wir, dass 50% der statischen Analyseergebnisse für den JC 3.0 Webserver bei der Analyse des durch eine JC 3.0 Webanwendung vervollständigten Webservers wiederverwendet werden können.

1 Einleitung

Wir motivieren die Ermittlung von Deadlocks in JC 3.0 Programmen (siehe Abschnitt 2) und die Entwicklung einer neuen, inkrementellen, zweistufigen Deadlock-Analyse für unvollständige JC 3.0 Programme (siehe Abschnitt 3), die in Abschnitt 4 beschrieben wird. Erste Evaluationsergebnisse im Hinblick auf die Effizienz unserer Deadlock-Analyse stellen wir in Abschnitt 5 vor, bevor wir den Stand unserer Arbeit in Abschnitt 6 zusammenfassen und die nächsten Arbeitsschritte aufzeigen.

* Dieses Papier stellt eine Erweiterung des auf der TAP 2009 vorgestellten Kurzpapiers dar.

2 Ermittlung von Deadlocks in JC 3.0 Programmen

JC 3.0 [2] ist die neue Spezifikation von Sun für Java auf Smart Cards. JC 3.0 Smart Cards sind internetfähig, sie enthalten einen Webserver, der dazu dient, vertrauliche Daten von Smart Card-Besitzern zu verwalten und zu schützen (siehe Abbildung 1). Wir betrachten den auf JC 3.0 Smart Cards gebrannten Webserver mit seinen Bibliotheken als unvollständiges Programm, das durch JC 3.0 Webanwendungen vervollständigt werden kann. Ebenfalls neu in Java für

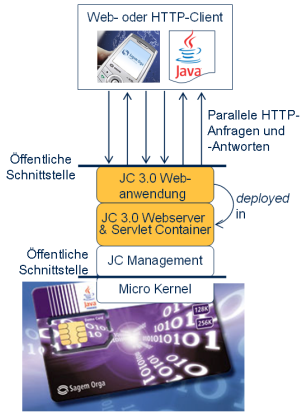


Abbildung 1. Überblick über JC 3.0 Architektur

Smart Cards ist das Konzept der Nebenläufigkeit. JC 3.0 Anwendungsentwicklern steht eine abgespeckte Version der Java 1.4 Nebenläufigkeitsanwendungsschnittstellen zur Verfügung, so werden z.B. Threads, jedoch keine Thread-Gruppen unterstützt. Die Qualitätssicherungsmaßnahmen für JC 3.0 Programme müssen diese Neuerungen berücksichtigen. Zum einen müssen sie Nebenläufigkeitseigenschaften von JC 3.0 Programmen überprüfen können. Zum anderen müssen sie mit der gestiegenen Komplexität der Programme auf JC 3.0 Smart Cards umgehen können.

Der Fokus unserer Arbeit liegt auf der Ermittlung von Deadlocks (als ein Beispiel für Nebenläufigkeitsfehler) in unvollständigen JC 3.0 Programmen. Wir untersuchen Deadlocks, die durch sich entgegenstehende gegenseitige Ausschlüsse und Lockobjekt-Anforderungen entstehen. In Java ausgedrückt betrachten wir Deadlocks, die durch fehlerhafte Verwendungen von `synchronized`- und `wait`-Anweisungen auftreten. Abbildung 2 zeigt ein Deadlock-Beispiel, das im Kontext einer Hotel-Gutschein-JC 3.0 Webanwendung auftreten kann, die auf der in [4] vorgestellten JC 3.0 Webanwendung aufbaut. Ein Hotel-Gutschein-Provider kann kurz nacheinander einem potentiellen Hotel-Besucher einen neuen Gutschein zur Verfügung stellen (siehe linke Seite der Abbildung) und sich statistische Informationen über Gutscheine des Hotel-Besuchers anzeigen lassen (siehe

rechte Seite der Abbildung). Werden diese beiden Aktionen verzahnt ausgeführt, kann es zum einem Deadlock kommen, da die Locks auf das `ServletContext`-Objekt der Webanwendung und die Hotel-Liste (Objekt `hotels`) in umgekehrter Reihenfolge in unterschiedlichen Ausführungskontexten und ohne gemeinsames Vorgänger-Lock-Objekt angefordert werden.

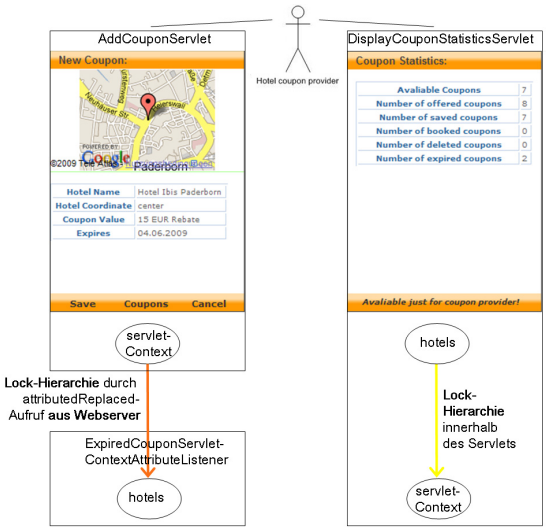


Abbildung 2. Deadlock-Beispiel in Hotel-Gutschein-JC 3.0 Webanwendung

3 Verwandte Arbeiten

3.1 Formale Methoden zur Ermittlung von Deadlocks in Java-Programmen

Es existieren mehrere Ansätze und Methoden zur Ermittlung von Deadlocks in Java-Programmen, von denen einige im folgenden vorgestellt werden. Williams et al. [9] schlagen eine statische Programmanalyse zur Aufdeckung von Deadlocks in Java-Bibliotheken vor, die jedoch die Beziehungen zwischen Bibliotheken und Benutzern der Bibliotheken einschränkt. So sind zum Beispiel Callbacks zwischen Bibliotheken und Bibliotheksbenutzern sowie Spezialisierungen von Bibliotheksklassen in Bibliotheksbenutzern verboten. Die in [8] beschriebene statische Programmanalyse zur Aufdeckung von Deadlocks ist auf vollständige Programme beschränkt. Beide Methoden sind unvollständig (*incomplete*), produzieren also „falsche Alarme“ (*false alarms*).

Im Gegensatz zu diesen statischen Programmanalysetechniken sind Software Model Checker wie zum Beispiel Java PathFinder [7] (JPF) vollständig (*complete*), jedoch durch den Aufbau des gesamten Programmzustandsraums weniger effizient in Bezug auf Laufzeit und Speicherverbrauch. Sie leiden unter dem sogenannten Problem der Zustandsraumexplosion (*state explosion problem*). Desweiteren erkennt JPF nur globale Deadlocks, also solche, die durch das Blockieren aller Threads des Programms zustande kommen. Lokale Deadlocks hingegen, also solche, die durch das Blockieren nur mancher Threads des Programms zustande kommen, kann JPF nicht aufdecken. Havelund [1] verwendet zur Laufzeit erhobene Trace-Informationen über Programme, um diese Deadlock-Aufdeckungseinschränkung zu überwinden.

3.2 Statische Programmanalyse für unvollständige Programme

Die Entwicklung von statischen Programmanalysen für unvollständige Programme ist durch die Wiederverwendbarkeit von Analyseergebnissen motiviert. Rountev et al. [5] schlagen die Generierung von sogenannten *summary information* für unvollständige Programme vor, die bei der Analyse der vervollständigten Programme wiederverwendet werden können. Thies [6] hat eine statische Programmanalyse für unvollständige Programme entwickelt und im Programmanalyse-Framework SAILDOWN implementiert, die im Gegensatz zu Rountevs Ansatz auf beliebige Programmausschnitte angewendet werden kann.

4 Deadlock-Analyse für unvollständige Java Card 3.0 Programme

Nach unserem Kenntnisstand wurde die Aufdeckung von Deadlocks in unvollständigen Programmen mittels statischer Programmanalyse kombiniert mit Model Checking zur Präzisierung der Analyseergebnisse noch nicht genauer untersucht. Wir stellen eine inkrementelle, zweistufige kombinierte formale Methode zur Aufdeckung von Deadlocks in unvollständigen JC 3.0 Programmen vor. Der unvollständige JC 3.0 Webserver und der durch JC 3.0 Webanwendungen vervollständigte Webserver dienen uns als Analyse- und Evaluationsgegenstände für die inkrementelle Anwendung unserer Deadlock-Analyse. Der Aufbau der Deadlock-Analyse wird im folgenden Unterkapitel beschrieben. Die innerhalb der Deadlock-Analyse verwendete Lockobjekt-Ermittlung mittels einer interprozeduralen Def-Use-Analyse wird in Unterabschnitt 4.2 vorgestellt.

4.1 Aufbau der inkrementellen, zweistufigen Deadlock-Analyse für unvollständige JC 3.0 Programme

Unsere Analyse zur Ermittlung von Deadlocks in unvollständigen und dann vervollständigten JC 3.0 Programmen besteht aus den folgenden Stufen (siehe Abbildung 3):

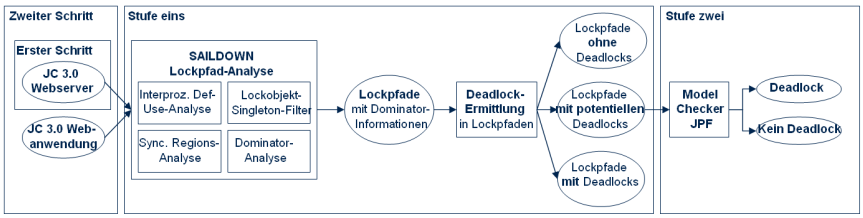


Abbildung 3. Überblick über inkrementelle, zweistufige Deadlock-Analyse

In Stufe eins ermittelt eine statische SAILDOWN-Analyse im ersten Schritt alle Lockpfade der abgeschlossenen Bestandteile des JC 3.0 Webservers, also solcher Webserver-Teile, deren Eigenschaften und Verhalten nicht mehr durch JC 3.0 Webanwendungen beeinflusst werden können. Ein Lockpfad besteht aus allen Objekten, über die ausgehend von einem JC 3.0 Webserver-Programmeintrittspunkt hierarchisch synchronisiert wird. Die in den abgeschlossenen Teilen des JC 3.0 Webservers enthaltenen Deadlocks werden durch Berechnung aller möglichen Kombinationen der Lockpfade ermittelt. Im zweiten Schritt werden alle Deadlocks für den durch eine JC 3.0 Webanwendung vervollständigten JC 3.0 Webserver berechnet. Dabei werden die Lockpfade für die abgeschlossenen Bestandteile des JC 3.0 Webservers aus dem ersten Schritt wiederverwendet.

Die Berechnung von Lockpfaden setzt voraus, dass in Hilfsanalysen ermittelt wurde, über welche Objekte im Programm synchronisiert wird (siehe Unterabschnitt 4.2 zur Beschreibung der interprozeduralen Def-Use-Analyse), und ob es sich bei den Objekten, über die synchronisiert wird, um einzelne Objekte oder um Objektmengen handelt (siehe Lockobjekt-Singleton-Filter in Abbildung 3). Desweiteren muss vorab berechnet werden, in welchen Codeabschnitten Locking-Verhalten enthalten ist (siehe Synchronized Regions-Analyse in Abbildung 3). Im Hinblick auf die Weiterverwendung der statisch berechneten Lockpfade innerhalb eines Model Checkers werden während der Lockpfad-Berechnung außerdem Dominator-Analyseinformationen ermittelt, die Auskunft darüber geben sollen, welche Kontrollflussverzweigungen genommen wurden, um über die in den Lockpfaden enthaltenen Objekte zu synchronisieren.

In Stufe zwei der Deadlock-Analyse werden alle Lockpfade mit potentiellen Deadlocks, also Deadlocks, für die die statische Analyse keine endgültigen Ergebnisse berechnen konnte, dem Model Checker JPF übergeben. Der Model Checker verwendet die Lockpfade, um den Programmzustandsraum zielgerichtet aufzubauen und so die in den Lockpfaden enthaltenen potentiellen Deadlocks endgültig zu überprüfen.

4.2 Lockobjekt-Ermittlung mittels interprozeduraler Def-Use-Analyse

Zur Ermittlung der Objekte, über die innerhalb eines Programms synchronisiert wird, haben wir die in [3] beschriebene interprozedurale Def-Use-Analyse

erweitert. Die interprozedurale Def-Use-Analyse bestimmt zu allen Definitionsstellen des Programms alle Verwendungsstellen innerhalb des Programms, die Def-Use-Ketten werden also interprozedural berechnet. Dies geschieht in drei Phasen (siehe Abbildung 4).

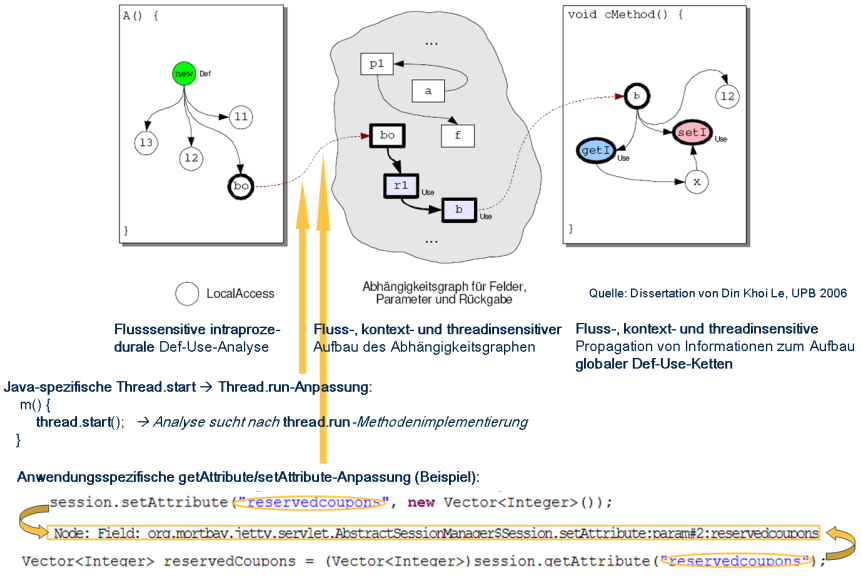


Abbildung 4. Überblick über Phasen der erweiterten interprozeduralen Def-Use-Analyse

In der ersten Phase werden den Definitionsstellen einer Methode (siehe zum Beispiel `new`-Anweisung im linken Teil der Abbildung 4) alle lokalen Verwendungsstellen innerhalb dieser Methode unter Berücksichtigung des Kontrollflusses zugeordnet. Resultat dieser flusssensitiven, intraprozeduralen Def-Use-Analyse sind alle lokalen Def-Use-Ketten von Methoden des Programms. Im linken und rechten Teil der Abbildung 4 sind die lokalen Def-Use-Ketten des Konstruktors der Klasse `A` beziehungsweise der Methode `cMethod` zu sehen. Die Knoten dieser lokalen Def-Use-Ketten stellen lokale Zugriffe dar (*LocalAccesses*, Schreib-/Lese-Zugriffe auf Operanden beziehungsweise das Ergebnis einer Methode basierend auf dem Stack-Modell einer Java Virtuellen Maschine). Die Kanten der lokalen Def-Use-Ketten repräsentieren die Def- beziehungsweise Use-Beziehungen.

In der zweiten Phase der interprozeduralen Def-Use-Analyse werden die Abhängigkeiten zwischen Methodenparametern und -rückgaben sowie Feldern fluss-, thread- und kontextinsensitiv berechnet und in einem Abhängigkeitsgraphen modelliert. Methodenparameter, Methodenrückgaben und Felder stellen die Kno-

ten des Graphen dar. Die Kanten des Graphen repräsentieren potentielle Flussinformationen von einem Knoten zu einem anderen. So symbolisiert zum Beispiel die Kante vom Methodenparameter `bo` zur Methodenrückgabe `r1` eine potentielle Zuweisung und Verwendung des Wertes von `bo` in `r1`.

Zwei Erweiterungen der interprozeduralen Def-Use-Analyse sind in Abbildung 4 dargestellt. Zum einen werden bei der Berechnung des Abhängigkeitsgraphen Java-spezifische Informationen über die Funktionsweise der Klasse `Thread` verwendet. Wird ein Thread durch Aufruf der Methode `Thread.start` gestartet, dann wird die `run`-Methode des Threads durch die Java Virtuelle Maschine ausgeführt. Die Parameter-, Rückgabe- und Feldinformationen müssen also an die Methode `run`-Methode des Threads propagiert werden. Zum anderen sind in einigen Klassen des JC 3.0 Webservers Methoden zum Setzen und Lesen von Feldern einer Klasse enthalten, die demselben Muster entsprechen: ein Feld wird durch einen eindeutigen Bezeichner identifiziert und mittels der Methode `setAttribute(String, Object)` gesetzt beziehungsweise mittels der `Methode` `getAttribute(String):Object` gelesen. Um die Definitionsstellen eines solchen Feldes präziser den Verwendungsstellen des Feldes zuordnen zu können, kodieren wir bei der Erstellung des Abhängigkeitsgraphen die eindeutigen Bezeichner des Feldes in den Knotenbezeichnungen der Methodenparameter (für den zweiten Parameter der Methode `setAttribute`) und der Methodenrückgaben (für den Rückgabewert der Methode `getAttribute`) (siehe unterste Zeile in Abbildung 4). Den eindeutigen Bezeichner des Feldes speichern wir bei der Verarbeitung der ersten Parameter der Methoden `getAttribute` und `setAttribute` zwischen (siehe zweitletzte Zeile in Abbildung 4).

In der dritten Phase der interprozeduralen Def-Use-Analyse werden lokale Def-Use-Ketten unter Verwendung des Abhängigkeitsgraphen zu globalen Def-Use-Ketten auf fluss-, thread- und kontextinsensitive Weise zusammengesetzt. Dazu werden lokale Zugriffe auf Methodenparameter und -rückgaben sowie Felder innerhalb einer Methode den zugehörigen Knoten des Abhängigkeitsgraphen zugeordnet. So werden zum Beispiel die lokalen Zugriffe im linken und rechten Teil der Abbildung 4 `bo` beziehungsweise `b` auf die Knoten des Abhängigkeitsgraphen `bo` beziehungsweise `b` in der Mitte der Abbildung 4 abgebildet.

Die interprozedurale Def-Use-Analyse kann trotz der flusssensitiven Bestimmung intraprozeduraler Def-Use-Ketten in der ersten Analysephase zur Bestimmung von interprozeduralen Def-Use-Ketten nebenläufiger Programme eingesetzt werden, da die lokale Def-Use-Analyse nur Thread-lokale Zugriffe behandelt.

5 Evaluation der Effizienz der Deadlock-Analyse für unvollständige JC 3.0 Programme

Wir haben eine einfache SAILDOWN-Analyse mit dem Ziel durchgeführt, den Anteil an wiederverwendbaren statischen Analyseergebnissen des JC 3.0 Webservers (inklusive der JC 3.0 API) zu bestimmen. Dazu haben wir ein sogenanntes *partial analysis system* (PAS, ähnlich einem Verband in der Datenflussanalyse)

bestehend aus vier in einer linearen Kette angeordneten Datenflusselementen modelliert (siehe drei davon in linkester Spalte in Tabelle 1), die die konstanten Anteile des Synchronisationsverhaltens von Methoden repräsentieren. Um

Locking-Verhalten einer Methode	Anzahl *fixed* SAMs	Anteil *fixed* SAMs	Anzahl *fixed* und *open* SAMs	Anteil *fixed* und *open* SAMs
locksNoObj	2501	89,2%	2936	53%
locksObj	**174**	6,2%	703	**12,7%**
nativeMethod	130	4,6%	1904	34,3%
Summe	2805	**50,6%**	5543	100%

Tabelle 1. Evaluationsergebnisse Locking-Verhalten des JC 3.0 Webservers

das PAS zu lösen, wird ein sogenanntes *single aspect model* (SAM) für jede Methode des JC 3.0 Webservers berechnet, das den direkten Effekt des Synchronisationsverhaltens einer Methode als konstanten Term und die Effekte von Methodenaufrufen als Anwendung weiterer SAMs zum Ausdruck bringt. Die Effekte von dynamisch gebundenen Methoden werden in einem PAS als sogenannte *method familiy single aspect models* (MF-SAMs) modelliert. Alle SAMs und MF-SAMs werden soweit wie möglich bei der Berechnung des Analyseergebnisses vereinfacht. Für den JC 3.0 Webserver sind ca. 50% der Analyseergebnisse des PAS abgeschlossen (*fixed*, siehe unterste Zeile und mittlere Spalte in Tabelle 1) und können folglich bei der Analyse des durch eine JC 3.0 Webanwendung vervollständigten Webservers wiederverwendet werden. 13% der SAMs und MF-SAMs enthalten Synchronisationsverhalten (siehe mittlere Zeile und linkeste Spalte in Tabelle 1), wovon 25% abgeschlossen sind (siehe mittlere Zeile und zweite Spalte von links in Tabelle 1), so dass eine inkrementelle Anwendung unserer Deadlock-Analyse auf einen erweiterten, vervollständigten Analysekontext sinnvoll und effektiv erscheint.

6 Zusammenfassung und Ausblick

Wir haben eine inkrementelle, zweistufige Deadlock-Analyse vorgestellt, die statische Programmanalyse und Model Checking miteinander kombiniert. Die Wiederverwendbarkeit von 50% der statischen Analyseergebnisse des JC 3.0 Webservers ermutigt uns und rechtfertigt es, diesen Ansatz weiter zu untersuchen. Grundlegende Analysen zur Berechnung von Synchronisationsabschnitten und Mengen von Objekten, über die synchronisiert wird, haben wir bereits implementiert beziehungsweise erweitert. Als nächstes werden wir eine SAILDOWN-Analyse zur Berechnung der Lockpfade des JC 3.0 Webserers und einer JC 3.0 Webanwendung modellieren, um diese Lockpfade dann im Hinblick auf Deadlocks zu untersuchen und potentielle Deadlocks durch einen Model Checker überprüfen zu lassen. Dies wird es uns ermöglichen, unser Deadlock-Analyse-

Tool in den Testprozess von JC 3.0 Programmen zu integrieren, um diesen zu vereinfachen.

Literatur

1. K. Havelund. Using runtime analysis to guide model checking of Java programs. pages 245–264. Springer, 2000.
2. http://java.sun.com/javacard/3.0/. Java Card 3.0 specification. 09.02.2009.
3. Dinh Khoi Le. *Automatische Verteilung mehrsträngiger Java-Programme.* PhD thesis, Universität Paderborn, 2006.
4. R. Oeters, U. Kastens, C. Rust, and al. Benefits from realizing a location-based hotel service with Java Card^{TM} 3.0. Technical report, Univ. of Paderborn, 2008.
5. A. Rountev, M. Sharp, and G. Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, LNCS 4959, pages 53–68, 2008.
6. M. Thies. *Combining Static Analysis of Java Libraries with Dynamic Optimization.* PhD thesis, University of Paderborn, 2001.
7. W. Visser and K. Havelund. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12. Press, 2000.
8. C. von Praun. *Detecting synchronization defects in multit-hreaded object-oriented programs.* PhD thesis, ETH Zürich, 2004.
9. A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.

Church vs. Curry
A Modern Introduction to the Fundamental Dichotomy of Type System Paradigms

Baltasar Trancón y Widemann

Universität Bayreuth
baltasar.trancon@uni-bayreuth.de

Abstract. There are two fundamental, complementary paradigms underlying the type systems of formal languages; one attributed to Church and one attributed to Curry. The Church style integrates the value and type level of the language, whereas the Curry style keeps them separate. This distinction entails wide-ranging differences in the usage of advanced type-related features, the algorithmic tractability of semantic problems, the amount of type information requested from the programmer, and the kind of expressive power expected from the language design and tools. We give an overview of the theoretical comparison of the paradigms, and examples of typical issues that arise out of their implementation in practical languages.

1 Introduction

Throughout the history of type systems for formal languages, there have been two complementary paradigms, each of which refers to one of the founding fathers of computer science, Church and Curry, respectively. The syntax and semantics of function calculi based on either paradigm have been explored in great detail in the encyclopedic, albeit slightly dated, work of Barendregt [1]. The present article summarizes the fundamental differences, and elucidates the obstacles in the search for a synthesis by examples in several contemporary languages.

1.1 What Type Are (Not)

The issues to be elaborated below have historically arisen in, and are most clearly represented by, the simply typed lambda calculus. That is, function abstraction and application and higher-order function types are the first-class citizens at the instance and type level, respectively, of the systems of interest. This is quite a narrow view on types, but it suffices to demonstrate the relevant phenomena.

Most importantly on the negative side, we do not consider dynamic aspects at all. Concerning the questions addressed in the present article, types are a disciplined form of set theory that provides semantic models of a formal language. Under this assumption, *dynamic type information* becomes a misnomer and should rather be called *metadata*. This exclusion extends to real-world type

system categories that refer to dynamic typing directly or indirectly, such as "weak typing" or "duck typing".

We do not consider object orientation, either. Firstly, object classes as types are related in an intricate way to dynamic typing (via virtual methods and overriding of behavior). Secondly, the usual presentations of object-oriented type systems, for all their pragmatic virtues, are mathematically unwieldy and not helpful for elucidating fundamental theoretical issues such as the dichotomy discussed here.

1.2 The Curry–Howard Correspondence and Type-Related Software Tools

The Curry–Howard correspondence (sometimes called isomorphism) is the formal interpretation of a single language both as a constructivist logical system and a type system. Under this dual view, propositions correspond to types and proofs to values. Hence the study of type systems has immediate meaning not only for a programming language compiler, but also for tools that discover, store or validate formal proofs.

1.3 Typing à la Church, Typing à la Curry

The distinction boils down to a deceptively simple syntactic criterion:

> *A type system is à la Church if the type of each variable is mentioned explicitly at the point where the variable is introduced, or à la Curry otherwise.*

This innocuous distinction makes an enormous difference for the role of types, summarized in table 1.

	Subject–Type Relation	Axiomatic Types (Variables)	Derived Types (Expressions)
à la Church	integration	declaration	evaluation
à la Curry	association	inference	unification

Table 1. Type System Mechanisms

In the Church style, types (of variables) are an integral part of value expressions. The type of a compound expressions is determined bottom-up by the types of its constituents. In the Curry style, types are a separate layer of information that is usually left implicit. Types cannot always be computed bottom-up, but are calculated like a logical puzzle, using unbound variables, inference and unification.

1.4 Type-Related Problems

There are four basic formal problems that relate to expressions of a typed language. These problems can be read off the structure of a type judgment, that is, a proposition of the form $\Gamma \vdash a : t$ pronounced "in *context* Γ, the *subject* a has the *type* t".

Type Reconstruction The construction problem where context and subject are given is called *type reconstruction*. The algorithmic task is to find some type that satisfies the judgment. A subject a is called *typeable* or *well-typed* in context Γ if and only if there is a solution.

For some type systems the solution to the type reconstruction problem is unique, and simply called the type of subject a in context Γ. Type ambiguity is usually resolved by imposing a partial order on types such that there is a unique minimal solution, called the *principal* type of subject a in context Γ. A type system with principal types for all typeable subjects is said to have the *principal type property*.

The problem of type reconstruction occurs directly whenever a definition written in a typed programming language is analyzed by a compiler.

Type Inhabitation The construction problem where context and type are given is called *type inhabitation*. The algorithmic task is to find some subject that satisfies the judgment. A type is t is called *inhabited* in context Γ if and only if there is a solution. A type system is called *consistent* if and only if there are uninhabited types in the blank context. This can be understood in terms of the Curry–Howard correspondence: In a consistent logical system, not all propositions are theorems.

The problem of type inhabitation occurs indirectly (by the Curry–Howard correspondence) whenever a formal proposition is verified by a prover.

Type Check The problem where subject and type are given is called *type check*. The context may be given as well, leaving a pure decision problem, or a minimal context (weakest precondition) may be deduced.

The problem of type check occurs indirectly (again by the Curry–Howard correspondence) whenever a formal proof is validated by a proof checker.

1.5 Translation

There are canonical translations between the two paradigms. Because of the gradient in explicit information, it is easier to translate from the Curch to the Curry style than vice versa.

Type Erasure By simply erasing all type declarations for variables from an expression typed à la Church, a corresponding expression typed (or rather typeable) à la Curry is obtained.

Type Annotation By recording the solution to the type reconstruction problem for an expression typed à la Curry, a corresponding expression typed à la Church is obtained. This translation is only canonical if the source type system has the principal type property.

2 Theoretical Dialectic

2.1 Example: Type Reconstruction

Consider the following example, which the reader familiar with functional programming might recognize as the application of the *twice* function to the *successor* function, typed à la Curry:

$$\big(\lambda x.\, \lambda y.\, x(x\ y)\big)(\lambda z.\, z + 1)$$

Here, the primitive constant 1 is of type $\mathbb{N}$ and the primitive binary operation $+$ is of type $(\mathbb{N}, \mathbb{N}) \to \mathbb{N}$.

Type reconstruction à la Curry employs both bottom-up deduction rules

$$\frac{\Gamma, (x : t) \vdash a : u}{\Gamma \vdash (\lambda x.\, a) : (t \to u)}\text{Abs} \qquad \frac{\Gamma \vdash a : (t \to u) \qquad \Gamma \vdash b : t}{\Gamma \vdash (a\, b) : u}\text{App}$$

and unification steps that may cause information flow in arbitrary direction and distance across the expression syntax tree. Figure 1 depicts the situation: Syntax tree edges are shown as solid lines, unification-based dependencies as dashed lines. Arrows indicate type information flow.

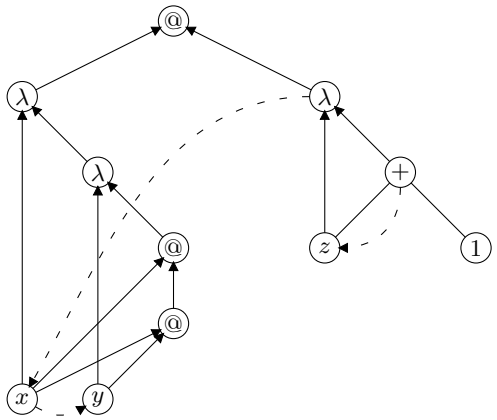


Fig. 1. Syntax Tree and Type Information Flow

The same expression, typed à la Church:

$$\big(\lambda(x : \mathbb{N} \to \mathbb{N}).\, \lambda(y : \mathbb{N}).\, x(x\ y)\big)(\lambda(z : \mathbb{N}).\, z + 1)$$

For the Church version, type check and reconstruction are a matter of straightforward bottom-up calculation. The amount of explicit type information is modest in this simple case. See below for more type-dominated examples.

2.2 Type System Extensions and Algorithmic Tractability

Even though the simply typed lambda calculus is not a very powerful type system on its own, there is a vast array of well-understood extensions, some available in either Church or Curry style exclusively, some in both. See [1] for a detailed exploration of the classical extensions. Of particular renown in the Church world are the dimensions of Barendregt's *lambda cube*, which add polymorphism, type functions and dependent types, respectively, and culminate in Coquand's *calculus of constructions* (CC). Equally famous in the Curry world, but far less powerful, is the Hindley–Milner type system of ML. This system owes its fame to a sad truth: For all Curry-style extended systems of substantial power, the type problems specified above are either trivial or undecidable. The Hindley–Milner system was the first Curry system with a nontrivial extension, namely first-order polymorphic types, that has both principal types and decidable type inference. There are few others.

2.3 Example: Type Parameterization

Type parameterization, that is, variable binding (abstraction) and substitution (instantiation) in type expressions, is one of the most useful and historically pervasive extensions of the simple type system. The two different interpretations of the role of types are reflected in the usage of type parameters à la Church and à la Curry, respectively.

Type Parameters à la Church The central notion in Church-style type parameterization is the *dependent product* which binds type variables. A dependent product type is a type expression of the form $\prod \alpha : t.\, u$, where α is a (type or instance) variable, t is a type (or kind) and u is a type scheme in which α may occur freely. The set-theoretic intuition behind the dependent product is that an instance of $\prod \alpha : t.\, u$ is a family (f_α), indexed by instances i of t, such that f_i is an instance of $u[\alpha \leftarrow i]$. In other words, $\prod \alpha : t.\, u$ is the space of *choice functions* for the family $(u_\alpha)_{\alpha \in t}$. Abstraction and instantiation are fully explicit in the syntax.

As an interesting consequence, an ordinary function type $t \to u$ may defined as $\prod \alpha : t.\, u$ where α does not occur freely in u. That is, type parameterization already entails simple types in the Church style, and both function application and type instantiation are merely special cases of dependent product indexing.

Type Parameters à la Curry The central notion in Curry-style type parameterization is *parametric polymorphism*. A polymorphic type is a type expression of the form $\forall \alpha.\, t$, where α is a type variable and t is a type scheme where α may occur freely. The set-theoretic intuition behind polymorphism is that an instance of $\forall \alpha.\, t$ is also (or can be "coerced" to) an instance of $t[\alpha \leftarrow u]$ where u is any type. Abstraction and instantiation are semantical notions: they take

the form of implicit conversions that occur at specified states of a type analysis algorithm; the archetypical example is Milner's algorithm W [2].

There is no simple relation between parametric polymorphism and function type construction. Type parameterization is independent of simple types in the Curry style.

Minimal Example: Self-Application The behavior of type parameterization in the two paradigms can be highlighted with a very simple example expression that has a number of astonishing semantical properties, namely the self-application combinator

$$\omega = \lambda x.\, x\ x$$

Typing this expression in various Curry-style type systems yields radically different interpretations:

- In the simple type system or the Hindley–Milner system, the combinator is not typeable.
- In the first-order polymorphic System F, the combinator has the type

  $$\forall\alpha.\, (\forall\beta.\, \beta \to \beta) \to \alpha \to \alpha$$

  That is, the combinator is polymorphic over α takes, a polymorphic function x and instantiates it (in some nontrivial way left as an exercise to the reader) to give the subexpression $x\ x$ the type $\alpha \to \alpha$.
- In a type system with least fixpoints, the combinator has the type

  $$\mu\alpha.\, \alpha \to \alpha$$

  That is, the principal type t such that $t = (t \to t)$. Its argument x has the same type.

Translating the expression via type annotation to Church-style systems also yields two alternatives, philosophically known as the *predicate* and *impredicative* interpretation, respectively:

- The predicative interpretation is

  $$\omega = \lambda(\alpha : *).\, \lambda\big(x : \textstyle\prod (\beta : *).\, \beta \to \beta\big).\, x\ (\alpha \to \alpha)\ (x\ \alpha)$$

  Its principal type is

  $$\textstyle\prod(\alpha : *).\, \big(\prod (\beta : *).\, \beta \to \beta\big) \to \alpha \to \alpha$$

  This interpretation corresponds roughly to the System F type above. The polymorphism of the combinator and the instantiations of its polymorphic argument are explicit. This interpretation is valid in any sufficiently strong Church-style type system (having second-order type-indexed dependent products).

– The impredicative interpretation is

$$\omega = \lambda\big(x : \prod(\alpha : *).\, \alpha \to \alpha\big).\, x\ \big(\prod(\alpha : *).\, \alpha \to \alpha\big)\ x$$

Its principal type is

$$\big(\prod(\alpha : *).\, \alpha \to \alpha\big) \to \big(\prod(\alpha : *).\, \alpha \to \alpha\big)$$

This interpretation corresponds roughly to the fixpoint type above. The argument to the combinator is polymorphic and is used both instantiated with its own type and uninstantiated. Hence the type variable α in the dependent product $t = \prod(\alpha : *).\, \alpha \to \alpha$ is implicated to range over all types including t itself. Church-style type systems where this interpretation is valid are called *impredicative*.

Assessment Typing à la Curry is obviously far more *concise*: The preceding paragraph shows how, in the pure Church style, even a very simple expression like ω is cluttered with (apparently redundant) type annotations. On the one hand, declaring the types of variables where they are introduced is considered good practice from the viewpoint of programming and documentation methodology. On the other hand, the inference of instantiations of parameterized objects can save the programmer from a great load of unproductive work.

Note that typing à la Church is more *precise*, which can turn into a decisive advantage in some circumstances, in particular for didactical reasons; for an example see section 3.4.

3 Practical Synthesis

We have argued that pure forms of either paradigm are unattractive to the working programmer, because he is forced to choose between the twin evils of verbosity of the syntax and undecidability of the semantics. It is thus only natural that practical languages strive to provide the best of both worlds. Not surprisingly, this is a challenging enterprise.

1. No example is known that demonstrates a truly balanced synthesis of the two paradigms. Practical languages typically implement one paradigm as a basis and emulate the other as far as possible.
2. The emulation of the second paradigm is necessarily heuristic and incomplete, and can introduce a great variety of complex limitations and pathological cases.

The remainder of this section illustrates examples of the typical strategies, limits and pitfalls of emulation by examples from practical languages. Rather that nailing down the technical details, we describe effects that might be observed by any inquisitive programmer.

3.1 Example: Haskell

Haskell is a modern functional programming language notable for its lazy evaluation strategy and complex type system. The type system is based on (Curry-style) Hindley–Milner, augmented with type classes and (beyond the standard Haskell98) many features such as functional dependencies, higher-rank-polymorphism, and kind inference. As a consequence of the numerous extensions, type inference is incomplete: there is the dreaded monomorphism restriction, and some definitions require explicit type signatures. Explicit typing is fine and should even be considered good practice for top-level definitions, but unfortunately it is not always viable for local definitions; type variables do not scope properly, even when the experimental language feature *scoped type variables* is turned on. This is due to the implicit abstraction step of Hindley–Milner type inference that binds type variables implicitly (and sometimes too early), out of control of the programmer. As an example, consider the following type classes from the standard prelude that specify enumerated and bounded types, respectively:

$$\begin{array}{ll} \textbf{class } \mathit{Enum}\ \alpha\ \textbf{where} & \textbf{class } \mathit{Bounded}\ \alpha\ \textbf{where} \\ \quad \mathit{toEnum} :: \mathit{Int} \to \alpha & \quad \mathit{minBound} :: \alpha \\ \quad \mathit{fromEnum} :: \alpha \to \mathit{Int} & \quad \mathit{maxBound} :: \alpha \end{array}$$

Now tentatively define a subclass of combinatorial types:

$$\begin{array}{l} \textbf{class } (\mathit{Enum}\ \alpha, \mathit{Bounded}\ \alpha) \Rightarrow \mathit{Combi}\ \alpha\ \textbf{where} \\ \quad \mathit{count} :: \mathit{Int} \\ \quad \mathit{listAll} :: [\alpha] \end{array}$$

A definition for the method *listAll* can be given easily using Haskell's enumerated range notation:

$$\mathit{listAll} = [\mathit{minBound} \mathinner{..} \mathit{maxBound}]$$

But the specification of *count* is illegal: Because the class-constituting type variable α does not occur in its signature, the method cannot be dealt with by Haskell's type inference mechanism. The Haskell solution is to equip *count* with a dummy parameter of type α that is ignored by its implementation. Since Haskell is lazy, the idiomatic dummy value is *undefined*, a special value that crashes when evaluated. This is a workable solution, but of questionable elegance. When defining implementations for combinatorial product and sum constructors, a number of other interesting issues come up.

$$\textbf{instance } (\mathit{Combi}\ \alpha, \mathit{Combi}\ \beta) \Rightarrow \mathit{Combi}\ (\alpha, \beta)\ \ldots$$

The details are left as an exercise to the reader, but the common baseline is that type variables do not work like ordinary variables that have clearly defined scopes and points of introduction.

3.2 Example: Java

Java is arguably one of the most popular contemporary programming languages. It features an object-oriented type system with (nominal) subtyping and Church-style variable declarations. Beginning with a version of the language definition marketed as Java 5 [3], type parameters have been added in a way that maintains the delicate balance between expressive power and backwards compatibility [4]. Summarizing the Java type system formally is a nightmare, because of several issues:

1. The interference between subtyping and parameterization is inherently complicated.
2. A form of type erasure is employed for backwards compatibility, such that overloading, overriding and dynamic type information do not bring type parameters to their full potential.
3. The array type constructor has not been unified with other parametric mechanisms, nor is it erased. To make matters worse, the historical array typing rules are unsound.

We shall focus on the parameterization mechanism of Java 5. Type variables may be bound and instantiated for definitions of either classes or individual methods. A limited form of inference is available, but only for method instantiation; the other three constructs are always explicit. A more general inference mechanism that reconstructs the types of variables is available as an academic prototype [5], but has not been incorporated into a production compiler.

The designers of Java 5, having backwards compatible syntax high on their priority list, chose to use the same angled-bracket notation for all four constructs. It is debatable whether this solution is elegant or confusing. As an example, consider the following fragment of the Java standard library dealing with lists, where parameterizations are annotated to distinguish class abstraction (†), class instantiation (‡), method abstraction ($^\sharp$) and method instantiation ($^\flat$):

```
package java.util;
interface List⟨E⟩† {...}
class Collections {
  static ⟨T⟩♯ List⟨T⟩‡ emptyList() {...}
  static ⟨T⟩♯ List⟨T⟩‡ unmodifiableList(List⟨? extends T⟩‡ l) {...}
  ⋮
}
```

Now consider the following simple Java program that calls the factory method `emptyList`, without instantiating its type parameter, in five subtly different contexts:

```
import java.util.*;
import static java.util.Collections.*;
```

```
class Test {
  List⟨String⟩‡ foo() {
    List⟨String⟩‡ c = emptyList();                    // Context 1
    c = emptyList();                                  // Context 2
    c = unmodifiableList(emptyList());                // Context 3
    bar(emptyList());                                 // Context 4
    return emptyList();                               // Context 5
  }
  void bar(List⟨String⟩‡ x) { }
}
```

Of these, the calls in contexts 1, 2 and 5 are legal; the type inference algorithm succeeds in finding the correct instantiation $\langle \text{String} \rangle^\flat$. Context 4 fails to compile with the error message

bar(List⟨String⟩‡) cannot be applied to (List⟨Object⟩‡)

which indicates that the type inference algorithm did not (and, in fact, is not allowed by the language specification to) use the available information to find the correct instantiation. Context 3 fails also, but produces the more indirect error message

incompatible types: found List⟨Object⟩‡, required List⟨String⟩‡

located at the assignment operator, indicating that the erroneous inference for the call of `emptyList` has propagated to foil an otherwise successful inference for the call of `unmodifiableList`. The solution in both cases is to annotate the offending expressions in contexts 3 and 4 with an explicit instantiation. For reasons too obscure to be explained here, the expression must be augmented to

Collections.⟨String⟩♭emptyList()

3.3 Example: Coq

Coq [6] is a sophisticated proof management system for constructivistic higher-order logic. It is based on a powerful Church-style type system that combines Coquand's *CC* with inductive data types. It features a user-friendly inference mechanism, but unfortunately the heuristics and technical limitations are not stated clearly. The Coq documentation [7] only promises cryptically:

> ***1.2.11 Inferable subterms***
> *Expressions often contain redundant pieces of information. Subterms that can be automatically inferred by Coq can be replaced by the symbol "_" and Coq will guess the missing piece of information.*

3.4 Example: Tofu

Tofu is an experimental functional language based on research done at the Software Quality Research Laboratory, Limerick [8]. It is intended as an executable formalism for mathematical documentation and specification of software. It does not contain recursion, nontermination or partial functions, but its type system is rich enough to admit functional programming at a decent level of power and abstraction, nevertheless.

Similar to Coq, Tofu is based on an advanced Church-style type system (CC), but comes with a specific focus on program compilation (to Java) rather than proof manipulation. Tofu features several mechanisms to compensate for the verbosity of the Church style:

1. A type declaration can be stated for a variable *name*, and shared between multiple introductions of homonymous variables.
2. Dependent product indices can be declared *optional*: either there is an explicit instantiation (using square brackets like LaTeX) or inference from the context is attempted. The inference algorithm is based on pattern matching on the types of following arguments. The strategy is simple, but effective in inferring the type parameters of most typical polymorphic functions.

Consider the following declarations, where both features are employed:

$$\begin{aligned}&\textbf{var } \alpha : * \\ &\mathit{fold} : \textstyle\prod[\alpha].\, (\alpha \to \alpha) \to \alpha \to \mathbb{N} \to \alpha\end{aligned}$$

The definition of *fold* is not shown here. It is meant to implement the catamorphism of the natural numbers, that is

$$\mathit{fold}\ f\ e\ n \rightsquigarrow f^n(e)$$

Given the successor function and a variable ranging over unary functions on naturals,

$$\mathit{succ} = \lambda(n : \mathbb{N}).\, n + 1 \qquad\qquad \textbf{var } f : \mathbb{N} \to \mathbb{N}$$

one may define a *primitively recursive* (sic) Ackermann function

$$\mathit{ack} = \mathit{fold}\ (\lambda f.\, \mathit{fold}\ f\ (f\ 1))\ \mathit{succ}$$

This is neither a paradox nor a refutation of the famous theorem that the Ackermann function is not primitively recursive; so there must be some cheating involved. Annotation with the omitted optional instantiations reveals the trick:

$$\mathit{ack} = \mathit{fold}\ [\mathbb{N} \to \mathbb{N}]\ (\lambda f.\, \mathit{fold}\ [\mathbb{N}]\ f\ (f\ 1))\ \mathit{succ}$$

The first actual type parameter $\mathbb{N} \to \mathbb{N}$ is the space of total unary functions on naturals, and not part of the universe of traditional first-order recursion theory, as it cannot be Gödelized. Hence the first instance of *fold* is not a legal component of a first-order primitively recursive function, even though it is perfectly legal in a higher-order setting such as Tofu.

4 Conclusion

We have explored the rift between the Church and Curry styles of type systems. In its pure form, each paradigm has certain properties that render it virtually useless in practice, either semantically or ergonomically. In a practical language, the choice of paradigm determines which kind of expressive power can be harnessed by construction, and which can only be emulated partially by heuristic tool support.

In the Church style, advanced type constructs are typically easy to add, whereas redundant type information is a serious ergonomical impediment, and its inference a difficult task. In the Curry style, concise notation comes for free, but adding constructs or allowing for disambiguating type information without compromising either the principal type property of the decidability of type inference is a challenge.

A language that achieves a balanced synthesis of the two paradigms has yet to be designed. We are certainly looking forward to it.

Acknowledgments

Thanks to Sebastian Fischer, CAU, Kiel, for a deeper insight into Haskell hacker lore.

References

1. Barendregt, H.P.: Lambda calculi with types. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science, Oxford University Press (1992) 117–309
2. Milner, R., Damas, L.: Principal type-schemes for functional programs. In: POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on principles of programming languages, ACM (1982)
3. Gosling, J., Steele, G., Bracha, G.: Java Language Specification, Third Edition. Prentice–Hall (2005)
4. Bracha, G.: Generics in the Java programming language (2004) `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.
5. Plümicke, M., Bäuerle, J.: Typeless programming in Java 5.0. In: PPPJ '06: Proceedings of the 4th international symposium on principles and practice of programming in Java, New York, NY, USA, ACM (2006) 175–181
6. ADT Coq: Coq (1984–2009) `http://coq.inria.fr`.
7. Coq Development Team: Coq reference manual, version 8.2 (2009) `http://coq.inria.fr/coq/distrib/current/refman`.
8. Trancón y Widemann, B., Parnas, D.: Tabular expressions and total functional programming. In Chitil, O., Horváth, Z., Zsók, V., eds.: Implementation and Application of Functional Languages (IFL 2007), Revised Selected Papers. Volume 5083 of Lecture Notes in Computer Science., Springer (2008) 219–236

Reversible Programming Languages

Robert Glück

University of Copenhagen

The principles of reversible programming languages are explicated and illustrated with reference to the design of a high-level imperative language, Janus. The fundamental properties for such languages include backward as well as forward determinism and reversible updates of data. The unique design features of the language include explicit postcondition assertions, direct access to an inverse semantics and the possibility of clean (i.e., garbage-free) computation of injective functions. We suggest the clean simulation of reversible Turing machines as a criterion for computing strength of reversible languages, and demonstrate this for Janus. We show the practicality of the language by implementation of a reversible fast Fourier transform. Our results indicate that the reversible programming paradigm has fundamental properties that are relevant to many different areas of computer science.

Joint work with Tetsuo Yokoyama and Holger Bock Axelsen [1].

References

1. Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In *Conference on Computing Frontiers. Proceedings*, pages 43–54. ACM Press, 2008.

Hardware/Software–Co-Design mit den MicroCore–Prozessor

Ulrich Hoffmann

fhwedel University of Applied Sciences
uh@fh-wedel.de,
WWW home page: http://www.fh-wedel.de/~uh

Zusammenfassung MicroCore ist ein in VHDL formulierter Mikroprozessor für den Einsatz in FPGA–basierten eingebetteten Systemen. Seine Architektur erlaubt eine weitgehende Skalierung der Adressräume und der Datenwortbreite unter Beibehaltung des Instruktionssatzes. Damit kann MicroCore gut an die jeweiligen Anforderungen und Ressourcenbeschränkungen angepasst werden.
Durch den Einsatz in FPGAs bietet es sich an, MicroCore um kundenspezifische Hardware zu erweitern, um so einen möglichst geringen Ressourcen–Verbrauch zu erreichen. Solche Hardware–Erweiterungen können als Prozessor–Peripherie realisiert werden, auf die MicroCore mit Ein-/Ausgabe–Befehlen zugreift, oder aber auch durch benutzerdefinierte MicroCore–Instruktionen, die diese Hardware ansprechen. Die reguläre MicroCore–Architektur stellt dafür ein zugängliches Umfeld zur Verfügung, in das sich neue Instruktionen gut einpassen. Das strukturierte, ingenieurmäßige Vorgehen ist dabei folgendermaßen: Neue Instruktionen werden zunächst durch Unterprogramme simuliert, um die geeignete Hardware/Software–Partitionierung und die gewünschte Schnittstelle zur Software zu ermitteln. Anschließend wird statt eines allgemeinen Unterprogrammaufrufs eine der 32 vorhandenen sog. User–Instruktionen (USR, Traps) eingesetzt. Dies sind Instruktionen, die in der MicroCore–Standardkonfiguration einen Unterprogrammaufruf an eine fest–vorgegebene Adresse auslösen. Im letzten Schritt wird die User–Instruktion in VHDL so abgeändert, dass sie keinen Unterprogrammsprung mehr auslöst, sondern statt dessen die gewünschte, ebenfalls in VHDL definierte Hardware anspricht. Die Schnittstelle zur Software bleibt unverändert.
Die Programmentwicklung für MicroCore erfolgt in einer interaktiven Host-Target–Konstellation: Ein auf dem Host–System befindlicher interaktiver Cross–Compiler erzeugt Maschinencode für das MicroCore–Target–System. Dieser Maschinencode kann über ein sog. Umbilical–Link in das Target–System geladen werden. Dort können dann unter Kontrolle eines Monitor–Programms einzelne Funktionen interaktiv und auch schrittweise direkt im Zielsystem mit der verfügbaren wirklichen Hardware ausgeführt werden.

1 MicroCore

MicroCore [1] ist ein stack–orientierter Mikroprozessor für den Einsatz in eingebetteten Systemen. Er ist in VHDL definiert und kann für FPGAs verschiedener Hersteller[1] synthetisiert werden.

Anders als die sog. Softcores der FPGA–Hersteller steht MicroCore nicht nur als parametrisierbare Netzliste sondern vollständig im Quellcode zur Verfügung. Dies ermöglicht ein durchgängiges Verständnis seiner Funktionsweise und eine für die jeweilige Aufgabenstellung sinnvolle Adaption und Erweiterung von MicroCore etwa durch spezifische auf die Aufgabe zugeschnittene Peripherie–Hardware. Wie

[1] Bereits in Anwendung sind (Stand 2009) FPGAs von Actel, Altera, Lattice und Xilinx.

solche Erweiterungen strukturiert durchgeführt werden können, soll im folgenden erläutert werden.

Auf der Software–Seite wird `MicroCore` durch eine Tool–Chain unterstützt: Ein parametrierbarer Forth–Crosscompiler übersetzt auf Desktop–Computern Forth–Programme in `MicroCore`–Maschinencode. So erstellte Anwendungsprogramme werden zusammen mit einem Realtime–Kernel auf die Zielsysteme geladen und dort ausgeführt. Zur Fehlersuche existiert ein interaktiver Debugger. Ein C–Compiler ist in Entwicklung.

1.1 Die `MicroCore`–Architektur

`MicroCore` besitzt eine Harvard–Architektur, bei der Daten– und Programmspeicher voneinander getrennt sind. Abbildung 1 zeigt die wesentlichen Komponenten.

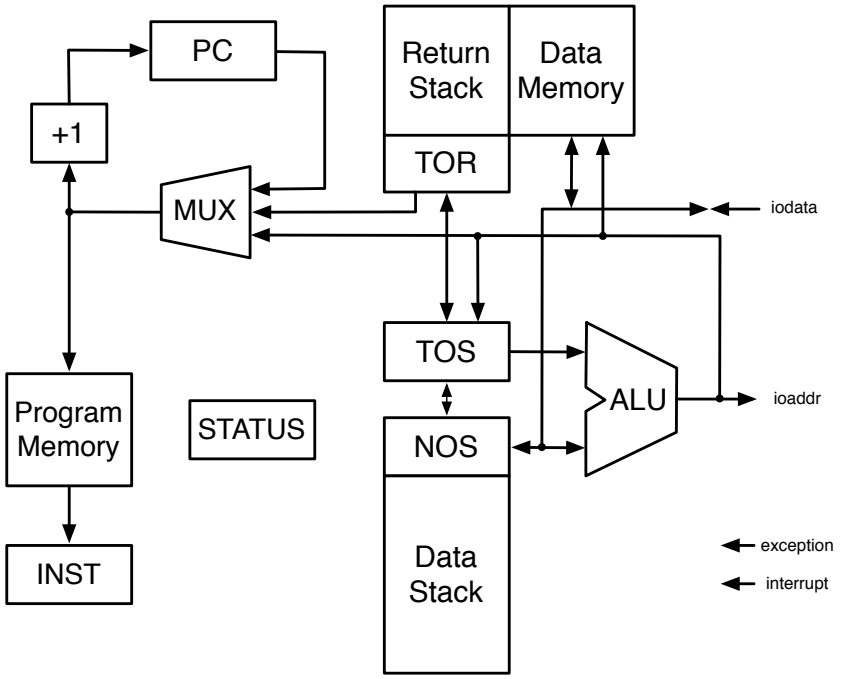


Abbildung 1. Die `MicroCore`–Architektur

Auffälligstes Merkmal dieser Architektur sind die beiden in Hardware ausgeführten Stacks `Data Stack` und `Return Stack`. Sie dienen `MicroCore` zum Verwalten von Operanden bzw. Unterprogrammrückkehr–Adressen und Zwischenergebnissen. `TOS` und `NOS` sind dabei die obersten beiden Elemente des Datenstacks, `TOR` das oberste Element des Returnstacks, das auch als schneller Schleifenindex verwendet werden kann.

Zum Ausführen von Programmen, adressiert der Programmzähler `PC` den Programmspeicher (`Program Memory`). Die aktuell auszuführende Instruktion lädt `MicroCore` dabei zur Dekodierung in das Instruktions–Register `INST`. Sequentielle Programmausführung führt zum Inkrementieren des Programmzählers. Bei Unterprogramm–Aufrufen bzw. –Rücksprüngen wird der Programmzähler aus der `ALU` bzw. vom `Return Stack` geladen.

Vom Programmspeicher getrennt ist der Datenspeicher `Data Memory`. Zu verarbeitende Daten werden im Datenspeicher abgelegt und zur Bearbeitung in das Stack–System geladen, dort manipuliert und dann in den Datenspeicher zurückgeschrieben. Das `STATUS`–Register enthält dabei die Informationen über den Systemzustand (etwa Interrupt–Status, u. ä.), insbesondere auch über das Ergebniss der zuletzt ausgeführten Rechenoperation (Zero-Bit, Overflow, ...).

Über externe Signale kann die reguläre Programmausführung unterbrochen werden (`interrupt`). `MicroCore` erreicht dabei eine sehr kurze Latenzzeit, da Unterbrechungen unmittelbar nach der gerade ausgeführten Instruktion berücksichtigt werden können. Zum Konservieren des aktuellen Ausführungskontextes genügt es, das `STATUS`–Register auf dem Datenstack zu retten, alle anderen Kontextinformationen sind ohnehin stack–verwaltet. Über sog. `exception`–Signale können Peripherie–Komponenten ihre Bereitschaft zur Kommunikation signalisieren. Ist eine Komponente nicht bereit zu kommunizieren, aber `MicroCore` versucht, die Kommunikation vorzunehmen, dann bewirkt die `exception`–Leitung, dass ein spezieller Interrupt ausgelöst wird. Dieser leitet bei Anwesenheit des Echtzeit–Betriebssystems (Realtime–Kernel) einen Prozess–Wechsel ein oder aber führt sonst zu einem Auf–der–Stelle–Treten des Prozessors, bis die Peripherie–Komponente kommunikationsbereit ist. Externe Peripherie–Komponenten werden, wie weiter unten beschrieben, über die Leitungen `iodata` bzw. `ioaddr` angeschlossen.

Eine der besonderen Stärken der `MicroCore`–Architektur ist die Unabhängigkeit der Programmausführung von der Breite der zu verarbeitenden Daten. Spezielle `lit`–Instruktionen konstruieren Literale *schrittweise* im `TOS`–Register unabhängig von der Datenbreite, ähnlich, wie dies im Transputer [2] durch `PFIX` und `NFIX`–Instruktionen geschieht.

Für konkrete Ausprägungen der `MicroCore`–Architektur kann neben der Datenwortbreite, die Größe der Speicherbereiche und die Tiefe der Stacks über Parameter im VHDL–Code angepasst werden. Auf diese Weise sind `MicroCore`–Versionen mit 12, 16, 24 und 32 Bit Datenwortbreite bereits zum Einsatz gekommen. Die Parametrierung fließt ebenfalls in die Tool–Chain ein, um zugeschnittene Entwicklungswerkzeuge zu erhalten.

In der gegenwärtigen Version 1.x von `MicroCore` haben Instruktionen eine Breite von 8 Bits und `lit`–Instruktionen können damit jeweils 7 Bits eines Literals erzeugen.

1.2 Der `MicroCore`–Befehlssatz

`MicroCore`–Instruktionen sind immer genau 8 Bit breit. Ihr genereller Aufbau lässt sich durch die Tabelle in Abbildung 2 verdeutlichen. Bit 7 bestimmt, ob die Instruktion ein Operationscode (Bit 7 = 0) oder eine `lit`–Instruktion (Bit 7 = 1) ist. Handelt es sich um einen Operationscode, so werden vier Typen — `BRA`, `ALU`, `MEM` und `USR` — unterschieden:

7	6	5	4	3	2	1	0
$80	$40	$20	$10	$8	$4	$2	$1
Lit/Op	Type		Stack		Group		

Abbildung 2. Das `MicroCore`–Instruktionsformat

`BRA` Operationscodes des Typs `BRA` umfassen relative/absolute, bedingte/unbedingte Sprünge und Unterprogramm–Aufrufe und –Rücksprünge.

`ALU` Operationscodes des Typs `ALU` sind binäre und unäre arithmetisch/logische Operationen, die auf dem Datenstack (Register `TOS` und `NOS`) ausgeführt werden. Die `Stack`–Bits 4 und 3 bestimmen dabei, ob Elemente auf den Datenstack gelegt oder von ihm entfernt werden oder ob er in seiner Tiefe unverändert bleibt. Die `Group`–Bits 2 bis 0 bestimmen, welche arithmetisch/logische Operation auszuführen ist.

`MEM` Operationscodes des Typs `MEM` sind lesende und schreibende Zugriffe auf Datenspeicher und Returnstack.

`USR` Die 32 Operationscodes von Typ `USR` stehen für applikationsspezifische Erweiterungen zur Verfügung. Ihre Standardfunktionalität sind Unterprogrammaufrufe zu fest–definierten Adressen. Diese Standardfunktionalität kann durch strukturierte Änderungen des VHDL–Quellcodes abgeändert werden, so dass die `USR`–Operationscodes ein anderes Verhalten bekommen können.

2 Systementwurf mit `MicroCore`

Das Einsatzgebiet von `MicroCore` sind eingebettete Systeme, die im Stil eines System-on-a-Chip (SoC) im Hardware/Software–Co-Design entworfen werden. Der Entwurf zerfällt dabei in einen Hardware– und einen Software–Teil, die aufeinander abgestimmt entwickelt werden. Abbildung 4 zeigt das grundsätzliche Vorgehen.

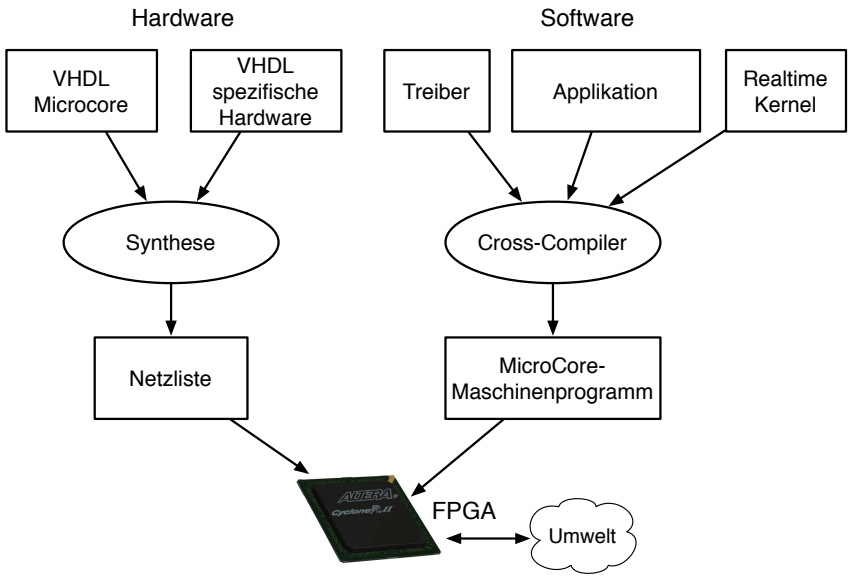


Abbildung 3. Sytementwurf mit `MicroCore`

Anders als in automatisierten Ansätze — etwa [3] oder [4] — wird hier die Hardware/Software–Partitionierung als ein intellektueller Ingenieur–Prozess angesehen, der weitreichende nichtfunktionale Anforderungen zu berücksichtigen hat. Der Ingenieur entscheidet aufgrund der (funktionalen und nichtfunktionalen) Anforderungen welche Teile des Systems durch Hardware (programmierte elektronische Schaltungen) und welche in Software (Programme für einen Mikroprozessor,

hier `MicroCore`) realisiert werden. Die Hardware–Teile werden in VHDL definiert, wobei nur die spezifische Hardware neu entworfen werden muss. `MicroCore` selbst wird lediglich passend für die Aufgabenstellung parametriert. Auf der Software–Seite entsteht das Applikationsprogramm und spezifische Treiber für die Projekt–Hardware, über die das Applikationsprogramm adäquat auf die Projekt–Hardware zugreifen kann. Ein auf `MicroCore` zugeschnittener Realtime–Kernel stellt typische Echtzeit–Betriebssystem–Funktionaliät zur Verfügung, die wiederholt in typischen Anwendungen benötigt wird.

Dieser Entwurfsprozess ist iterativ. Nachdem eine erste Version des Systems entworfen ist, wird analysiert, ob das entstandene System bereits alle Anforderungen erfüllt. Ist dies noch nicht der Fall, möglicherweise werden Echtzeitanforderungen noch nicht erfüllt, findet ein *Tuning* statt: Typischerweise werden Kernfunktionalitäten identifiziert, deren Berechnung beschleunigt werden muss, um die Echtzeitanforderungen zu erfüllen. Das heißt Teile, die bislang auf der Software–Seite realisiert waren, werden nun in Hardware realisiert und auf diese Weise beschleunigt. Zur Ermittlung der zu beschleunigenden Kernfunktionalitäten wird auf statische Programmanalysen, dynamische Messungen an konkreter Hardware und auf die Ingenieurs–Erfahrung zurückgegriffen.

Die Messungen an der konkreten Hardware werden durch ein interaktives Host/Target–Entwicklungssystem begünstigt. Das `MicroCore`–Zielsystem ist dabei mit einem Montor–Programm ausgestattet, das über eine *Nabelschnur* genannte serielle Verbindung mit dem Host–Computer verbunden ist. Dieser stellt Massenspeicher, Tastatur und Bildschirm zur Verfügung.

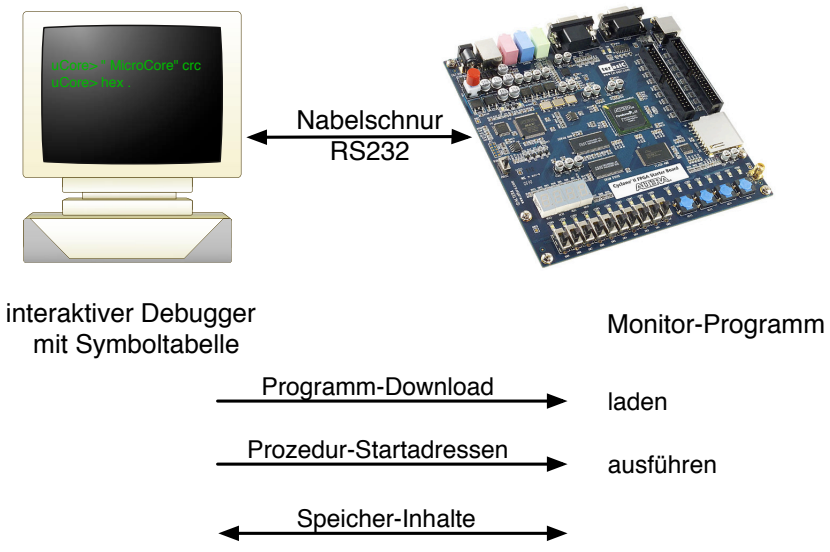


Abbildung 4. Programmentwicklung mit `MicroCore`

Über die Nabelschnur–Verbindung werden Programme in den `MicroCore`–Programmspeicher geladen und Unterprogramme durch Übertragen ihrer Anfangsadresse zum Target–System gestartet. Außerdem können Target–Datenspeicherinhalte inspiziert und modifiziert werden. Auf dieser Kommunikation aufbauend, ist ein interaktiver Debugger realisiert, der eine direkte Steuerung des Target–Systems erlaubt. Abbildung 4 zeigt dieses Vorgehen.

3 Anbindung spezifischer Hardware

Ein Teil der Systementwurfs–Aufgaben ist, die projekt–spezifische Hardware mit `MicroCore` zu verbinden. Das übliche Vorgehen dabei bindet die Hardware über die Leitungen `iodata` und `iodaddr` direkt an das Stack–System an. In `MicroCore`–Programmen kann so auf die Hardware mit den verfügbaren `MEM`–Operationscodes *memory–mapped* wie auf den Datenspeicher zugegriffen werden. Abbildung 5 zeigt dieses Vorgehen.

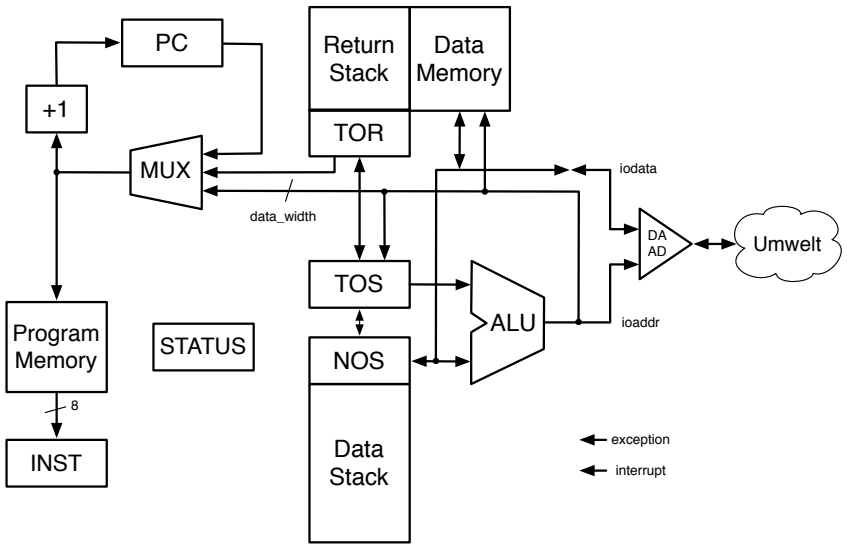


Abbildung 5. Anbindung spezifischer Hardware

Ein anderer Weg, Hardware mit `MicroCore` zu verbinden, ist die Realisierung von projekt–spezifischen `USR`–Instruktionen. Hierbei wird die Bedeutung einer oder mehrerer der 32 verfügbaren `USR`–Instruktionen so abgeändert, dass sie passende Effekte im Stack–System durchführen.

Die `USR`–Instruktionen können auch dazu verwendet werden, das oben angesprochene Tuning durchzuführen. Der ingenieurmäßige Prozess erfolgt dabei in drei Schritten, die in Abbildung 6 skizziert sind:

1. Zu optimierende Kernfunktionalitäten (innere Schleifen) werden identifiziert und zunächst in Software als Unterprogramm realisiert. Auch hier wird iterativ vorgegangen, bis der Zuschnitt der Kernfunktionalität adäquat für die anschließende Realisierung in Hardware ist. Das Anwendungsprogramm spricht die Kernfunktionalität durch einen Unterprogrammaufruf an.
2. Die selbe — bis jetzt durch ein Unterprogramm realisierte Funktionalität — wird nun in Software, durch eine `USR`–Instruktion mit Standardfunktionalität (Unterprogrammaufruf an eine festdefinierte, implizite Adresse) umgesetzt. Die dabei zum Einsatz kommende Software bleibt im wesentlichen unverändert, der Maschinencode für die Verwendung der Kernfunktion enthält nun statt eines Unterprogrammaufrufs die entsprechende `USR`–Instruktion.
3. Im dritten Schritt wird nun die `USR`–Instruktion so geändert, dass die Kernfunktionalität in Hardware realisiert wird. Die Implementierung ändert sich also,

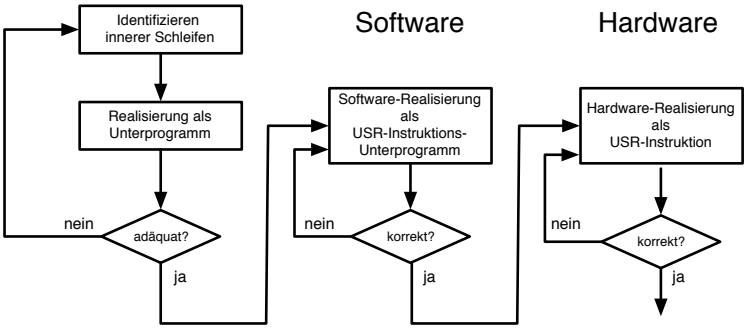


Abbildung 6. Prozess zum Tuning eines `MicroCore`–Systems

aber das Verhalten der `USR`–Instruktion im Stack–System und die Schnittstelle zum Programm bleiben gleich: Nach wir vor wird die Kernfunktionalität durch die gleiche `USR`–Instruktion angesprochen. Funktional findet keine Änderung statt. Die nichtfunktionalen Echtzeitanforderungen können erfüllt werden, wenn die in Hardware realisierte Kernfunktionalität schnell genug ist.

Zur Absicherung jeder dieser drei Schritte finden exakte Überlegungen bzw. Beweise und Tests statt, die sicher stellen, dass zum Ende jedes Schrittes die gewünschten Eigenschaften gelten. Erst dann wird der nächste Schritt gemacht.

4 Beispiel: Cyclic Redundancy Check

Um die bisherigen Ausführungen zum Hardware/Software–Co-Design mit `MicroCore` zu verdeutlichen, soll in diesem Abschnitt beispielhaft die Realisierung einer Funktion zur Berechnung von zyklischen Redundanzprüfung [5] vorgeführt werden. Zyklische Redundanzprüfungen (*cyclic redundancy check*, CRC) basieren auf Division von durch Bitfolgen repräsentierten Polynomen. Das dabei entstehende Rest–Polynom bei Division durch ein sog. *Generator–Polynom* ist der *CRC–Wert* . Treten bei der Datenübertragung einer Nachricht wenige Bitfehler auf, so ändert sich der CRC–Wert der Nachricht. Eine Überprüfung des CRC–Werts kann also (einfache) Übertragungsfehler aufdecken. Näheres über zyklische Codes und zyklische Redundanzprüfung findet sich in der einschlägigen Literatur, etwa in Kapitel 6 von [6].

```
11010110110000
10011                    1x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0
-----
 10011
 10011
 -----
  000010110
      10011
      -----
       010100
        10011
        -----
          1110  (Rest)
```

Abbildung 7. Berechnung eines CRC (Quelle: wikipedia)

Abbildung 7 zeigt, wie eine entsprechende Polynomdivision stattfindet: Für die Nachricht 11010110110000 wird mit dem Generatorpolynom 10011 ($1x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0$) der CRC–Wert 1110 bestimmt.

4.1 Software–Realisierung

Hier soll der CRC-16 nach CCITT mit dem Generatorpolynom $x^{16} + x^{12} + x^5 + 1$ betrachtet werden.

In der Praxis wird des CRC–Wert nicht bit–weise berechnet sondern byte–weise: Die Nachricht wird in 8 Bit breite Stücke zerteilt die jeweils in einem Schritt verarbeitet werden. In Abbildung 8 ist die Implementierung eines solchen CRC–Schritts und die Iteration über eine gesamte Nachricht in C und in Forth dargestellt. Eingangsparameter der Kernfunktion `crc-ccitt-step` sind der bisher berechnete CRC–Wert und das neu zu verarbeitende Byte; Ausgangsparameter/Funktionswert ist der aktualisierte CRC–Wert. Für CRC–16 wird die Berechnung mit einem CRC–Wert von hexadezimal `FFFF` begonnen.

```
unsigned int  crc_ccitt_step(unsigned char b
                             unsigned int crc)

  crc = (unsigned char)(crc >> 8) | (crc << 8);
  crc ^= b;
  crc ^= (unsigned char)(crc & 0xff) >> 4;
  crc ^= (crc << 12);
  crc ^= ((crc & 0xff) << 4) << 1;
  return crc

unsigned int crc_ccitt(unsigned char* data
                       int len)
  unsigned int crc = 0xFFFF;
  for (int i=0; i<len; i++)
     crc = crc_ccitt_step(data[i], crc);

  return crc;
```

```
: lshift ( x n -- x' )  1- times 2* ;
: rshift ( x n -- x' )  1- times u2/ ;

: crc-ccitt-step ( crc b -- crc' )
  swap dup 8 lshift
  swap 8 rshift or xor
  dup $FF and 4 rshift xor
  dup 12 lshift xor
  dup $FF and 5 lshift xor
  $FFFF and
;

: crc-ccitt ( addr len -- crc )
  $FFFF rot rot
  bounds ?DO  I @ crc-ccitt-step
              LOOP
;
```

Abbildung 8. Berechung des CRC–16 in C und Forth

Aus dem Anwendungsprogramm wird diese Funktionalität durch Unterprogrammaufrufe angesprochen und auch die Funktion `crc-ccitt` ruft `crc-ccitt-step` als Unterprogramm auf, wie in Abbildung 9 zu sehen ist. Die `lit`–Instruktionen vor

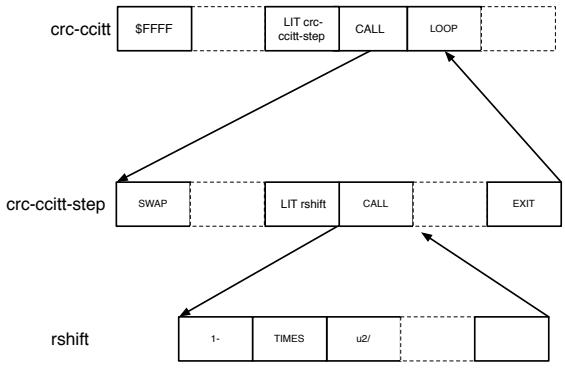


Abbildung 9. Aufrufstruktur bei der Software–Realisierung als Unterprogramm

dem `CALL` in der ersten Zeile konstruieren die Anfangsadresse des `crc-ccitt-step`–Unterprogramms.

Das `CALL` führt dann den passenden Unterprogrammaufruf durch. `crc-ccitt-step` selbst verwendet die Funktion `rshift`, um die nötigen Schiebe–Operationen durchzuführen.

4.2 Software–Realisierung als `USR`–Instruktion

Nach ausgiebiger Prüfung, dass die so gewählte Kernfunktionalität angemessen ist, wird sie nun im zweiten Schritt in Software als `USR`–Instruktion 6 realisiert. Abbildung 10 zeigt die Realisierung in Forth für den `MicroCore`–Crosscompiler und die resultierende Aufrufstruktur.

```
6 USR: crc-step  ['] crc-ccitt-step nop branch ;USR

: crc-ccitt ( addr len -- crc )
  $FFFF rot rot
  bounds ?DO  I @ crc-step
              LOOP
;
```

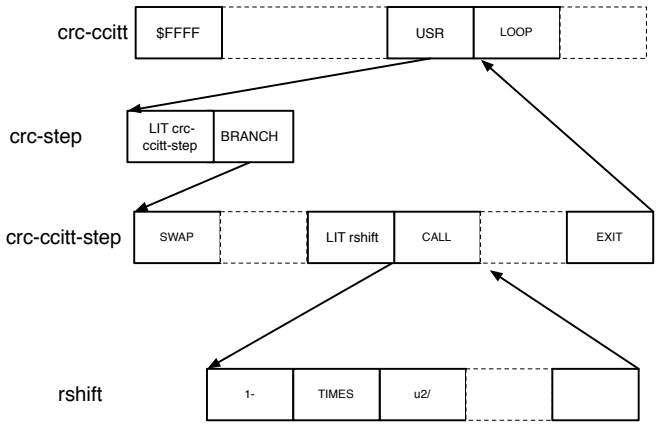


Abbildung 10. Aufrufstruktur bei der Software–Realisierung als `USR`–Instruktion

In `crc-ccitt` wird nun nicht mehr ein Unterprogrammaufruf kompiliert, sondern eine `USR`–Instruktion. Diese führt ja in ihrer Standardbedeutung einen Unterprogrammaufruf zu einer festen Adresse aus. Dort steht ein Sprung an den Anfang des unveränderten `crc-ccitt-step`. Die Funktionalität von `crc-ccitt` bleibt gleich, was durch geeignete Überlegungen und Tests sichergestellt wird.

4.3 Hardware–Realisierung als `USR`–Instruktion

Die Implementierung der `USR`–Instruktion 6 wird nun bedeutungserhaltend in VHDL vorgenommen. Abbildung 12 (im Anhang) zeigt den entsprechenden VHDL–Ausschnitt. Hier findet eine ganz andere Massage der Bits statt als in der C bzw. Forth–Implementierung und der Ingenieur muss sich davon überzeugen, dass diese äquivalent zur Realisierung in Software ist.

Die `USR`–Instruktion 6 soll nun keinen Unterprogrammaufruf an eine feste Adresse mehr vornehmen, sondern das Stack–System mit Hilfe der gerade definierten

VHDL–Funktion manipulieren. Hierzu sind Änderungen in `MicroCore` nötig: Als Datenstack–Effekt muss ein Element vom Stack entfernt werden; der Returnstack muss unverändert bleiben und die im Sequencer dekodierte `USR`–Instruktion 6 muss die VHDL–Funktion `crc_ccitt_step` verwenden. Die Änderungen im `MicroCore`–Sequencer zeigt Abbildung 11.

```
    CASE  i_type  IS
...
  WHEN OTHERS => -- op_USR
CASE i_USR IS
                    WHEN "00110" =>  -- USR 6
                       pop_stack;
                       tos_new <= crc_ccitt_step(tos_reg, nos_reg);
                    WHEN OTHERS =>
                      IF  inst=INT_OP  THEN
                         push_stack(tos_reg);
                         tos_new <= status;
                      END IF;
                  END CASE;
   END CASE;
```

Abbildung 11. Änderungen im `MicroCore`–Sequenzer

Auf der Ebene des Anwendungsprogramms aber ändert sich nichts: Die Funktion `crc-ccitt` enthält nach wie vor die `USR`–Instruktion 6 (wie in der obersten Zeile der Grafik in Abbildung 10) nur das diese eben nun keinen Unterprogrammaufruf mehr vornimmt sondern die gewünschte Funktionalität direkt in Hardware realisiert. Die Funktionalität von `crc-ccitt-step` ist unverändert. Natürlich gilt es zu überprüfen, das die nichtfunktionalen Anforderungen nun erfüllt werden.

5 Zusammenfassung und Ausblick

Bei der Entwicklung von System-on-a-Chip–Lösungen mit `MicroCore` findet die Partitionierung von Aufgaben in Software– und Hardware–Bestandteile nicht automatisch sondern in einem iterativen Engineering–Prozess statt. Dieser unterstützt durch ein strukturiertes und abgesichertes Vorgehen das Tuning, bei dem Kernfunktionalität identifiziert und statt in Software in Hardware realisiert wird, um nichtfunktionale Anforderungen — etwa Echtzeitanforderungen — erfüllen zu können.

Das gezielte Realisieren von Kernfunktionalität in Hardware hat eine Reihe von Anwendungsgebieten. Als Beispiele seien hier schnelle spezifische Multiplikationen, die Berechnung von Hash–Werten, kryptographische Algorithmen oder schnelle Fourier–Transformationen genannt.

`MicroCore`–Spezialoperationen für digitale Filter wurden bereits industriell eingesetzt.

Literatur

1. Klaus Schleisiek: Micro Core an open-source, scalable VHDL synthesisable dual stack Harvard processor for FPGAs. EuroForth Conference, Schloss Dagstuhl, (2001)
2. Iann M. Barron, The Transputer. in D. Aspinall. ed. The Microprocessor and its Application: an Advanced Course. p.343. Cambridge University Press (1978).
3. T. Callahan, R. Hauser, J. Wawrzynek: The GARP Architecture and C Compiler. IEEE Computer 33(4), 62-69, April 2000
4. M.B. Gokhale, J.M. Stone: NAPA C: Compiling for a Hybrid RISC/FPGA–Architecture. Proc. IEEE Symp. on FPGAs for Custom Computing Machines (1998)
5. Wikipedia: Zyklische Redundanzprüfung URL: `http://de.wikipedia.org/wiki/Cyclic_Redundancy_Check` letzter Abruf: 2009-07-10
6. Wolfgang Willems: Codierungstheorie. de Gruyter (1999).

6 Anhang

```
function crc_ccitt_step
   (d: std_logic_vector(7 downto 0);
    crc:  std_logic_vector(15 downto 0))
   return std_logic_vector is

   variable result: std_logic_vector(15 downto 0);

 begin
   result(0) := d(4) xor d(0) xor crc(8) xor crc(12);
   result(1) := d(5) xor d(1) xor crc(9) xor crc(13);
   result(2) := d(6) xor d(2) xor crc(10) xor crc(14);
   result(3) := d(7) xor d(3) xor crc(11) xor crc(15);
   result(4) := d(4) xor crc(12);
   result(5) := d(5) xor d(4) xor d(0) xor crc(8) xor crc(12) xor crc(13);
   result(6) := d(6) xor d(5) xor d(1) xor crc(9) xor crc(13) xor crc(14);
   result(7) := d(7) xor d(6) xor d(2) xor crc(10) xor crc(14) xor crc(15);
   result(8) := d(7) xor d(3) xor crc(0) xor crc(11) xor crc(15);
   result(9) := d(4) xor crc(1) xor crc(12);
   result(10) := d(5) xor crc(2) xor crc(13);
   result(11) := d(6) xor crc(3) xor crc(14);
   result(12) := d(7) xor d(4) xor d(0) xor crc(4) xor crc(8) xor crc(12) xor crc(15);
   result(13) := d(5) xor d(1) xor crc(5) xor crc(9) xor crc(13);
   result(14) := d(6) xor d(2) xor crc(6) xor crc(10) xor crc(14);
   result(15) := d(7) xor d(3) xor crc(7) xor crc(11) xor crc(15);

   return result;
 end crc_ccitt_step;
```

Abbildung 12. Die Realisierung des CRC–Schritts in VHDL

Zur Programmierung raumzeitlicher diskreter Systeme

Hermann von Issendorff

Institut für Netzwerkprogrammierung
21745 Hemmoor
hviss@issendorff.de

Zusammenfassung: Abstrahiert man ein raumzeitliches diskretes System von Funktionalität und Metrik, dann reduziert es sich auf ein dreidimensionales Knotennetz. Wir zeigen in diesem Beitrag, dass ein solches Knotennetz vollständig, d.h. bijektiv, auf ein orientiertes planares Netz oder eine orientierte eindimensionale Symbolkette, d.h. ein Programm, abgebildet werden kann. Die Programmiersprache ist eine mehrsortige Termalgebra und wird Akton-Algebra genannt. Bereits in der abstrakten Form lassen sich Strukturen wie die Doppelhelix der DNA, α-Helices und β-Sheets als Programm modellieren. Durch Zuweisung von Eigenschaften an die Knoten, z.B. durch digitale oder analoge Funktionen oder räumliche Metrik, kann Akton-Algebra konkretisiert werden.

1 Einführung

Die belebte Natur macht uns die Programmierung raumzeitlicher diskreter Systeme vor: Sie erzeugt Ketten von Aminosäuren durch sequentielles Übersetzen von Gen-Code. Die fertige Aminosäurenkette faltet sich in einer geeigneten Umgebung zu einem Protein zusammen, d.h. ein eindimensionales Gebilde wird zu einem dreidimensionalen. Der Vorgang kann durch Änderung der Umgebung umgekehrt werden. Zwischen Aminosäurenkette und Protein besteht damit eine bijektive und bikontinuierliche Abbildung, ein Homöomorphismus. Mit anderen Worten: Die Aminosäurenkette stellt ein Programm dar, in dem die Information über die dreidimensionale Struktur des Proteins vollständig enthalten ist.

Die räumliche Struktur eines Proteins entsteht chemisch/physikalisch dadurch, dass die Aminosäuren neben der Kettenbindung untereinander zusätzliche schwächere Bindungen eingehen. Durch äusseren Einfluss, z.B. durch ein sich anlagerndes weiteres Molekül, kann das Protein in eine andere metastabile homöomorphe Struktur übergehen. Mit anderen Worten: Ein Protein kann induziert verschiedene Zustände annehmen und ist damit zur Informationsspeicherung und -verarbeitung in der Lage.

Der Vorgehensweise der Natur folgend, wird in dieser Arbeit eine mehrsortige Termalgebra vorgestellt, die die Sprache der belebten Natur enthält, aber auch konkrete metrische Strukturen bzw. darauf ablaufende Datenverarbeitung darzustellen erlaubt.

Ein diskretes physikalisches System besteht aus einer Menge materieller Komponenten, die aktiv oder statisch sein können. Sind die Komponenten aktiv, d.h. produzieren sie physikalische Objekte oder werten sie Funktionen aus, dann sind sie zeitlich gerichtet und werden in einer partiellen zeitlichen Ordnung aktiviert. Sind sie statisch,

dann kann man ihnen eine partielle Aufbauordnung zuweisen, die ebenfalls eine zeitliche Richtung induziert.

Abstrahiert man ein diskretes physikalisches System, beispielsweise einen Digitalrechner, von seiner Metrik, d.h. von den räumlichen Abmessungen der Komponenten, dann schrumpft das System zu einem dreidimensionalen gerichteten Netz von ausführbaren Aktivitäten, die in den Komponenten realisiert sind. Abstrahiert man ein diskretes physikalisches System, beispielsweise wieder einen Digitalrechner, von seiner Funktionalität, dann bleibt ein dreidimensionales Netz von Bausteinen übrig, die die Abmessungen der Komponenten haben. Abstrahiert man zugleich von der Metrik und der Funktionalität, dann bleibt ein dreidimensionales Netz von Knoten übrig, das nur noch die räumliche Anordnung und die Abhängigkeiten der Knoten zeigt. Zwei beliebige Knoten dieses Netzes können abhängig sein oder nicht, und jeder abhängige Knoten kann mit einer endlichen Menge von Vorgänger- bzw. Nachfolgerknoten verbunden sein.

Gibt man den Netzknoten ihre ursprünglichen funktionalen und metrischen Eigenschaften zurück, dann erhält man wieder das ursprüngliche System. Das gerichtete Knotennetz erweist sich daher als gemeinsame strukturelle Grundlage aller diskreten physikalischen Systeme. Wenn dazu - wie im Fall von Protein und Aminosäurenkette - das gerichtete Knotennetz bijektiv und bikontinuierlich auf eine Symbolkette abgebildet werden kann, dann definiert diese Abbildung eine Programmiersprache, die eine gemeinsame Grundsprache aller diskreten physikalischen Systeme ist. Akton-Algebra, abgekürzt *AA*, ist eine solche Programmiersprache.

Zur Vermeidung von Missverständnissen sei darauf hingewiesen, dass die Abstraktion von Metrik und Funktionalität nicht etwa Abstraktion von Raum und Zeit bedeutet. Letztere würde ein reales diskretes System auf einen gerichteten Graphen reduzieren, d.h. auf eine raumzeitlose mathematische Struktur. Die Raum- und Zeit-Beziehungen zwischen Netzknoten sind aber genau die physikalischen Eigenschaften, auf der die *AA* aufbaut. Ein Netzknoten, d.h. die metrische und funktionale Abstraktion einer Komponente, hat eine endliche räumliche Ausdehnung, und die Durchquerung eines Netzknotens braucht eine endliche Zeit.

Die raumzeitlichen Beziehungen zwischen Programmiersprachen und diskreten physikalischen Systemen haben augenscheinlich bisher weder in der Informatik noch in der Physik Beachtung gefunden. Bisherige Programmiersprachen haben keine explizite Raumzeitsemantik und sind zudem allgemein auf Datenverarbeitung fixiert. Selbst die dem Hardware-Entwurf dienenden HDL-Sprachen [Jansen 03] sind auf die Spezifikation von Funktionen und eine grobe Modellierung des Zeitverhaltens beschränkt, d.h. haben keinerlei Bezug zur räumlichen Struktur. Sie alle enden an der Barriere der Register-Transfer-Ebene, die die Hardware-Strukturen versteckt. Auch auf den Gebieten der Chemie oder der Molekularbiologie [Alberts et al. 02] lassen sich keine Ansätze zu strukturbeschreibenden Programmiersprachen finden.

In dieser Arbeit kann aus Platzgründen nur ein Überblick über den Aufbau und einige der vielen Möglichkeiten der Anwendung der *AA* gegeben werden. Wir beschränken uns daher im nächsten Abschnitt auf die Einführung der Strukturelemente, die für die abstrakte *AA* erforderlich sind. Im dritten Abschnitt werden sodann funktionale und metrische Konkretisierungen behandelt.

2 Elemente und elementare Strukturen der *AA*

Ein metrikfreies gerichtetes Knotennetz kann nur in einem relationalen Bezugssystem beschrieben werden. Als solches bietet sich ein Beobachter an, der zwischen *links-rechts*, *oben-unten* und *vorne-hinten* unterscheidet. Zur einheitlichen Orientierung der Knotennetze ist es weiterhin erforderlich, unter diesen Relationspaaren eine privilegierte Raumrichtung auszuwählen. Hierfür wird die Richtung von *links* nach *rechts* gewählt. Da, wie bereits erwähnt, jede Aktivität Zeit braucht, ist die privilegierte Raumrichtung zugleich die Richtung der Zeit, d.h. *links* kann als *früher* und *rechts* als *später* interpretiert werden.

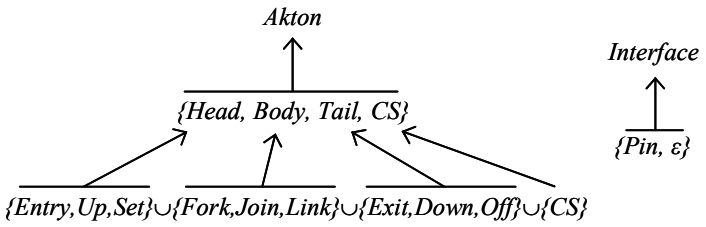


Abbildung 1: Sortenhierarchie der Basisknoten eines abstrakten gerichteten Knotennetzes (links) und die Hierarchie ihrer Schnittstellen (rechts).

Die Knoten eines gerichteten Knotennetzes lassen sich immer in Basisknoten zerlegen. Diese sind in einer Sortenhierarchie darstellbar, in der mit jeder Ebene weitere Sorteneigenschaften eingeführt werden. Die ersten drei Ebenen sind in Abbildung 1 dargestellt. An der Spitze der Hierarchie der Basisknoten steht die Allsorte *Akton*, die einen Input und einen Output als Interface hat. Auf der zweiten Sortenebene wird zwischen 4 Sorten unterschieden, dem Anfangsknoten *Head*, dem inneren Knoten *Body*, dem Endknoten *Tail* und dem geschlossenen Knoten *CS* (*Closed System*). *Head* hat einen leeren Input und nichtleeren Output, *Tail* genau das Umgekehrte. Die Inputs und Outputs von *Body* sind nichtleer und die von *CS* sind leer. Auf der dritten Sortenebene werden Verbindungsstrukturen spezifiziert. Zur Sorte *Body* gibt es die Knotensorten *Fork*, *Link* und *Join*, zur Sorte *Head* die Knotensorten *Entry*, *Up* und *Set*, und zur Sorte *Tail* die Knotensorten *Exit*, *Down* und *Off*. *Fork*, *Link* und *Join* unterscheiden sich durch die Zahl der *Pin* genannten Anschlüsse zu anderen Knoten. *Fork* hat ein *Pin* im Input und zwei im Output, *Join* genau das Umgekehrte. *Link* hat beidseitig ein *Pin*. Die Teilsorten zu *Head* haben nur ein *Pin* im Ausgang, und *Tail* nur einen im Eingang. Das *Entry* in *Head* und das *Exit* in *Tail* beschreiben die Anschlüsse des Knotennetzes zur Umgebung. Die weiteren Teilsorten sind zur Sequenzialisierung des Knotennetzes erforderlich. *Up* und *Down* sind ein syntaktisch gekoppeltes Paar zur Beseitigung von Kreuzungen, und ebenso *Set* und *Off* zur Beseitigung von Zyklen und Querverbindungen.

Die Schnittstellen zwischen den Knoten werden in einer separaten Sortenhierarchie beschrieben. Hier ist *Interface* die Allsorte, die in der nächsten Ebene *Pin* und ε (leer) als abstrakte Teilsorten hat. Beide Teilsorten können zu verschiedenen Klassen von Teilsorten konkretisiert werden. Bei Funktionalisierung können der Sorte *Pin* z.B.

digitale oder analoge Variable als Teilsorten zugeordnet werden. Bei Metrisierung kann der Sorte ε beispielsweise die konkrete Teilsorte *Gap* zugeordnet werden.

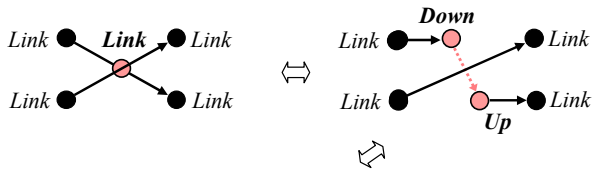


Abbildung 2: Planarisierung räumlicher Strukturen durch Aufschneiden der Kreuzungen und Kennzeichnung der Schnittenden mit Down/Up-Paaren.

Teilnetze eines Knotennetzes werden algebraisch als Terme bezeichnet. Zwei unmittelbar benachbarte Aktonterme können voneinander abhängig oder unabhängig sein. Liegt von zwei benachbarten Termen ein Term *x links* und damit zeitlich *vor* Term *y* und ist die nichtleere Ausgangsschnittstelle von *x* gleich der Eingangsschnittstelle von *y*, dann wird das durch den *Next* genannten Infix-Operator '>' beschrieben. Gemeinsam bilden sie den Term *(x>y)*. Sind zwei benachbarte Terme *x* und *y* voneinander unabhängig und liegt *x* oberhalb von *y*, dann wird das durch den *Juxta* genannten Infix-Operator '/' beschrieben. Gemeinsam bilden sie den Term *(x/y)*. Aktonterme beliebiger Grösse können damit wie ein einzelnes Akton behandelt werden. Zudem verfügt *AA* über die wichtige Eigenschaft der Komposition.

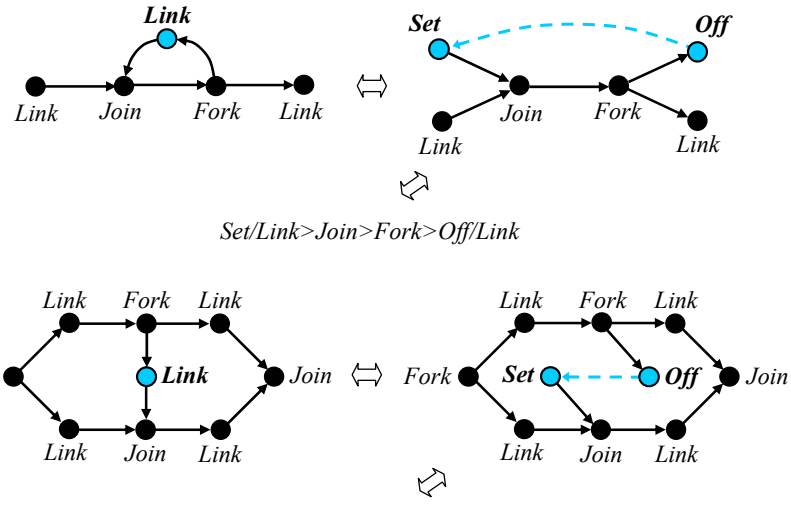


Abbildung 3: Linearisierung planarer Strukturen durch Aufschneiden von Zyklen und Querverbindungen und Kennzeichnung der Schnittenden mit *Set/Off*-Paaren.

Mit diesem Rüstzeug ist es jetzt möglich, jedes abstrakte gerichtete Knotennetz homöomorph in eine orientierte Knotenkette umzuformen. Das geschieht in zwei Schritten. Im ersten Schritt wird das räumliche Knotennetz planarisiert. Dabei auftretende Kreuzungen werden durch Aufschneiden der hinteren Verbindung und Einfügung eines aus *Down* und *Up* bestehenden Knotenpaares eliminiert. Die Überführung einer Kreuzung in einen homöomorphen Aktonterm wird in Abbildung 2 gezeigt. Ein planares Knotennetz kann noch Zyklen oder Querverbindungen enthalten, die der Linearisierung entgegenstehen. Beide Strukturen lassen sich durch Aufschneiden und Einfügung eines zweiten, *Off* und *Set* genannten Knotenpaares eliminieren. Die Überführung von Zyklen und Querverbindungen in homöomorphe Aktonterme wird in Abbildung 3 gezeigt.

Die Basisknoten der untersten Hierarchieebene in Abbildung 1 sind die Basisterme und damit das Vokabular der *AA*-Programmiersprache. Jeder Term, der sich durch Verknüpfung von Basistermen bilden und auf die Sortenhierarchie zurückführen lässt, ist gültig. Variable der Sortenmengen der dritten Ebene werden mit einer Ausnahme mit dem ersten Buchstaben des zugehörigen Basisterms bezeichnet, also *E(ntry), U(p), S(et), F(ork), J(oin), L(ink), (E)X(it), D(own), O(ff)*. Die Sortenmengen zu *Fork* und *Join* haben die Besonderheit, dass nur die Basisterme planar sind, multiple *F*- und *J*-Terme dagegen immer Kreuzungen enthalten. Wo erforderlich, werden multiple Terme in der Form *Tm*, $m \in \mathbf{N}$ beschrieben.

Zur vereinfachten Schreibung von *AA*-Termen werden noch zwei Zählkonventionen eingeführt. Eine *Juxta*-Kette von *x*-Termen der Länge *n,* $n \in \mathbf{N}$*,* wird durch *x^n* beschrieben und eine *Next*-Kette entsprechend durch *n*x*.

2.1 Anwendungen der abstrakten *AA*

Um einen Eindruck von der Bandbreite darstellbarer Strukturen zu geben, werden jetzt verschiedene Beispiele behandelt. Als erstes betrachten wir einen Tetraeder, d.h. die einfachste räumliche Struktur, die zudem ungerichtet ist und weder einen Eingang noch einen Ausgang hat (Abbildung 4).

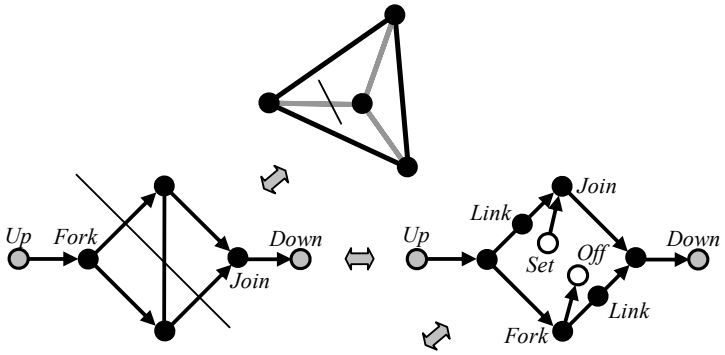


Abbildung 4: *AA*-Darstellung eines Tetraeders in räumlicher, planarer und linearer Form.

Der Tetraeder hat aus der Sicht des Beobachters mindestens eine hintenliegende Kante. In einem ersten Schritt wird diese aufgeschnitten, die Schnittenden mit *Up* und *Down* versehen und *Up* nach links und *Down* nach rechts gestreckt. Die durch *Up* und *Down* eingeführte Richtung wird soweit möglich auf die dazwischen liegenden Kanten übertragen. Die resultierende planare Struktur ist damit bis auf eine innere Kante gerichtet und im Raum orientiert. Diese Kante kann einer von zwei Richtungen zugeordnet werden. Durch einen *Set/Off*-Schnitt, der auch das Einfügen von je einem *Link* in die äusseren Kanten erfordert, kann sie sodann in eine linearisierbare Form gebracht und als *AA*-Term beschrieben werden.

Die nächsten beiden Beispiele beinhalten die Modellierung einer α-Helix und eines ß-Sheet, d.h. von Strukturen, die bei der Faltung einer Aminosäurenkette in ein Protein als Zwischenstrukturen auftreten [Alberts et al. 02]. In Abbildung 5 ist eine aus zwei Schleifen bestehende rechtsdrehende α-Helix und ein durch zweifache Faltung der Aminosäurenkette gebildetes ß-Sheet gezeigt. Die Aminosäuren sind einheitlich als *B*-Terme beschrieben. Die den Strukturen zugeordneten *AA*-Terme können als Aminosäurenketten interpretiert werden. Wichtig ist, dass die individuelle Zuordnung der *U/O*- wie auch der *S/O*-Paare syntaktisch gewährleistet ist.

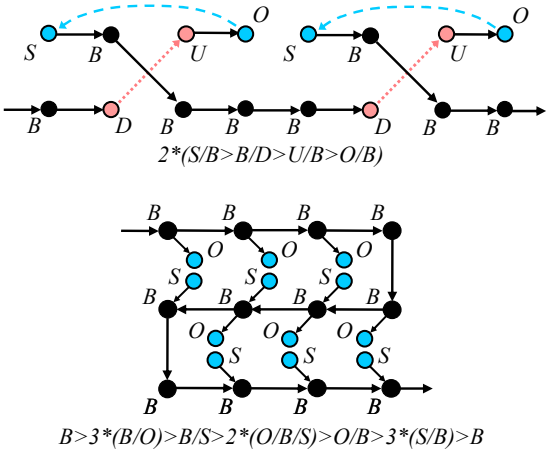


Abbildung 5: Modellierung von α-Helix (oben) und ß-Sheet (unten), der wesentlichen Teilstrukturen von Proteinen.

3 Konkretisierung der *AA*

Die abstrakte Knotennetze beschreibende AA kann nun auf vielfältige Weise konkretisiert werden. Die Konkretisierung lässt sich dadurch erreichen, dass den Sorten der dritten Ebene der Akton-Hierarchie weitere Teilsorten hinzugefügt werden, deren Benennung den allgemeinen Begriff "Knoten" auf die Semantik spezieller Systeme einengt. Das Ergebnis sind Programmiersprachen, die beschreibenden Charakter haben. Führt man zudem weitere Teilsorten in der Interface-Hierarchie ein, dann ergeben sich operationale Programmiersprachen, die je nach Definition Funktionen auswerten oder metrische Objekte konstruieren.

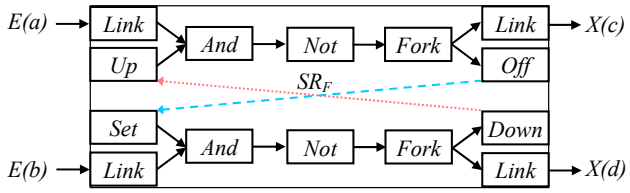
Wir demonstrieren in den nächsten beiden Unterabschnitten zuerst die Funktionalisierung und dann die Metrisierung eines digitalen Systems, d.h. eines SR-Flipflops.

3.1 Funktionalisierung am Beispiel eines SR-Flipflops

Wie allgemein bekannt, lässt sich jede digitale Funktion allein durch *Nand*- oder *Nor*-Elemente realisieren. Üblicherweise kommen aber auch andere Elemente zur Anwendung. Hier führen wir die Basisknoten *And*, *Or*, *Not* und *Wire* ein, wobei *Not* die inverse Funktion und *Wire* die 1-Funktion bedeutet. *And* und *Or* sind Teilsorten der Sorte *Join*, *Not* und *Wire* sind Teilsorten der Sorte *Link*, d.h.

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

Unter Verwendung dieser Aktonsorten lässt sich die Struktur beliebiger digitaler Systeme beschreiben. Als Beispiel verwenden wir ein *SR*-Flipflop (Abbildung 6).



SR_F = *(Link/Up>And>Not>Fork>Link/Off)/*
(Set/Link>And>Not>Fork>Down/Link)

Abbildung 6: Planare Darstellung der Schaltung eines *SR*-Flipflops. Darunter die Darstellung als Programm.

Durch die zusätzliche Erweiterung der *Interface*-Sorte *Pin* mit den Teilsorten *0,1,#*, wobei "#" die Bedeutung "*unbestimmt*" hat, d.h.

$$Pin:=\{0,1,\#\}$$

und der Definition aller Endsorten, insbesondere von *And, Or* und *Not*, als Funktionen wird *AA* zu einer operationalen digitalen Programmiersprache.

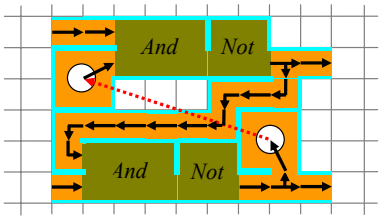
3.2 Metrisierung am Beispiel eines SR-Flipflops

Durch Konkretisierung der Knoten mit Metrik wird das Knotennetz zu einem Komponentennetz. Die Komponenten können die verschiedensten Formen und Grössen haben und sogar verformbar sein. Die Metrik kann damit beliebig kompliziert werden. Wir wählen hier als einfachste Metrik ein kartesisches Koordinatensystem mit einem einheitlichen Raster. Zu diesem Zweck wird die Leersorte ε in der *Interface*-Hierarchie durch die Sorte *Gap* konkretisiert, d.h.

$$\varepsilon:=\{Gap\}.$$

Zusätzlich wird festgelegt, dass *Pin* und *Gap* die gleiche Rasterbreite haben. Der Einfachheit halber beschränken wir uns auf eine planare Darstellung. Unter diesen Annahmen wird aus dem Knotennetz ein rechtwinkliges Komponentennetz, das nach wie vor von links nach rechts orientiert ist. Das Ziel ist jedoch im Allgemeinen die möglichst kompakte Faltung des Netzes in eine vorgegebene Fläche zur Vermeidung unnötig langer Verbindungen bzw. grosser Signallaufzeiten. Die Faltung wird dadurch möglich, dass der Beobachter die Verbindungswege durchwandert und dabei das an ihn gekoppelte relative Bezugssystem mitnimmt. Zu diesem Zweck werden Teilsorten zu *Fork* und *Wire* eingeführt, die die Wegrichtung ändern oder beibehalten. Die relative Lage ihrer Ausgänge zum Eingang wird durch die Richtungen *left*, *right* und *straight* gekennzeichnet und als Abkürzung *l,r,s* an die Bezeichner *F* und *L* angefügt:

$$Fork:=\{F_{lr}, Fl_{ls}, F_{sr}\} \text{ und } Wire:=\{W_s, W_l, W_r\}$$



SR_M = *(2*W_s/V_u>And>Not>F_{sr}>W_s)/*
*(W_r>W_s>W_l>W_r>4*W_s>2*W_l)/*
(W_s>And>Not>W_s>F_{ls}>V_d/W_s)

Abbildung 7: Planares Layout und metrische Programmbeschreibung eines *SR*-Flipflops.

Abbildung 7 zeigt ein mögliches Layout als Bild und in Programmform. Die Komponente *V* stellt ein *Via* dar, deren relative Grösse der Realität angepasst ist. Via_u ist Teilsorte von *Up* und Via_d von *Down*, d.h. $Up:=\{ Via_u \}$ und $Down:=\{ Via_d \}$.

Literaturverzeichnis

[Alberts et al. 02] Alberts, B.; Johnson, A., Lewis, J., Raff, M., Roberts, K., Walter, P.: "*Molecular Biology of the Cell*", New York: Garland Publishing (2002).

[Jansen 03] Jansen, D. et al.: "The Electronic Design Automation Handbook"; Kluwer Academic Publishers (2003).

Recency Types for JavaScript (Extended Abstract)

Phillip Heidegger Peter Thiemann
Universität Freiburg, Germany
{heidegger,thiemann}@informatik.uni-freiburg.de

April 11, 2009

Scripting languages are popular because their dynamic type systems accelerate the development of short programs. For larger programs, static analysis becomes an important tool for detecting programming errors.

We specify such an analysis as a type system for a call-by-value lambda calculus with objects and prove its type soundness. The calculus models essential features of JavaScript, a widely used dynamically-typed language: objects as property maps, type change during object initialization, and prototypes.

In JavaScript, the type of a property can change with every assignment. As long as a property is not yet assigned, accessing it returns the value `undefined`, which is a regular JavaScript value. Hence, in a flow-insensitive type system, each property type would have to contain the type `undefined` as a subtype (e.g., as component of a union type). As it is a run-time error to dereference `undefined` or to call it as a function, the type system must reject any attempts to use the property in this way. Although JavaScript converts `undefined` to a string, a number, or a boolean as needed, this conversion may not be intended by the programmer. Hence, the imprecise type gives rise to too many false positives.

We tackle this problem in a novel way with *recency types*. Recency types arise in a flow-sensitive type system, which distinguishes between the most recently created object at some program point and all objects previously created at the same program point. For each object creation site, it assigns one type to the most recent object and another type to all remaining objects. The crucial observation is that the type of the most recent object describes exactly one object. Hence, any update to the value of this object can be precisely reflected as a strong update in the type of the object.

Recency types fit the typical initialization pattern in imperative languages where the programmer allocates a number of objects and then initializes these objects. Quite often, no further properties are defined after the initialization and the type of the properties rarely changes. During the initialization phase, objects are most recent and their types can be updated in a flow sensitive manner. Later on, when the shape of the object does not change anymore, it falls back to flow insensitive typing.

The idea of a recency abstraction can be fruitfully transported to a type system. This type system is amenable to analyzing programs in scripting languages, in particular handling object initialization. Its distinguishing feature is the ability to handle type changes and prototypes precisely.

We designed and implemented a type inference algorithm and exercised it with example programs. We are extending the implementation with the goal to apply it to realistic programs.

Ein graphischer Debugger für Haskells Software Transactional Memory

Fabian Reck

Institut für Informatik

Christian-Albrechts-Universität zu Kiel

fre@informatik.uni-kiel.de

Nebenläufige Programmierung ist ein wichtiges Mittel um verschiedene Probleme in komplexen Anwendungen zu lösen. Durch den expliziten Einsatz von *Locks* zur Gewährleistung des gegenseitigen Ausschlusses beim Zugriff auf gemeinsam genutzte Ressourcen können jedoch Probleme wie zum Beispiel *Deadlocks* auftreten. Wichtig ist auch, dass der Einsatz von Locks dazu führen kann, dass Abstraktionsebenen verloren gehen. Um diesen Problemen zu begegnen wurde das Konzept des Software Transactional Memory (STM) entwickelt. STM erlaubt es, auf einer höheren Ebene einen Block von Aktionen als *atomar* zu kennzeichnen. Das darunterliegende System sorgt dann dafür, dass sich das Programm so verhält, als würden diese Aktionen atomar ausgeführt. In Haskells Implementierung von STM wurde dieses Konzept noch um sequenzielle und alternative Komposition und die Möglichkeit einen Thread zu blockieren, bis eine bestimmte Bedingung erfüllt ist, erweitert.

Doch trotz Software Transactional Memory ist die Entwicklung von nebenläufigen Programmen noch immer schwierig. So kann es für den Entwickler recht schwierig sein, zu verstehen, wie sich komplexe Transaktionen verhalten.

Aus diesem Grund habe ich im Rahmen meiner Diplomarbeit den bereits vorhandenen graphischen Concurrent Haskell Debugger erweitert. In meinem Vortrag werde ich den erweiterten Concurrent Haskell Debugger vorstellen und demonstrieren, wie sich dadurch Transaktionen visualisieren lassen. Zusätzlich werde ich erläutern, wie sich die in den Concurrent Haskell Debugger eingebaute automatische Suche nach Deadlocks auch bei Programmen mit Software Transactional Memory einsetzen lässt.

Actions in the Twilight: Eine Erweiterung zu Software Transactional Memory

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
ariem@cs.uu.nl

Transactional Memory ([2],[1]) hat sich zu einer vielversprechende Alternative für multi-threaded Anwendungen entwickelt. Im Vergleich zu traditionellen locking-basierten Verfahren vereinfacht es das Programmieren von nebenläufigen Anwendungen, indem es transparent und feingranular Daten vor gleichzeitigem Zugriff durch mehrere Threads sichert und zugleich Modularität, Skalierbarkeit und Wiederverwendung von Code bietet. Es sind mittlerweile auch Software-Implementierungen verfügbar, die sich durchaus kompetitiv zu anderen Synchronisationsparadigmen verhalten.

Programmierer, die Software Transactional Memory (STM) in ihren Anwendungen einsetzen wollen, stehen jedoch in der Praxis vor einem Problem: Die Verwendung von bestehenden APIs ist i.d.R. nicht möglich, da diese oft auf Protokollen mit Handshake oder Locking basieren, um I/O und andere Systemservices bzw. nebenläufige Datenstrukturen zu nutzen. Dies wiederum steht im Konflikt zur Nutzung von STM Operationen.

In letzter Zeit wurden Vorschläge zur sicheren Integration dieser Operationen in das TM Paradigma unterbreitet. Welc et al. [4] und Spear et al. [3] schlagen hierzu *irrevocable* und *inevitable* Transaktionen vor. Bei der Ausführung solcher Transaktionen wird eine Serialisierung der laufenden Transaktionen erzwungen, um so Konflikte und Rollback zu vermeiden. Diese Serialisierung reduziert die dem Programm inhärenten Nebenläufigkeit, was zu einer niedrigeren Auslastung des Systems führt. Darüber hinaus verhindert es auch die Integration von Handshake-basierten Protokollen innerhalb von Transaktionen.

Twilight STM ist eine mächtige STM Implementierung, die es erlaubt, mehrere Transaktionen, deren Commit nicht fehlschlagen dürfen, nebenläufig abzuschliessen. Hierzu stellt Twilight STM neben den typischen STM Lese- und Schreiboperationen eine zweigeteilte Commitphase bereit. Zu Begin des Commits wird zunächst die Gültigkeit der Speicheroperationen geprüft, bevor die geschriebenen Werte für andere Transaktionen zum Lesen und Schreiben freigegeben werden. Im Vergleich zu anderen Zwei-Phasen-Commitprotokollen kann der Programmierer in der sogenannten *Twilight Zone* zwischen der Prüfung auf Konflikte und tatsächlichem Commit flexibel und in Abhängigkeit der jeweiligen Anwendung auf Konflikte reagieren. Durch Korrektur der zu schreibenden Werte kann dann beispiel-

sweise auf ein Rollback verzichtet werden. Eine erweiterte API unterstützt die Inspektionen von Konflikten und deren Korrektur.

Darüberhinaus können in der Twilight Zone I/O Operationen, Systemservices oder Handshake-Protokolle ausgeführt werden. Diese externe Operationen sind direkt global sichtbar und erlauben es Transaktionen miteinander zu kommunizieren. Obgleich wir Garantieen (Freiheit von Deadlocks, weak atomicity, progress) für die Twilight API geben können, obliegt es dem Programmierer, diese Eigenschaften für seinen Twilight Code sicherzustellen.

Unsere Benchmark-Ergebnisse zeigen, dass durch Twilight STM anwendungsspezifische Strategien zum *contention management* einfach und erfolgreich umgesetzt werden können. Desweiteren lassen sich die Twilight Zones zum Debuggen von STM Applikationen effektiv einsetzen.

References

1. Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.
2. Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. 14th ACM Symp. PODC*, pages 204–213, Ottowa, Ontario, Canada, 1995. ACM Press, New York, NY, USA.
3. M. F. Spear, M. M. Michael, and M. L. Scott. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing, TRANSACT'08, Salt Lake City, UT.*, 2008.
4. Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, New York, NY, USA, 2008. ACM.

Persistent Analysis Results [1]

Adrian Prantl, Jens Knoop and Markus Schordan

Institut für Computersprachen
Technische Universität Wien
email: {adrian, knoop}@complang.tuwien.ac.at
and
University of Applied Sciences Technikum Wien
email: schordan@technikum-wien.at

Abstract In this article we propose a new approach for storing the results of static program analyses to make them persistent and accessible for future use by both application programmers and tools. There are two key ideas. First, to attach the analysis results to the source code instead of the intermediate representation or the control-flow graph; we call this *behaviour-carrying code*. Second, to prepare this code to make analysis results automatically checkable by both static and dynamic verifiers. This opens up multiple new applications for program and tool development for compiler developers and more generally software engineers. The approach has been implemented in the *SATIrE* source-to-source analysis framework using the *Termite* program transformation library. First experiments show encouraging results and proved already valuable to increase the quality of analyzer implementations.

1 Motivation

Static program analysis aims at computing information about the runtime behaviour of a program without executing it. Theoretically well-founded are approaches based on data-flow analysis and abstract interpretation [Kil73, CC77, Hec77, Muc97]. Most commonly the information computed for a program point holds either *universally*, i.e., for every program execution reaching this point, or *existentially*, i.e., for some program execution reaching it. On the algorithmic side, this corresponds to *must* and *may* program analyses, respectively. *Available expressions* and *reaching definitions* are typical examples of properties computed by *must* and *may* analyses, respectively:

[1] This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract No P18925-N13, *Compiler Support for Timing Analysis (*(*CoSTA*), `http://costa.tuwien.ac.at/`, and the Commission of the European Union within the 7th EU R&D Framework Programme under contract No 215068, *Integrating European Timing Analysis Technology* (*ALL-TIMES*), `http://www.mrtc.mdh.se/projects/all-times/`.

– Available Expressions: An expression is computed along *every* path reaching a particular program point (must-information)
– Reaching Definitions: A definition of a variable is valid along some program path reaching a particular program point (may-information)

Results of analyses like for available expressions or reaching definitions are the basis of optimizing program transformations applied by a compiler. For example, variables computed to have a unique constant value at runtime can be used to replace expressions by their values; a transformation known as *constant folding*. Optimizing compilation, however, is not the only domain where results of program analyses are valuable. Another important domain concerns non-functional program properties such as resource consumption in terms of time or memory usage. Programmers often need to get a better grip on such run-time characteristics of the completed software, and thus need to get information on a program's spatial and temporal worst-case behaviour.

These days, the fields of program optimization and resource analysis are essentially served by different communities – often with little sharing between. For example, in the field of resource analysis a broad range of analyzers exists, but few are integrated with a compiler. One reason for this separation might be the data formats used to store analysis results. In a typical compiler, analysis results are temporary information which are calculated on demand for the purpose of a particular optimization. For this reason, they are usually attached to the control-flow graph (CFG) of a program and are thus not easily visible to the outside world. While adequate and efficient for the purpose of optimizing compilation, this has the drawback of impacting the portability of these results in order to make them amenable for other uses. This of course hurts the goal of interoperability between various tools. Next we present our approach to improve on this situation.

2 The Source-to-Source Workflow

In principle, a source-to-source program analyzer can be viewed as a portable stand-alone component that can be plugged before an off-the-shelf compiler system and that outputs analysis results in some external format (cf. [Sch08]). The resulting workflow is illustrated in Figure 1:

1. Perform a (potentially complex and expensive) analysis to some source program.
2. Attach the analysis results to the source code itself.

This way the analysis results become explicitly visible for the user. We call this *behaviour-carrying code* (*BCC*), in analogy to *proof-carrying code* (*PCC*) introduced by Necula and Lee [Nec97, NL98], which has a similar flavour. Besides informing the user about certain aspects of the run-time behaviour of the program, the BCC can be subject to other analyzers extending the behaviour information further. Moreover, at any step, the BCC can be passed on to the

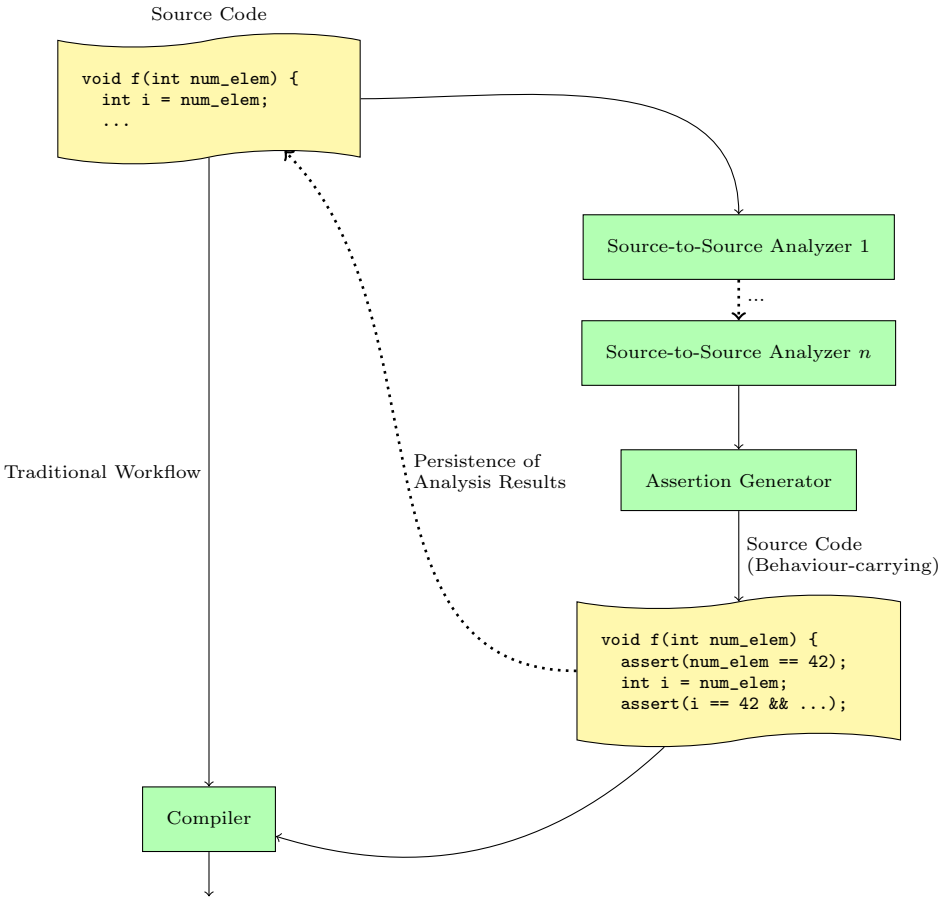


Figure 1. Source-to-source analysis work flow

compiler to proceed with compiling it and to generate executable code. Transformations applied by the various stages of the compiler, in particular its optimizer stage, can directly refer to this information without need to (re-) compute it; thus speeding-up the compilation process.

The feedback of the BCC of a source program into the workflow of the source-to-source analyzer is illustrated in Figure 1, too, making up the third and fourth step of the complete workflow:

3. Feed the BCC of a source program to the source-to-source analyzer obtaining a new BCC with extended behaviour information.
4. Stop analyzing the BCC and pass it on to the compiler.

Considering constant propagation as an example Figure 1 suggests another application mode, which generally leads to stronger analysis results. This requires that the source-to-source analyzer can operate across the boundaries of program modules. This way, it can benefit by checking and using the information on variables with constant values of used (and already analyzed) modules for strengthening the analysis results for the module currently under consideration.

"Plain" BCC is already useful to inform the user on certain aspects of the run-time of a program and can easily be reused and exploited by other program analyses typically applied throughout the compilation process. Next we demonstrate how to prepare plain BCC in order to make analysis results automatically checkable, both by static and dynamic verifiers. This opens up multiple new applications as we illustrate by three examples subsequently.

3 Checkable Assertions

Intuitively, the results of a program analysis provide information about the possible states of the underlying program at the various program points. Considering the two points immediately before and after the execution of a statement, this information can be thought of in terms of a pre- and a postcondition describing a property of the program states which may be valid immediately before and after executing this statement. Using the analogy of a (valid) *Hoare assertion* [Hoa69],

$$\{P\}\ Q\ \{R\}$$

where P is the precondition of a program Q and R a postcondition capturing the effect of Q wrt P, P and R are *assertions* which ensure a certain property of program Q. In the same fashion also the results of a program analysis attached to a statement can be considered assertions about the program behaviour at this statement. Some programming languages offer an integrated concept of assertions such as C in terms of *assert()* macros or Eiffel in terms of *contracts*. These language-provided assertions can be automatically checked at run-time and are useful for security checks. In the following we present our approach of how to present analysis results such that they receive the flavour of assertions, which can then be subject of static and dynamic verifiers.

To this end we define a program transformation that modifies the analysis information attached to a program such that the attached analysis information can be used like an assertion. It is worth noting that this does not affect the semantics of the program while offering the advantage of having the analysis information immediately available.

Conceptually, this program transformation can be thought of to work by transforming the analysis information attached to a program point into an assertion. An implementation might be organized to directly output assertions and skipping the intermediate step of outputting plain analysis information. In our approach, we distinguish assertions addressing *universally* and *existentially* valid program properties, respectively. Using the C language as source language for demonstration, we will show next how to express the two kinds of assertions for this setting.

We use the following notational conventions: P denotes the result-predicate from the ideal analysis, $\mathcal{M}_P$ the set returned by the implemented analyzer and L the location of interest in the analyzed program.

3.1 Assertions for Universally Valid Information

Intuitively, an information is *universally valid* at a location L if it is valid on every path reaching L. Program analyses computing universally valid information are commonly called *must*-analyses. Representing universally valid information of an analysis in terms of an assertion requires that the assertion specifies a test for the respective analysis information. This is illustrated by the following example:

Example 1. Constant propagation (a *must*-analysis) yields universally valid information in terms of expression-constant pairs: $P(exp, const) : exp = const$. In the C programming language, an assertion expressing this information can take the form `assert(exp == const)`.

Following [ALSU07], an analysis result is *safe* if it does not report any false positives. Using the notion of safety, the above example can be generalized for any universal information as summarized in Observation 1. Intuitively, this observation follows immediately from the definition of safety and the fact that an assertion `assert(`$\mathcal{M}_P$`)` will fail, if the result is not safe.

Observation 1. If an analysis reports a set $\mathcal{M}_P$ to universally fulfill a predicate P at location L, a statement `assert(`$\mathcal{M}_P$`)` inserted at L will evaluate to *true*, iff $\mathcal{M}_P$ is *safe*.

3.2 Assertions for Existentially Valid Information

Intuitively, information is *existentially valid* at a location L if it is valid on some path reaching L. Program analyses computing existentially valid information are commonly called *may*-analyses. Since existentially valid information need not hold along all paths, an assertion addressing such a property cannot restrict

itself to simply specify a test for it (obviously, the result *can* be false). Instead, it must specify a test for the precision of the analysis information. The following example illustrates this idea.

Example 2. A *may*-alias analysis yields sets of potentially aliasing pointer variables: $P(p_1, ..., p_n)$: $addr(p_i) = addr(p_{i+1})$, $i \in \{1, \ldots, n-1\}$. An assertion has the form `assert(q1 != q2 && q1 != q3 && ... &&` $\texttt{q}_{n-1}$ `!=` $\texttt{q}_n$`)`, where $q_i \neq p_j$ are all remaining pointer variables. For a program with k pointer variables and an analysis result containing j aliasing variables, the assertion consists of $n * (n-1)/2, n = k - j$ comparisons.

We call an analysis result *precise* if there are no other solutions. This means that any combination of parameters which is not part of the analysis result can not be in P. Putting all this together, leads us to our second observation, which, intuitively, follows from the definition of precision and the reasoning that the analysis result includes all solutions which can be seen as follows: For all tuples $\bar{x}$ that are not in the analysis result P is not satisfied. $\forall \bar{x} \notin \mathcal{M}_P \; . \; \neg P(\bar{x}) \iff \forall \bar{x} \in P . \bar{x} \in \mathcal{M}_P \iff P \subseteq \mathcal{M}_P$.

Observation 2. If an analysis reports a set $\mathcal{M}_P$ to existentially fulfill a predicate P at location L, a statement `assert(`$\forall \bar{x} \notin \mathcal{M}_P \; . \; \neg P(\bar{x})$`)` will evaluate to *true*, iff $\mathcal{M}_P$ is *precise*.

We have now shown how to transform the program to explicitly contain meta-information about itself that was brought forward by an analysis.

4 Applications for behaviour-carrying code

In Section 2 we have already seen how the source-to-source workflow allows for a more flexible interaction of analysis tools and compilers. By interlacing the analyzed information with the source code the subsequent system compiler can take advantage of new analyses without the necessity to re-implement them. Behaviour-carrying code also serves to make the analysis results persistent. By encoding analysis results in the source code they can be stored for future use in a universally recognizable data format. Re-using this information then only requires to read this information and to run the analysis again in order to re-transform the external format into the internal one. It is worth noting that this does not require a fixed point iteration rather a single visit of each node. In many cases even a weaker analysis will suffice, since the information is now available in a more explicit fashion.

In this section we present two applications of behaviour-carrying code – one targeted at software engineers and one targeted at compiler developers.

4.1 Behaviour-carrying code for the end-user

Bring forward informal interfaces. Behaviour-carrying code in its "plain" incarnation is not only useful in the program analysis and compilation work

flow but can also prove a valuable tool for the software engineer. With the help of BCC, the software engineer can get a clearer picture of large software projects. It can serve as *documentation* and *code understanding* tool. Often engineers are confronted with the maintenance of complex legacy systems that are not fully understood and/or documented. When faced with the task to add a new feature to such a system it is crucial to know the behaviour of the existing system beforehand. We believe that behaviour-carrying code can be a valuable means for documenting a *specification* extracted from legacy projects via static analyses.

Freeze interfaces against unwanted modifications. This idea can be further refined in the respect of fully behaviour-carrying code. We can use a static program analysis to reveal invisible interface conventions and to use the BCC as a means to future-proof the interface. Figure 2 shows an example how this could be applied in practice.

```
enum e_types classify() { ... }
...
void f()
{
  enum e_types c = classify();
  assert(c >= 0 && c < 4);
```

Figure 2. Enforcing interfaces with an interval analysis

In this example a larger piece of software has been analyzed by an interval analysis. This analysis computes the intervals of values that an integer variable holds at each location. By executing the assertions of the analysis results the transformed program virtually performs a dynamic range check for the used range of the value domain. These restricted value ranges often reflect implicit conventions introduced by the programmers and can be visualized with behaviour-carrying code. The programmers can review the generated assertions and use them to freeze certain aspects of the software against undesirable future modifications.

4.2 For the authors of analyzers: Automatic testing

In our daily work of implementing static program analyses the topic of testing is an important issue. Realistically it is not possible to create a bug-free implementation of a program analysis without thorough testing, since it is very easy to miss certain corner-cases. Bugs in the analyzer can have two reasons: Either they come from an incomplete or even wrong specification or they can be traced to errors in the implementation. In both cases, they will surface as an analysis result that can not be validated against the program at run-time. By applying the behaviour-carrying code transformation to an analyzed program,

the resulting program can be used as a test for the analysis: If an assertion fires, the analysis result that was tested for did not match the reality. This method of simply executing the transformed program, however, does have the drawback of uncertain path coverage. Typical test suite programs often lack proper entry points/*main*-functions or are otherwise incomplete. To overcome this issue, one solution is to also generate stubs for these missing functions.

In the SATIrE implementation, for example, we create an artificial main function that calls the other functions in the module that are to be tested. Such a generated main function of course needs to adhere the calling conventions of the other functions in the module. For parameterless functions, this is easily achieved, otherwise it is necessary to fake input values depending on the function signature. A naïve approach would be to call every function in the module. To improve the performance when testing multiple combinations of input values, it pays off to take the call graph into account to reduce the overall number of function calls. By calculating the transitive closure of the call graph, a minimal set of function "clusters" can be identified. One call to the first node of each cluster guarantees that every function is reachable from the main function.

If the execution speed is circumstantial, there is also the option to use a formal verification engine to prove the assertions. (Bounded) software model checkers such as Blast [BHJM07] and CBMC [CKL04] fill this niche. In our experience CBMC seems to be more suited for this kind of programs as it was consistently faster in our experiments. Using a model checker, we can get a verification of the analysis result. If the model checker produces a counter-example we know that either the analysis specification, analysis implementation or the model checker is erroneous. It must be noted that the runtime of model checkers can be quite unpredictable. In our experiments the model checker failed to terminate within reasonable time or space requirements for a substantial number of test cases. This is because, in the worst case, the model checker has to examine every vector of the state space of the program.

As an interesting twist, the combination of model checking and automatic assertion insertion can also be used in the opposite direction. Since the model checker can find out whether an analysis result is valid, we can turn the tables by repeatedly making an *informed guess* until the model checker can prove it to be correct. Using search strategies like binary widening and binary tightening we can "analyze" information where cheaper methods fail. For a detailed experience report the reader is referred to [PKK+09].

From our experience we can conclude that a combination of test executions and model checking (with sensitive time- and memory limitations) is a valuable means to aid the development of static program analyzers. It helped us to pinpoint problems introduced by recent changes in the interprocedural control flow graph construction that otherwise would probably have gone unnoticed until much later.

5 Implementation

We implemented the BCC transformer for the SATIrE source-to-source analyzers. The analyzers generated by SATIrE operate on an interprocedural control flow graph built from a C++ abstract syntax tree (AST). The analysis results are stored in attributes of the AST. The transformer implementation was written in Prolog using the TERMITE[1] library. This library contains abstractions, traversals and query predicates that facilitate operating on the external term representation that is exported by SATIrE analyzers.

Figure 3 shows the relevant portion of a BCC transformer for a loop bound analysis. A loop bound analysis is concerned with finding upper bounds for the trip count of loop constructs. The transformer works by introducing a new unsigned integer variable that is initialized to zero before the loop is entered. The new variable is incremented by one for each iteration of the loop body. After the loop, an assertion is inserted stating that the counter variable is smaller or equal than the loop bound. An example of a program transformed this way is shown in Figure 4. The transformer reads the loop bound attribute from the loop body, generates the counter and places the loop inside of a new compound statement[2] that serves as scope for the counter variable. The term representation also contains bookkeeping information about source file location of every node, which was replaced by "..." for improved readability in the listing.

6 Extensions: Context sensitivity

In this section we investigate how to take context information tethered to call strings into account, which leads to more precise interprocedural analyses [SP81]. Intuitively, call strings allow keeping call contexts up to a predefined length n (= length of the call string) separate. As an illustrating example, the result of an interprocedural interval analysis with call string length 1 and its context-insensitive counterpart is given in Figure 5. To account for context-sensitivity in behaviour-carrying code, several options are available:

1. Introducing an extra *call-string*-argument to every function call that keeps track of the current context. This has a slight impact on performance and dynamic memory footprint caused by the additional bookkeeping.
2. Generating explicit copies of the deepest n functions in the expanded call graph. While this transformation is likely to increase execution speed, it has the drawback of increasing code size significantly.

Both methods involve a major refactoring of the original program. While a less invasive method would seem preferable, it has to be noted that the modified program also carries an increased potential for subsequent optimizations that is comparable to that of inlining.

[1] Termite Web Page: `http://www.complang.tuwien.ac.at/adrian/termite/`

[2] For historic reasons the compound statement is misleadingly called "basic block" by the underlying ROSE compiler.

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

  counter_decl('__bound', ..., CounterDecl),
  counter_inc('__bound', ..., Count),
  counter_assert('__bound', N, ..., CounterAssert),

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

Figure 3. Source-to-source transformer for loop bounds

```
int binary_search(int x) {
  int fvalue, mid, up, low = 0, up = 14;
  fvalue = -1; /* all data points are positive */
  {
    unsigned int __bound = 0;
    while(low <= up){
      ++__bound;
      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;
    }
    assert(__bound <= 7);
  }
  return fvalue;
}
```

Figure 4. Generated assertions for loop bounds

```
int g(int x) {
// pre info: Context(0,0):[(TOP,TOP):x->(0,0)]
//           Context(0,1):[(TOP,TOP):x->(42,42)]
//           Merged:[(TOP,TOP):x->(0,42)]
  return x + x;
// post info: Context(0,0):[(TOP,TOP):x->(0,0),$tmpvar$retvar->(0,0)]
//            Context(0,1):[(TOP,TOP):x->(42,42),$tmpvar$retvar->(84,84)]
//            Merged:[(TOP,TOP):x->(0,42),$tmpvar$retvar->(0,84)]
}

int f(int x) {
  return g(0);
}

int h(int x) {
  return g(42);
}
```

Figure 5. Calling contexts and merged interval information

7 Conclusions

With behaviour-carrying code we presented a portable way to make the results of static program analyses available for a wide range of applications. While the testing of analyzers will be of interest particularly for compiler developers, the interoperability perspectives could prove most useful and valuable for other users as well. As we have seen in the loop bound example, the transformation is not obvious for all types of analyses. However, what might look as a shortcoming of the approach can also be regarded as a chance: By compiling analysis results into information that can be validated easily (and most important: automatically and locally!), compilers can suddenly profit from new analyses without having to re-implement them. This way we hope to increase the collaboration between stand-alone static analyzers and compiler systems in the future.

The validation of static program analyzers provided the original inspiration for the presented transformation and was already successfully implemented in the static analysis framework SATIrE.[3]

Acknowledgements The authors would like to thank Gergő Bárány for many helpful comments on earlier versions of this article.

References

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Pearson Education, second edition, 2007.

[3] SATIrE web page: `http://www.complang.tuwien.ac.at/satire/`

[BHJM07] 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.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CKL04] 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.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[Nec97] George C. Necula. Proof-carrying code. In *Conf. Rec. 24th Annual Symp. on Principles of Prog. Lang. (POPL'97)*, pages 106 – 119. ACM, NY, 1997.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5):333–344, 1998.

[PKK+09] Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec, and Markus Schordan. From trusted annotations to verified knowledge. In *On-Site Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009)*, Dublin, 2009.

[Sch08] Markus Schordan. Source-to-source analysis with satire - an example revisited. In Florian Martin, Hanne Riis Nielson, Claudio Riva, and Markus Schordan, editors, *Scalable Program Analysis*, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

Tracing the Meta-Level: PyPy's Tracing JIT Compiler

Carl Friedrich Bolz
Heinrich-Heine-Universität Düsseldorf
Institut für Informatik
Softwaretechnik und Programmiersprachen

We attempt to apply the technique of Tracing JIT Compilers [3, 2] in the context of the PyPy project[1][4, 1], i.e. to programs that are interpreters for some dynamic languages, including Python. Tracing JIT compilers can greatly speed up programs that spend most of their time in loops in which they take similar code paths. However, applying an unmodified tracing JIT to a program that is itself a bytecode interpreter results in very limited or no speedup.

In this talk we show how to guide tracing JIT compilers to greatly improve the speed of bytecode interpreters. One crucial point is to unroll the bytecode dispatch loop, based on two hints provided by the implementer of the bytecode interpreter. The technique is already mature enough to be applied to a number of example interpreters, but also to PyPy's full Python interpreter, giving interesting speedups.

References

[1] C. F. Bolz and A. Rigo. How to *not* write a virtual machine. In *Proceedings of the 3rd Workshop on Dynamic Languages and Applications (DYLA 2007)*, 2007.

[2] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, Nov. 2006.

[3] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 144–153, Ottawa, Ontario, Canada, 2006. ACM.

[4] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 944–953, Portland, Oregon, USA, 2006. ACM.

[1] `http://codespeak.net`

Optimizing High Abstraction-Level Interpreters*

Stefan Brunthaler

Institut für Computersprachen
Technische Universität Wien
brunthaler@complang.tuwien.ac.at

Abstract. Many users and companies alike feel uncomfortable with execution performance of interpreters, often also dismissing their use for specific projects. Specifically virtual machines whose abstraction level is higher than that of the native machine they run on, have performance issues. Several common existing optimization techniques fail to deliver their full potential on such machines. This paper presents an explanation for this situation and provides hints on possible alternative optimization techniques, which could very well provide substantially higher speedups.

Key words: Interpreters, Virtual-Machine Abstraction, Optimization Techniques.

1 Motivation

1000 : 10 : 1. These are the slowdown-ratios of an inefficient interpreter, when compared to an efficient interpreter, and finally to an optimizing native code compiler. Many interpreters were not conceived with any specific performance goals in mind, but rather striving for other goals of interpreters, among them portability, and ease of implementation. This also means that there is a huge benefit in optimizing an interpreter before taking the necessary steps to convert the tool chain to a compiler. There are common optimization techniques for interpreters, e.g. threaded code [Bel73],[Ert01],[EG03b], superinstructions [EG04], and switching to a register based architecture [SCEG08]. The mentioned body of work provides careful analyses and in-depth treatment of performance characteristics, implementation details.

Those optimization techniques, however, have one thing in common: their *basic assumption* is that interpretation's most costly operation is instruction dispatch, i.e., in getting from one bytecode instruction to its successor. While this assumption is certainly true for the interpreters of languages analyzed in the corresponding papers, e.g. Forth, Java, and OCaml, our recent results indicate that it is specifically not true for the interpreter of the Python programming language. We find that this correlates with a difference in the virtual-machine abstraction levels between their corresponding interpreters.

* This paper revises previous work, which has recently been presented at the *4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation* (*BYTECODE'09*), York, UK, 29th March 2009, part of ETAPS 2009 [Bru09].

In virtual machines where the abstraction level is very low, i.e., essentially a 1 : 1 correspondence between bytecode and native machine code, the basic assumption of dispatch being the most costly operation within an interpreter is valid. Members of this class are the interpreters of Forth, Java, and OCaml, among others. Contrary to those, interpreters that provide a high abstraction level do not support this assumption. The interpreters of Python, Perl, and Ruby belong here. Additionally, we analyze the interpreter of Lua, which is somewhere in between both classes.

In their conclusion on their technique, Piumarta and Riccardi [PR98] suppose the following: "*The expected benefits of our technique are related to the average semantic content of a bytecode. We would expect languages such as Tcl and Perl, which have relatively high-level opcodes, to benefit less from macroization. Interpreters with a more RISC-like opcode set will benefit more — since the cost of dispatch is more significant when compared to the cost of executing the body of each bytecode.*" Our work shows, whether their expectations turn out to be correct, and makes explicit what is only implicitly indicated by their remark. Specifically we contribute:

- We categorize some virtual machines according to their abstraction level. We show which characteristics we consider for classifying interpreters, and provide hints regarding other programming languages than those discussed.
- We subject optimization techniques to that categorization and analyze their potential benefits with respect to their class. This serves as a guideline for a) implementers, which can select a set of suitable optimization techniques for their interpreter, and b) researchers which can categorize other optimization techniques according to our classification.

In contrast to our previous work [Bru09], this paper presents additional optimization techniques for high abstraction level interpreters, as well as a more concise representation of the examples and the differences between the classes of low and high abstraction level virtual machines.

2 Categorization of Interpreters

To get a big picture on the execution profile of the Python interpreter, we collected 9 million samples of instruction execution times running the `pystone` benchmark on a modified version of the Python 3.0rc1 interpreter, which samples CPU cycles. We sampled *Operation Execution*, *Dispatch* and *Whole Loop* costs. Operation execution contain the number of cycles spent for all native machine instructions in operation implementation. Dispatch costs contain the number of cycles spent for getting from one operation to another, e.g. in a switch-statement from one case to another. Python's interpreter, however, does not dispatch directly from one bytecode to another, but maintains some common code section which is conditionally executed before dispatching to the next instruction. To account for that special case, we measured so called whole loop costs, which

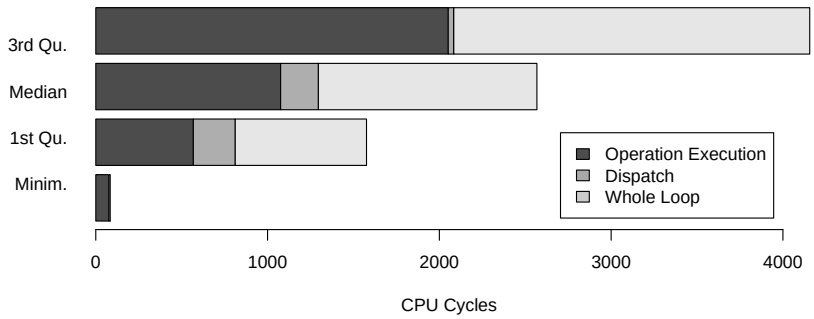


Fig. 1. CPU cycles per code section for Minimum, 1st quartile, median, and 3rd quartile measures.

measure the CPU cycles from the first and last instructions within the main loop.

Based on extensive previous work, [Bel73],[Ert01],[EG03b],[EG04],[SCEG08], we expected that instruction dispatch would be the most costly interpreter activity for Python's virtual machine, too. Figure 1 shows our results obtained by examining CPU cycles for the Python 3.0rc1 interpreter, running on a Pentium 4, 3 GHz, with Xubuntu 8.04.

Section	Min	1st Quartile	Median	3rd Quartile
Op-Execution	75	568	1076	2052
Dispatch	84	812	1296	2084
Whole Loop	84	1576	2568	4156

Table 1. Minimum, 1st quartile, median, and 3rd quartile values for CPU cycles per code section. All values here are *inclusive*, i.e., values for *Dispatch* include *Operation Execution*, and *Whole Loop* includes values of *Dispatch*, and *Operation Execution* respectively.

Our results do not support the assumption of instruction dispatch being the most costly operation for the Python interpreter, actually operation execution is. For a detailed explanation of why this is, we present a comparative example of one instruction implementation of the interpreters of Java, and Python Section 2.1. The corresponding results of OCaml and Lua can be found in the original BYTECODE'09 paper [Bru09].

Java, according to the latest Java Language Specification [GJSB00], the Java instruction set consists of 205 operations, including reserved opcodes. Instructions are typed for the following primitive types: integers, longs, floats, doubles, and addresses. In our example we take a look at the Sable VM, version 1.13.

Python, is a multi-paradigm dynamically typed programming language, that enables hybrid object-oriented/imperative programming with functional programming elements [pyt08]. It has 93 instructions in Python 3.0rc1. Most of its operations support ad-hoc polymorphism, e.g. `BINARY_ADD` concatenates string operands, but does numerical addition on numerical ones [pyt].

It is worth noting that our results are not restricted to those programming languages only. Actually, we conjecture that this is true for the interpreters of programming languages with similar characteristics, i.e., for the Python case this also includes Perl [per], and Ruby [rub].

2.1 Categorization based on the comparative addition example

Our classification scheme requires the assessment of the abstraction level of several virtual machines interpreting different languages. In order to do so, we take a representative bytecode instruction present in all our candidates and analyze their implementations. This enables us to show important differences in bytecode implementation, and in consequence allows us to classify them accordingly.

The representative instruction we use for demonstration is integer addition, e.g. for Java we take a look at `IADD`, for Python `BINARY_ADD`. We refer the reader to our previous work [Bru09] for details on OCaml's `ADDINT` and Lua's `OP_ADD`.

We have highlighted the relevant implementation points by using a bold font, and use arrows for additional clarity.

```
case SVM_INSTRUCTION_IADD:
    {
         jint value1 = stack[stack_size - 2].jint;
         jint value2 = stack[--stack_size].jint;
         stack[stack_size - 1].jint= value1 + value2;
         break;
    }
```

Fig. 2. Implementation of Java's integer addition operation, `IADD` in Sable VM v1.13.

As we can see in Figure 2, the SableVM does not implement addition on the virtual machine level, but expresses the bytecode addition by leveraging the addition used by the compiler, i.e., exposes the addition instruction of the native machine. Consequently, the virtual machine addition is expressed using a single native machine instruction. This constitutes our class of *low abstraction*

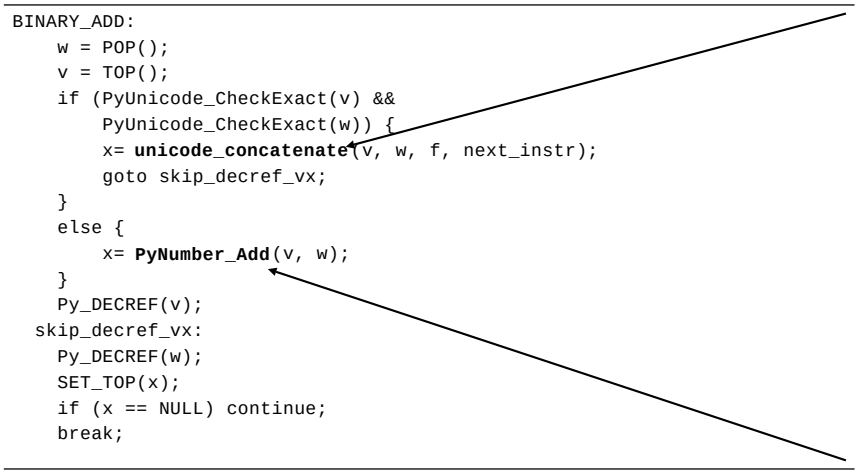


Fig. 3. Implementation of integer addition in Python 3.0rc1.

level virtual machines, where the interpreter is only a thin layer above a real machine.

Python's case (Figure 3) shows a contrary picture: the upper arrow shows that `BINARY_ADD` does unicode string concatenation on string operands by calling `unicode_concatenate`. On non-string operands it calls `PyNumber_Add`, which implements dynamic typing and chooses the matching operation based on operand types, indicated by the lower arrow. In our integer example, the control flow would be: `PyNumber_Add`, `binary_op`, and finally `long_add`. If, however, the operands were float, or complex types, then `binary_op` would have diverted to `float_add`, or `complex_add`, respectively (cf. Figure 4).

Aside from this ad-hoc polymorphism, the addition in Python 3.0 has an additional feature: it allows for unbounded range mathematics for integers, i.e., it is not restricted by native machine boundaries in any way. As a direct consequence, the original Python `+`-operator bound add instruction cannot be directly mapped onto one native machine instruction in the interpreter. This constitutes our second class, namely *high abstraction level virtual machines*.

Our classes are by no means completely separated and disjoint, since interpreters can be members of both classes, having some subset of instructions belong into one set, and a separate subset of instructions to the other. This is exemplified in Lua in which addition has characteristics of both classes (cf. Figure 6 in [Bru09])

2.2 Comparison of Low and High Abstraction Level Virtual-Machines

The previous section introduces our two classes of interpreters, namely:

– Low abstraction level, where operation implementation can be directly translated to a few native machine instructions. Figure 5(a) shows the implementation of the interpreter's add instruction, and how the actual add is realized using a single machine add instruction.
– High abstraction level, where operation implementation requires *significantly* more native machine instructions than for low abstraction level. Analogous to Figure 5(a), Figure 5(b) shows the relative impact of implementing a complex add. Frequent characteristics for high abstraction level are:
  - *Ad-Hoc Polymorphism*: a) either polymorphic operations are selected for concrete tuples of operand types, e.g. in Python, or b) operand-type coercion into compatible types for a given operation implementation, e.g. in Lua or Perl.
  - *Complex Operation Implementation*: In Python's case, this complexity directly maps to unbounded integer range mathematics for numeric operations, or full unicode support at the interpreter level.

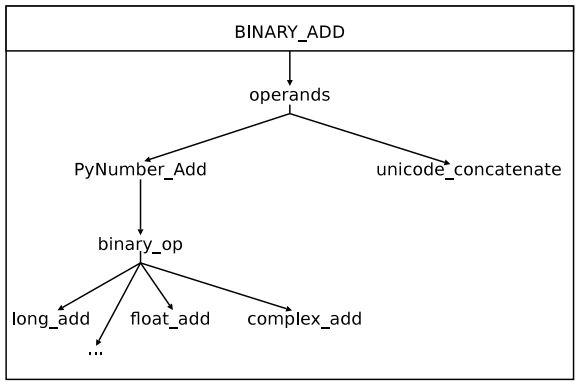
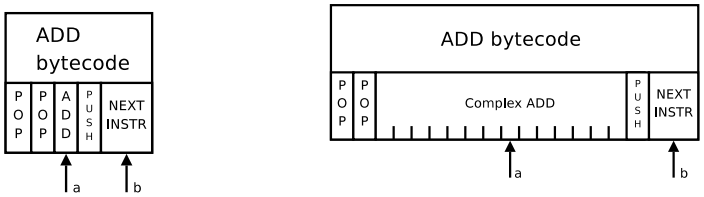


Fig. 4. Ad-hoc polymorphism in Python's `BINARY_ADD` instruction.



(a) Low Abstraction Level Virtual-Machine

(b) High Abstraction Level Virtual-Machine

Fig. 5. Illustration of Virtual-Machine Abstraction Levels. Important are the different ratios of $a : b$ which affects the relative optimization potential of various optimization techniques.

3 Optimizations for Low Abstraction Level Interpreters

The well known techniques for interpreter optimization focus on reduction of instruction dispatch cost. As shown in Figure 5(a), virtual machines with low abstraction level are particularly well suited for those techniques, since dispatch often is their most expensive operation. Threaded code [Ert01] reduces the instructions necessary for branching to the next bytecode implementation. The regular switch dispatch technique requires 9-10 instructions, whereas e.g. direct threaded code needs only 3-4 instructions for dispatch [EG01], with only one indirect branch. Superinstructions [EG04] substitute frequent blocks of bytecodes into a separate bytecode, i.e., they eliminate the instruction-dispatch costs between the first and last element of the replaced block.

Recent advances in register based virtual machines [SCEG08], however, suggest a complete architectural switch from a stack-based interpreter architecture to a register based model. This model decreases instruction dispatches by eliminating a large number of stack manipulation operations, i.e., the frequent LOAD/STORE operations that surround the actual operation. The paper reports that 47% of Java bytecode instructions could be removed, at the expense of growing code size of about 25%. Table 2 shows a list of achievable speedups for low abstraction level interpreters.

Optimization Technique	Speedup Factor	Reference
Threaded Code (compared to switch dispatch interpreter)	up to 2.02	[EG03b]
Superinstructions (compared to threaded code interpreter)	up to 2.45	[EG03a]
Replication + Superinstructions (compared to threaded code interpreter)	up to 3.17	[EG03a]
Register vs. Stack Architecture (both using switch dispatch)	1.323 avg	[SCEG08]
Register vs. Stack Architecture (both using threaded code)	1.265 avg	[SCEG08]

Table 2. Reference of reported speedup factors for several techniques.

For high abstraction level virtual machines, these speedups are not nearly as high. Vitale and Abdelrahman [VA04] actually report that applying their optimization technique to Tcl has *negative* performance impacts on some of their benchmarks, because of instruction cache misses due to complex operation implementation leading to excessive code growth—the main characteristic we use to identify high abstraction level virtual machines.

This, however, does not mean that these techniques are irrelevant for virtual machines with a high abstraction level. Actually, quite the opposite is true: once techniques for optimizing the high abstraction level are implemented (cf. Section 4), the ratio of operation-execution vs. instruction-dispatch (as indicated

by the arrows `a` and `b` in Figures 5(a) and 5(b)) has favorably changed the optimization potential of those techniques mentioned in Table 2. Therefore, our categorization merely provides an ordering of relative merits of various optimization techniques, such that considerable deviations in expected/documented vs. actually measured speedups are not stunningly surprising anymore.

4 Optimizations for High Abstraction Level Interpreters

Figure 5(b) shows that realizing a high abstraction level interpreter instruction, requires many native machine instructions. Therefore, we are interested in cutting down the costs here, since they provide the greatest speedup potential. Aside of the already mentioned complex operation implementation, the following list contains additional overheads often found in high abstraction level interpreters:

- ad-hoc polymorphism
- reference counting
- (un-)boxing

Therefore, successful optimization efforts need to address these issues. With respect to optimizing ad-hoc polymorphism, a look at the history of programming languages provides valuable insights: the efficient implementations of Smalltalk [DS84] and Self [HU94], [Höl95] present multiple possible solutions. Particularly, adding type feedback to our high abstraction level interpreters would be effective in reducing the overhead caused by ad-hoc polymorphism.

Garbage collection is another big issue in many interpretative systems, particularly those using reference counting. While simple to implement and not entirely without benefits, it has been shown to cause lots of overhead in stack-based interpreters, where many increments and/or decrements are caused by the regular stack manipulation. Deutsch and Bobrow [DB76] have noticed this and suggested the use of deferred reference counting to resolve this issue. Changing garbage collection in Python from reference counting to another technique has already been proposed multiple times, but has not been adopted by the standard distribution.

Finally, many programming languages that are interpreted by high abstraction level virtual machines do not offer primitive data types but use objects everywhere. Therefore, the actual data has to be unboxed before the actual processing and boxed after the processing. In Python 3.0, for instance, all integers are represented by instances of the `PyLong` class. Consequently, the interpreter is not able to just use a native-machine add instruction for the operands, since they do not represent the actual values but are instances of the `PyLong` class, i.e., pointers into the heap. Successful optimization in that case could add finer sub-structural types in combination with the type feedback mentioned above: if after the first execution we know that operands are not pointers to the heap but carry their actual data, a specialized add instruction could leverage an existing native machine add for operation implementation. This could be achieved by using tagged data.

5 Related Work

Romer et al. [RLV+96] provide an analysis for interpreter systems. Their objective was to find out whether interpreters would benefit from hardware support. However, they conclude that interpreters share no similarities, and therefore hardware support was not meaningful. Among other measurements, they collected average native instructions per bytecode (Section 3 of their paper). This is a sort of a black-box view on our classification scheme based on comparable source code examples from bytecode implementations. Finally, there is no link to optimization techniques, too.

Contrary to Romer et al. [RLV+96], Ertl and Gregg [EG03b] found that at least within the subset of efficient interpreters, hardware support in the form of branch target buffers would significantly improve performance for the indirect branch costs incurred in operation dispatch. Their in-depth analysis by means of simulation of a simple MIPS CPU found that the indirect branching behavior of interpreters is a major cause for slowdowns. Another important result of Ertl and Gregg is that a Prolog implementation, the Warren Abstract Machine based YAP [CRA], is very efficient, too: This implies that there is no immanent performance penalty associated with dynamically typed programming languages. Their class of efficient interpreters maps perfectly well to our category of low abstraction level virtual machines.

Adding to their set of efficient interpreters, they also provide results for the interpreters of Perl, and Xlisp—both of which achieve results that do not fit within the picture of efficient interpreters. This is where we introduce the concept of high abstraction level interpreters, and how it correlates to optimization techniques. Interestingly, for Xlisp they mention the following: "*We examined the [Xlisp] code and found that most dynamically executed indirect branches do not choose the next operation to execute, but are switches over the type tags on objects. Most objects are the same type, so the switches are quite predictable.*" This directly translates to our situation with high abstraction level interpreters (Section 4).

In his dissertation, Hölzle also notes that a problem for an efficient SELF interpreter would be the abstract bytecode encoding of SELF [Höl95], with a point in case on the `send` bytecode, which is reused for several different things. Interestingly, Hölzle observes that instruction set architecture plays a very important role for the virtual machine, and conjectures that a carefully chosen bytecode instruction set could very well rival his results with a native code compiler.

6 Conclusions

We introduced the classes of high and low abstraction levels for interpreters, and categorized some interpreted systems into their corresponding classes. Using them, we subjected various known optimization techniques for their relative optimization potential. Techniques that achieve very good speedups on low abstraction level interpreters do not achieve the same results for high abstraction

level virtual machines. The reason for this is that the ratio of native instructions needed for operation execution vs. the native instructions needed for dispatch, and therefore their relative costs changes. In low abstraction level interpreters the ratio usually is $1 : n$, i.e., many operations can be implemented using just one machine instruction, but dispatch requires n instructions, which varies according to dispatch technique used, and is costly because of its branching behavior. Quite contrary for high abstraction level interpreters: here operation execution usually consumes much more native instructions than dispatch does, which lessens the implied dispatch penalties.

Our classes are not mutually exclusive, an interpreter can have both, low and high abstraction level instructions. For our classes of high abstraction level interpreters, exemplified by Python and Lua—but conjectured to be true for Perl and Ruby, too—type feedback looks particularly promising. When faced with other programming languages but the same situation, i.e., a discrepancy in expected and reported speedups for low abstraction level techniques, our mileage may vary. In such a situation, only detailed analysis of an interpreter's execution profile can tell us where most time is spent and which techniques are most promising with regard to optimization potential.

In closing, we want to mention that our objectives are to demonstrate the relative optimization potential for different abstraction levels between an interpreter's virtual machine instruction set and the native machine it runs on.

Acknowledgments

I want to thank Jens Knoop and Anton Ertl for their feedback and comments on earlier versions of this paper, which has improved its presentation.

References

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[Bru09] Stefan Brunthaler. Virtual-machine abstraction and optimization techniques. In *Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '09)*, pages 19–30. Elsevier, 2009.

[CRA] CRACS. Yap–Yet Another Prolog. `http://www.dcc.fc.up.pt/~vsc/Yap/`.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the SIGPLAN '84 Symposium on Principles of Programming Languages (POPL '84)*, pages 297–302, New York, NY, USA, 1984. ACM.

[EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference on Parallel Processing*, pages 403–412, London, UK, 2001. Springer-Verlag.

[EG03a] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI '03)*, pages 278–288, New York, NY, USA, 2003. ACM.

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5, 2003.

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. [ivm04], pages 7–14. General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.

[Ert01] M. Anton Ertl. Threaded code variations and optimizations. In *EuroForth*, pages 49–55, TU Wien, Vienna, Austria, 2001.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Höl95] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Stanford, CA, USA, 1995.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 326–336, 1994.

[ivm04] *IVME '04: Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME '04)*, New York, NY, USA, 2004. ACM. General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.

[per] Perl. `http://www.perl.org`.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 291–300, New York, NY, USA, 1998. ACM.

[pyt] Python. `http://www.python.org`.

[pyt08] python.org. *Python v3.0 documentation*, December 2008. `http://docs.python.org/3.0/`.

[RLV+96] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *In Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 150–159. ACM Press, 1996.

[rub] Ruby. `http://ruby-lang.org`.

[SCEG08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. [ivm04], pages 42–50. General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.

Compatibility Criteria for Java Packages

Yannick Welsch
welsch@cs.uni-kl.de
Software Technology Group
University of Kaiserslautern

14.04.2009

In this talk, I analyze the interface evolution of Java packages regarding compatibility. I want to determine the criteria for a Java package implementation to be substitutable for another one in all possible contexts, leading to a program satisfying the typing rules and context conditions.

Although the Java Language Specification defines properties for binary compatibility, this notion is not as strong as source compatibility. In fact, source compatibility implies binary compatibility, but not vice versa. For example, introducing a new field with the same name as an existing field, in a subclass of the class containing the existing field declaration, does not break binary compatibility with preexisting binaries. However, at the source code level, this may lead to source incompatibility (type error). A new declaration is added, changing the meaning of a name in an unchanged part of the source code, while the preexisting binary for that unchanged part of the source code retains the fully-qualified, previous meaning of the name.

The two following topics can be addressed by giving criteria for source compatibility. Consider a setting where a library implementer develops a new version of the library. By respecting the compatibility criteria, he can ensure that the new version of his library will still compile with existing client code. Another far more complex topic is the following one. To define a notion of behavioral equivalence of two package implementations, a notion of compatibility is needed as only compatible package implementations can lead to identical behavior.

In an attempt to specify the criteria for package compatibility, I have formalized a Java subset in the spirit of Classic Java, enhanced by packages and access modifiers. The compatibility criteria I give are directly derived from the typing rules and context conditions.

As byproduct of my research on compatibility criteria, I was able to find a bug in the Eclipse Java compiler regarding overriding of methods across package boundaries.

JavaGI: bessere Interfaces für Java

Stefan Wehr

Institut für Informatik, Universität Freiburg
wehr@informatik.uni-freiburg.de

JavaGI erweitert Java 1.5 um ein verbessertes und allgemeineres Interface–Konzept. Das von Haskells Typklassen–Mechanismus beeinflusste Konzept erlaubt eine Reihe interessanter Features:

1. Implementierungen von Interfaces können retroaktiv erfolgen, d.h. abgekoppelt von den Definitionen der beteiligten Klassen und Interfaces sein.
2. Interfaces können Referenzen auf die implementierende Klassen enthalten. Damit lassen sich binäre Methoden spezifizieren.
3. Interfaces können sich über mehrere Typen erstrecken und damit verschränkt–rekursive Abhängigkeiten widerspiegeln.
4. Interfaces können nicht nur als Typen sondern auch in Typconstraints verwendet werden.
5. Bestimmte Methoden von Interfaces erlauben symmetrischen Multi–Dispatch.
6. Retroaktive Implementierungen von Interfaces können Bedingungen unterworfen sein, die es erlauben, Implementierungen auf bestimmte Typen einzuschränken.
7. Interfaces können statische Methode enthalten.

Diese Features subsummieren bestimmte Designpatterns, erhöhen die Ausdrucksstärke des Typsystems und erlauben einheitliche Lösungen von Problemen im Bereich der Erweiterung und Integration bestehender Softwaresysteme, für deren Lösung bisher mehrere verschiedene Spracherweiterungen nötig waren.

Die Sprache JavaGI besitzt eine Formalisierung, für welche Soundness und Entscheidbarkeit des Typsystems, sowie Determiniertheit der Ausführungsregeln bewiesen sind. Desweiteren gibt es eine Implementierung der Sprache, welche auf dem in der Entwicklungsumgebung "Eclipse" verwendeten Java Compiler beruht. Mit Hilfe dieser Implementierung wurde eine Reihe von Fallstudien durchgeführt, welche die praktische Anwendbarkeit der Sprache demonstrieren. Außerdem zeigen Bechmarks, dass der Compiler Code mit guter Performanz erzeugt.

Weitere Informationen zu JavaGI, inklusive des Compilers, sind im Internet unter der URL `http://www.informatik.uni-freiburg.de/~wehr/javagi/` zu finden.

Literatur

1. S. Wehr, R. Lämmel, and P. Thiemann. JavaGI: Generalized interfaces for Java. In E. Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 347–372, Berlin, Germany, July 2007. Springer.
2. S. Wehr and P. Thiemann. Subtyping existential types. In *10th FTfJP, informal proceedings*, 2008.

Zutaten für erweiterbare Programmiersprachen

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

Erweiterbare Programmiersprachen erlauben ihrem Benutzer, die Syntax der Sprache mehr oder weniger umfassend zu erweitern, beispielsweise durch die Definition zusätzlicher Operatoren und Kontrollstrukturen oder durch die Einführung neuer Deklarationsmöglichkeiten. Üblicherweise werden neue syntaktische Konstruktionen durch einen mehr oder weniger komfortablen und flexiblen Ersetzungsmechanismus auf vordefinierte Sprachmittel zurückgeführt.

Beispielsweise bietet der C/C++-Präprozessor die Möglichkeit, Makroaufrufe im Programmtext durch ihre (ggf. parametrisierte) Definition zu ersetzen und damit in begrenztem Maße syntaktische Erweiterungen der Sprache vorzunehmen. Common Lisp bietet einen wesentlich mächtigeren Makromechanismus an, der nahtlos in die Basissprache integriert ist, so dass der Ersatztext eines Makros bei Bedarf erst zur Laufzeit des Programms berechnet werden kann. Da Common Lisp außerdem eine sehr einfache und einheitliche syntaktische Grundstruktur besitzt (geklammerte Ausdrücke bestehend aus einem Funktions- oder Operatorsymbol und beliebig vielen Operandenausdrücken), lassen sich Erweiterungen ebenfalls nahtlos integrieren.

Dylan bietet ebenfalls einen mächtigen Makromechanismus an, bei dem zwischen Definitions-, Anweisungs- und Funktionsmakros unterschieden wird, mit denen unterschiedliche Teile der zugrundeliegenden Grammatik erweitert werden können. Moderne funktionale Sprachen wie z. B. ML oder Haskell bieten die Möglichkeit, benutzerdefinierte Operatorsymbole einzuführen und anschließend zu verwenden. Andere moderne Sprachen wie z. B. Scala oder Seed7 bieten ebenfalls bestimmte, zum Teil relativ weitreichende Erweiterungsmöglichkeiten.

Alle oben genannten Ansätze erlauben Erweiterungen allerdings nur innerhalb bestimmter Grenzen, die durch die syntaktische Grundstruktur der Sprache vorgegeben sind. Ausgehend von Ausdrücken als syntaktisches Grundmuster, die ganz allgemein aus Operanden und Operatoren aufgebaut sind, lassen sich jedoch Sprachen konzipieren, die nahezu beliebig syntaktisch erweiterbar sind:

1. Damit die Syntax von Ausdrücken beliebig erweiterbar ist, muss es nicht nur möglich sein, vordefinierte Operatorsymbole (wie z. B. in C++) zu überladen, sondern nach Belieben *neue Operatorsymbole* einführen zu können. Neben den bekannten Präfix-, Infix- und Postfixoperatoren sollten beliebige *Operatorkombinationen* (wie z. B. `...?...:...` in C/C++ oder `select...from... where...` in SQL) definierbar sein.

2. Da man Anweisungen wie z. B. `if...then...else...end` oder `repeat...until...` ebenfalls als Operatorkombinationen interpretieren kann, ist damit auch die Syntax von Anweisungen beliebig erweiterbar.

3. Da man auch Typen wie z. B. `int`, `int*` oder `int[10]` als Ausdrücke auffassen kann − `int` ist ein atomarer Typausdruck, `int*` die Anwendung des Postfixoperators `*` auf den Teilausdruck `int` und `int[10]` eine Anwendung der Operatorkombination `...[...]` auf die Operanden `int` und `10` −, lässt sich auch die Syntax von Typen beliebig erweitern.

4. Schließlich können auch Deklarationen wie z. B. `x : int` als Ausdrücke interpretiert werden (der linke Operand des Doppelpunkt-Operators ist ein Name, der rechte ein Typ), sodass auch deren Syntax prinzipiell beliebig erweiterbar ist.

Im Vortrag werden die oben skizzierten Ideen genauer erläutert und auf die Implementierung eines entsprechenden Parsers eingegangen.

Proposing Order-Sorted Algebra as Foundation for Declarative Programming*

Bernd Braßel and Rudolf Berghammer

Institut für Informatik,
Christian-Albrechts-Universität zu Kiel
{bbr,rub}@informatik.uni-kiel.de

Abstract. Order-sorted algebras provide a well developed and expressive framework for theoretical considerations about programming languages. We show that a number of declarative program paradigms fit well into this setting. For this aim we introduce a simple order, based on which strict functional programs, lazy functional programs and lazy functional logic programs with run-time choice or call-time choice can be modeled. An interesting feature of the presented approach is that the semantic differences between strictness/laziness on one hand and run-time/call-time choice on the other hand are only reflected in the type of variables allowed in the programs. Especially, this feature allows the admission of mixed programs without any change in the underlying theory.

1 Introduction

The presented work introduces an application of order-sorted algebra to functional and functional logic programming languages. We show that a simple order structure suffices to capture four basic semantical notions within a single formal framework. The result is a unified approach entailing the following advantages.

- By connecting functional logic programming to order-sorted algebra standard results from a well developed theory become applicable to these programming languages. This enables for example the use of standard transformations to obtain unsorted conditional rewrite systems and, further, standard unconditional rewrite systems for a given program.
- The presented framework contains an explicit specification of the laws of non-deterministic choice. This allows for a flexibility to examine alternative approaches to non-determinism by providing different specifications. This flexibility could be used, for example, to integrate approaches to encapsulated search [1, 5] or probabilistic programming languages in the future.
- Because of the unifying integration the presented framework allows the representation of programs with an arbitrary mix of the semantic notions involved. This could be useful to examine, e.g., the problems involved with approaches to encapsulated search. In the presented framework these problems can be expressed as a clash of intuitions about the sort of program variables.

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

– The high level of abstraction supported by order-sorted algebra makes the presented framework modularly extensible. Further specifications modeling, e.g., typed programs could be added without losing the achieved results.

The remaining introductory subsection will provide minimalistic examples to distinguish the four basic notions of semantics integrated in the presented framework. Section 2 develops the new representation of functional (logic) programs as equations over an order-sorted signature. Section 3 discusses the semantics of the resulting programs as well as the connection to former approaches. Especially, that section contains the discussion of how the four semantic notions exemplified in Section 1.1 are integrated within the framework. Section 4 concludes.

1.1 Four Basic Semantic Notions

The formal framework presented in this work integrates four different semantic notions by introducing a simple order of sorts. For functional programming languages the basic semantic notions are well known as call-by-value, call-by-name and call-by-need. With the extension to functional *logic* programming call-by-need is to be further distinguished into call-time choice and run-time choice [4]. The differences between the semantics become apparent in non right-linear program rules, i.e., when variables are duplicated on the right hand side of a rule. The following example is therefore a standard minimal one.

Example 1. Consider the following functional program rule.

```
double x = x+x
```

When evaluating the expression `double (0+1)`, call-by-value allows the following derivation only.

```
double (0+1) = double 1 = 1+1 = 2
```

With call-by-name, in contrast, the sub-expression `(0+1)` is copied and may therefore be evaluated more than once as in the following derivation.

```
double (0+1) = (0+1) + (0+1) = 1 + (0+1) = 1 + 1 = 2
```

Like call-by-name, call-by-need allows to apply the rule for `double` before fully evaluating the arguments. In contrast to call-by-name, however, not sub-expressions but *references to sub-expressions* are copied. This requires syntactic extensions to, e.g., graph rewriting or the introduction of `let`-expressions, as in the following derivation.

```
double (0+1) = let x=0+1 in x+x =
               let x=1   in x+x = 1+1 = 2
```

When considering functional programs only it is a well known and intuitive fact that call-by-need semantics corresponds to call-by-name. With the extension to non-deterministic choice, this correspondence is becoming less tight. With this extension there are two possibilities to interpret call-by-need derivations as illustrated in the next example.

Example 2. A minimalistic way to define a non-deterministic operation in functional logic programming languages is the following.

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

We consider the evaluation of the expression `double coin` and separate independent non-deterministic possibilities by the symbol "`|`". In a call-by-value derivation of the expression, the possible choices for `code` are done before applying the rule for `double`. The resulting semantics is called call-time choice.

```
double coin = double 0 = 0+0 = 0
            | double 1 = 1+1 = 2
```

In call-by-name the sub-expression `coin` is copied resulting in more non-deterministic combinations. The corresponding semantics is called run-time choice.

```
double coin = coin + coin = 0 + coin = 0 + 0 = 0
                                     | 0 + 1 = 1
                          | 1 + coin = 1 + 0 = 1
                                     | 1 + 1 = 2
```

Call-by-need is compatible with both variations of choice semantics. A derivation call-by-need with run-time choice could be represented as follows.

```
double coin = let x=coin in x+x
            = let x=0|1  in x+x = 0|1 + 0|1
            = ... = 0 | 1 | 1 | 2
```

And call-by-need with call-time choice is realized in the next derivation.

```
double coin = let x=coin in x+x
            = let x=0 in x+x = 0
            | let x=1 in x+x = 2
```

The four basic semantic notions considered in the following are therefore call-by-value, call-by-name, call-by-need with run-time choice and call-by-need with call-time choice.

2 Functional (Logic) Programs

We assume the reader to be familiar with the notions of order-sorted algebra [2] as well as that of a *relation*, a *partial order*, the *least* and *largest element* of a partial order, an *equivalence class* and the *transitive and symmetric closure* of a relation. We recall that the *connected components* of a partial order $(M, \leq)$ are the equivalence classes of the transitive and symmetric closure of $\leq$. A subset $D \subseteq M$ is called *directed* iff for all $a, b \in D$ there exists an element $c \in D$ with $a \leq c$ and $b \leq c$. A partial order is called *locally directed* iff each connected component is directed.

A functional logic program will be defined as a restricted set of equations over an order-sorted signature. Therefore, the first step will be to define the sorts used in a functional logic program together with an appropriate order on these sorts. After that we will define what kind of symbols and equations will be used to construct functional logic programs. The sorts for functional logic programs will be defined to distinguish four classes of terms: *(total) values*, *partial values*, *choices (over partial values)* and finally arbitrary *expressions*. The distinction between these classes is at the heart of our discussion of different semantic conceptions, such as call-by-name versus call-by-value and run-time choice versus call-time choice. The notions of order-sorted algebra will allow us to conveniently state that, e.g., values are a subsort of partial values.

Definition 1. *Let* $S_P = \{Val, PVal, Ch, Exp\}$. *The* set of program sorts $(S_P, \leq_P)$ *is the smallest partial order containing the chain* $Val < PVal < Ch < Exp$.

Obviously, the set of program sorts is locally directed. In functional and functional logic programs alike there is a basic partition of the symbols of a program signature into *constructor* and *operation* symbols. In addition to these user defined symbols we will use the constant symbol $\bot$ and the binary symbol $\sqcup$. Constructors are used to build values while the inclusion of $\bot$ yields a partial value. The symbol $\sqcup$ is used to construct terms of sort Ch and, finally, the appearance of an operation symbol in a term leads to its sort being Exp. This is the content of the following definition.

Definition 2. *Let* C *and* O *be two disjoint (standard) signatures with natural numbers as arity such that* $\bot \notin C_0 \cup O_0$ *and* $\sqcup \notin C_2 \cup O_2$. *Then the* program signature of $C \cup O$ *is the triple* $(S_P, \leq_P, C' \cup O' \cup L)$ *where*

$$\begin{aligned} C' := & \bigcup \{\{c : Val^n \to Val, c : PVal^n \to PVal, \\ & \qquad c : Ch^n \to Ch, c : Exp^n \to Exp\} \mid c \in C_n\} \\ O' := & \quad \{o : Exp^n \to Exp \mid c \in O_n\} \\ L := & \quad \{\bot : \varepsilon \to PVal, \sqcup : ChCh \to Ch, \sqcup : ExpExp \to Exp\} \end{aligned}$$

Here, for $p \in S_P$ *the notation* p^n *denotes a sequence* $w = p \ldots p$ *with* $|w| = n$.

The symbols in the subsets of C *are called constructor symbols and those in the subsets of* O *are called operation symbols, respectively. By* Σ_C *and* Σ_O *we denote the sub-signature of constructor and operation symbols, respectively.*

We will prove that program signatures are order sorted signatures and then give some examples of terms constructed from program signatures and their respective terms.

Proposition 1. *Program signatures are regular and coherent order-sorted signatures. Moreover, any rewrite system over a program signature is compatible.*

Proof. Let $\Sigma = (S, \leq, M)$ be a program signature. Then Σ satisfies the monotonicity condition for order-sorted signatures [2] since all ranks in Σ are of the

term t	$LS(t)$
$\texttt{Cons}(\texttt{True}, \texttt{Nil})$	Val
$\texttt{Cons}(\texttt{True}, \bot)$	$PVal$
$\texttt{Cons}(\texttt{True} \sqcup \texttt{False}, \bot)$	Ch
$\texttt{Cons}(\texttt{True}, \texttt{tail}(\texttt{Nil}))$	Exp

term t	$LS(t)$
$\texttt{Cons}(v, \texttt{Nil})$	Val
$\texttt{Cons}(\texttt{True}, p)$	$PVal$
$\texttt{Cons}(c, \bot)$	Ch
$\texttt{Cons}(\texttt{True}, e)$	Exp

Fig. 1. Different Terms and their Respective Sorts

form $\langle p^n, p\rangle$ and therefore $o \in M_{w,p} \cap M_{w',p'}$ and $w \leq w'$ always imply $p \leq p'$. Moreover, Σ is regular since for any s and natural number n the ranks

$$\{\langle w, p\rangle \mid s : w \to p \in \Sigma, |w| = n\}$$

form a chain. Finally, Σ is coherent by Definition 1. Rewrite systems over Σ are compatible since each n-ary symbol has a rank $\langle Exp^n, Exp\rangle$. Therefore, replacing a sub-term of program sort s with a term of program sort $s' > s$ again yields a well-sorted term in general, regardless of the rewrite rules. □

Example 3. Let $C := C_0 \cup C_2$ where $C_0 := \{\texttt{True}, \texttt{False}, \texttt{Nil}\}$ and $C_2 = \{\texttt{Cons}\}$ and let $O := O_1 := \{\texttt{tail}\}$. Then the left table of Figure 1 shows terms over the program signature of $C \cup O$ together with their least sort. By definition variables are also sorted. Let X be an according set of variables and $v \in X_{Val}$, $p \in X_{PVal}$, $c \in X_{Ch}$ and $e \in X_{Exp}$. The terms with variables and their least sorts are shown in the right table of Figure 1. Moreover, note that no order-sorted substitution can map the variable v to, e.g., the term $\texttt{tail}(\texttt{Nil})$ or to $\texttt{Nil} \sqcup \texttt{Cons}(\texttt{True}, \texttt{Nil})$.

We are now ready to give our notion of a program.

Definition 3. *Let C be a set of constructor and O a set of operations symbols, and Σ- be the program signature for $C \cup O$. Then a* program over Σ *is a pair (Σ, E) such that the following conditions hold.*

- *E is a set of* program rules. *There are two forms of rules: Σ-equations $l = r$ and Σ-*inequations *$l \sqsupseteq r$. In both forms we call l the left-hand side and r the right-hand side of the rule. For any rule l, r must satisfy the conditions*
  - *l is linear, i.e., each variable in $\mathcal{V}ar(l)$ occurs at most once in l*
  - *l is of the form $f(\overline{t_n})$ where $f \in O_n$ and for each t_i holds $t_i \in \mathcal{T}_{\Sigma_C(X)}$*
- *If $l = r$ is a rule in E there must be no other rule $l' = r'$ or $l' \sqsupseteq r'$ in E such that l, l' are unifiable, i.e., there is no substitution σ with $\sigma(l) = \sigma(l')$.*

3 Semantics

Definition 3 introduces two kinds of program rules: equations and inequations. Standard derivation rules define how new equations can be derived from given ones [2]. However, we still have to define how the inequations contained in a program should be treated in such a derivation. For this we follow the classic way to integrate inequations into an equational setting: by embedding orders into

(semi) lattices. The idea is that inequations of the form $l \sqsupseteq r$ will be treated as equations $l = l \sqcup r$. Not incidentally, the symbol $\sqcup$ will serve at the same time as an operator for non-deterministic choice. The benefit we get from the general setting of arbitrary equations over program signatures is that we can define the laws for $\sqcup$ within the same framework as the programs. Thus, the setting allows us to define not only program rules but also the semantical logic within the same framework. The additional equations fixing semantical properties will be called the *program logic* in the following.

3.1 Semi Lattices, Inequations and Program Logic

We start our semantic considerations by defining the notion of a semi lattice.

Definition 4. *A semi lattice is a structure $(S, \sqcup)$ satisfying the following laws for all elements of S.*

$$x \sqcup x = x \quad (\mathsf{idem}) \qquad x \sqcup y = y \sqcup x \quad (\mathsf{comm}) \qquad (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \quad (\mathsf{assoc})$$

A structure $(S, \sqcup, \bot)$ is called a semi lattice with identity iff $(S, \sqcup)$ is a semi lattice which additionally satisfies the following law for all elements of S.

$$x = \bot \sqcup x \quad (\mathsf{bot})$$

Semi lattices induce a partial order $(S, \sqsubseteq)$ defined by $x \sqsubseteq y$ iff $x \sqcup y = y$. Moreover, for a semi lattice with identity, $\bot$ is the least element of $(S, \sqsubseteq)$.

A *semi lattice homomorphism* is a function $h : S \to S'$ where S, S' are semi lattices such that $h(a \sqcup_S b) = h(a) \sqcup_{S'} h(b)$. As we will see below, constructors will be defined to be semi lattice homomorphisms. A *monotone function* can be likewise understood as a homomorphism on orders. For such a function $h : S \to S'$, we have that $a \sqsupseteq_S b$ implies $h(a) \sqsupseteq_{S'} h(b)$. Below we will define that all operations introduced in a program are monotone.

In accordance with Definition 4, inequations are treated as syntactic sugar for special equations. The reason to add inequations at all is to make our programs look similar to those used in existing functional logic languages. The main difference now is that we require the use of $l \sqsupseteq r$ for overlapping rules instead of also using the symbol $=$. Actually, we think that adding such a requirement to existing languages could improve the programming practice. With such a requirement, the program from Example 2

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

would be rejected by the compiler, since the left-hand sides of the rules are trivially unifiable. Rather, the program should be written as

```
coin ⊒ 0
coin ⊒ 1
```

and by stating it this way, we would have more confidence that the programmer knows "what he is doing". In addition we will not induce the feeling that the equation `True`=`False` should be derivable from the program rules.

Like a program, a program logic is a set of equations over a program signature. Unlike the program, however, the program logic is not restricted syntactically. This allows us to add the equations fixing the laws of non-determinism.

Definition 5. *Let P be a program over a program signature Σ with operations O and constructors C. The set of Σ-equations $PL(P)$ contains the equations of a semi lattice with identity (Definition 4) together with the following equations, where the variables $x, y, z, \overline{x_n}, \overline{y_m}$ are of sort Exp.*

$$\{f(\overline{x_n}, x \sqcup y, \overline{y_m}) \sqsupseteq f(\overline{x_n}, x, \overline{y_m}) \mid f \in O_{n+m+1}\} \quad \text{(mon)}$$
$$\{c(\overline{x_n}, x \sqcup y, \overline{y_m}) = c(\overline{x_n}, x, \overline{y_m}) \sqcup c(\overline{x_n}, y, \overline{y_m}) \mid c \in C_{n+m+1}\} \quad \text{(hom)}$$

3.2 Program Classification

The definition of program sorts allows us to define different classes of programs. The distinction concerns the variables used within the program. If, for instance, all variables have to be of sort *Val* then every argument of any function has to be a constructor term. Obviously, this corresponds to the arguments being fully evaluated. In the other extreme if variables are of sort *Exp* the resulting program will be call-by-name. The two intermediate possibilities classify the different approaches to non-determinism in programs with call-by-need semantics.

Definition 6. *When all the variables contained in program rules are*

of sort	*the program obeys*
Val	*call-by-value (with call-time choice)*
PVal	*call-by-need with call-time choice*
Ch	*call-by-need with run-time choice*
Exp	*call-by-name (with run-time choice)*

A program containing variables of different sorts will be called mixed. *In addition there is the classification into functional and functional logic programs. A program is called* functional *iff all program rules are of the form $l = r$, do not contain the symbol $\sqcup$ and satisfy the proposition $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A program which is not functional is called a* functional logic program.

In the remainder of this section we will exemplify the different program classes by revisiting the introductory example from Section 1.1. After that we will strengthen our result by showing that functional logic programs with call-by-need and call-time-choice correspond in our framework to the classical modeling of this class [3].

Example 4. Recall the program in Example 1.

```
double x = x+x
```

First we consider `x` to be of sort *Exp*. Then mapping `x` to `0+1` yields a valid substitution and therefore we can derive from the equation `double x = x+x` and `0+1=0+1` that `double (0+1) = (0+1) + (0+1)` by the substitutivity rule [2]. This yields the validity of the corresponding equational deduction from Example 1 (assuming suitable equations for addition).

```
double (0+1) = (0+1) + (0+1) = 1 + (0+1) = 1+1 = 2
```

When `x` is of sort *Val*, in contrast, `0+1` is not a valid substitution for `x`. Therefore, the above deduction is not possible. The only derivation to yield a constructor value is the call-by-value deduction of Example 1.

```
double (0+1) = double 1 = 1+1 = 2
```

That program variables of sort *PVal* yield a call-by-need semantics is less obvious. Indeed, the only valid deduction to a value in such a program corresponds to the call-by-value one above. Therefore we consider an example for which call-by-value and call-by-need indeed have different semantics. For this we consider `Z` and `S` the symbols to construct Peano numbers.

```
loop = loop
isZero (S x) = True
```

For this program the expression `isZero (S loop)` is not greater or equal than any constructor value when `x` is of sort *Val*. If, however, the sort of `x` is *PVal*, then we may substitute `x` by $\bot$ and get the following deduction.

```
isZero (S loop) = isZero (S (⊥ ⊔ loop))      (bot)
                = isZero (S ⊥ ⊔ S loop)       (hom)
                ⊒ isZero (S ⊥)               (mon)
                = True
```

The above considerations make it is easy to see that programs with variables of sort *Val* feature call-time and those of sort *Exp* feature run-time choice. Therefore, the further classification will consider call-by-need programs, only.

Example 5. Reconsider the program from Example 2 (in the new syntax).

```
coin ⊒ 0
coin ⊒ 1
```

When the variable `x` in the definition of `double` is of sort *Ch*, mapping it to `0⊔1` yields a valid substitution. Therefore we get the following deduction.

```
 double coin = double (0 ⊔ coin)
             = double (0 ⊔ (1 ⊔ coin))
             = double ((0 ⊔ 1) ⊔ coin)      (assoc)
             ⊒ double (0 ⊔ 1)               (mon)    (∗)
             = (0 ⊔ 1) + (0 ⊔ 1)
             ⊒ 0 + (0 ⊔ 1) ⊔ 1 + (0 ⊔ 1)    (mon)
             ⊒ 0+0 ⊔ 0+1 ⊔ 1+0 ⊔ 1+1         (mon)×2
          ... = 0 ⊔ 1 ⊔ 2                   (idem)
```

When `x` is of sort $PVal$, we can likewise proceed to the term `double (0⊔1)` in the line marked $(*)$. But then the above deduction is no longer valid as `x` may not be substituted with `0⊔1`. Instead we can only derive values in the following way.

```
double (0⊔1) = double 0 ⊔ double 1 ⊔ double (0⊔1)   (mon)×2
             ⊒ double 0 ⊔ double 1                   def ⊒
              = 0+0 ⊔ 1+1
              = 0 ⊔ 2
```

The discussion above employed the general assumption of functional logic programming that one is only interested in values, i.e., constructor terms which are terms of sort $PVal$ in our setting. In general, the terms considered as values should match the maximal sort of the variables allowed in the program. A program employing variables of sort $PVal$, for instance, should consider the set T_{PVal} for values whereas a call-by-value program should employ T_{Val}. This is the content of the following definition.

Definition 7. *Let P be a Σ-program and s a program sort. Then the s-semantics of a given Σ-term e, denoted by $[\![e]\!]_s$, is the set $[\![e]\!]_s = \{v \in T_s \mid e \sqsupseteq v\}$.*

In order to lift this definition to an algebraic semantics one would need to use standard constructs to obtain a complete lattice from the lattice defined by the Σ-terms. Well known constructions are completion by ideals or Dedekind-MacNeille completion [6, Section 2.2]. We omit the according development due to space constraints. Also due to space constraints we consider the above observations as sufficient to demonstrate the correspondences for call-by-value. As call-by-need with call-time choice is a more subtle concept we show that programs with variable sort $PVal$ indeed obey this semantics. For this we prove correspondence with the classical setting [3] in the next subsection. Regarding the two remaining semantics employing run-time choice we think that our approach is indeed novel as detailed in the conclusion.

4 Conclusion

We have applied order-sorted algebra to define a semantics for functional and functional-logic programs. It is based on a linearely ordered set of four sorts, the representation of programs as equations over an order-sorted signature and the sorts of variables allowed in a program. As main result we have modeled four import semantic notions in a uniform and succinct manner: call-by-value, call-by-name and call-by-need with run-time choice and call-time choice.

References

1. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

2. Joseph A. Goguen and José Meseguer. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.
3. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
4. H. Hußmann. *Nondeterministic Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
5. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
6. Ralph McKenzie, George McNulty, and Walter Taylor. *Algebras, Lattices, Varieties*. Wadsworth and Brooks, Monterey, California, 1987.

Fast and Accurate Strong Termination Analysis with an Application to Partial Evaluation*

Michael Leuschel[1], Salvador Tamarit[2], and Germán Vidal[2]

[1] Institut für Informatik, Universität Düsseldorf, D-40225, Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de
[2] DSIC, Technical University of Valencia, E-46022, Valencia, Spain
{stamarit,gvidal}@dsic.upv.es

Analysing the termination of logic programs is a challenging problem that has attracted a lot of interest (see, e.g., [2, 3, 7, 8] and references therein). However, *strong* termination analysis (i.e., termination for any selection rule) has received little attention, a notable exception being the work by Bezem [1].

A logic program strongly terminates if it terminates for any selection rule. Clearly, considering a particular selection rule—like Prolog's leftmost selection rule—allows one to prove more goals terminating. In contrast, a strong termination analysis gives valuable information for those applications in which the selection rule cannot be fixed in advance (e.g., partial evaluation, dynamic selection rules, parallel execution).

In this paper, we introduce a fast and accurate size-change analysis [4] that can be used to infer conditions for both strong termination and quasi-termination of logic programs. In the paper we also provide several ways to increase the accuracy of the analysis without sacrificing scalability. In the experimental evaluation, we show that the new algorithm is up to three orders of magnitude faster than the previous implementation, meaning that we can efficiently deal with realistic programs exceeding 25,000 lines of Prolog.

One major application of this algorithm is for offline partial evaluation of large programs. In the experimental evaluation we show that, with our new algorithm, we can now deal with realistic interpreters, such as the interpreter for the full B specification language from [6]. Together with the selective use of hints [5], we have obtained both a scalable and an effective partial evaluation procedure.

The full version of the paper will appear in the proceedings of WFLP 2009, Springer-Verlag.

References

1. M. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.
2. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
3. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.
4. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28:81–92, 2001.

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant GVPRE/2008/001, by the *UPV* (Programs PAID-05-08 and PAID-06-08), and DAAD PPP D/06/12830 together with *Acción Integrada* HA2006-0008.

5. M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Proc. of LOPSTR'08*, pages 119–134. Springer LNCS 5438, 2009.
6. M. Leuschel and M. Butler. ProB: A model checker for B. *Proceedings FME 2003*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
7. Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Proving Termination for Logic Programs by the Query-Mapping Pairs Approach. In *Program Development in Computational Logic*, pages 453–498, 2004.
8. S. Verbaeten, K. Sagonas, and D. De Schreye. Termination Proofs for Logic Programs with Tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, 2001.

Funktionaler Fauler Nichtdeterminismus

Sebastian Fischer

Institut für Informatik
Christian-Albrechts Universität zu Kiel

Logisch-funktionale Programmierung zeigt, dass die Kombination von bedarfsgesteuerter mit nichtdeterministischer Auswertung weitreichende Vorteile zum Lösen von Suchproblemen hat. Diese Vorteile erschließen sich rein funktionalen Programmierern nur bedingt, da die exisitierende Unterstützung für bedarfsgesteuerte Auswertung und Nichtdeterminismus in funktionalen Sprachen nicht einfach kombiniert werden kann.

Ich stelle eine gemeinsame Arbeit mit Oleg Kiselyov und Chung-chieh Shan vor, in der wir eine—in die funktionale Programmiersprache Haskell eingebette—Sprache für nichtdeterministische Berechnungen mit bedarfsgesteuerter Auswertung entwickeln. Neben einer gleichungsbasierten Spezifikation des für logisch-funktionale Sprachen entwickelten Begriffes *Call-Time Choice*, der das Verhalten der genannten Kombination klärt, stelle ich Techniken vor, die es erlauben, diese Spezifikation effizient zu implementieren. Unsere Implementierung hat ein vergleichsweise hohes Abstraktionsniveau und erlaubt zum Beispiel die Integration beliebiger Suchstrategien. Sie bietet—neben ihrer Verwendbarkeit als rein funktionale Bibliothek—eine flexible Basis für die Übersetzung logisch-funktionaler Programme nach Haskell. Erste Vergleiche mit exisitierenden Übersetzern für die logisch-funktionale Programmiersprache Curry sind vielversprechend.