UiO : Department of Informatics
University of Oslo

Tagungsband des 35ten Jahrestreffens der GI-Fachgruppe “Programmiersprachen und Rechenkonzepte”

Proc. of the 35th Annual Meeting of the GI Working Group “Programming Languages and Computing Concepts”

Jens Knoop, Martin Steffen and
Baltasar Trancón y Widemann (Eds.)

Research report 482, August 2018

ISBN 978-82-7368-447-9
ISSN 0806-3036

35tes Jahrestreffen der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"

Bad Honnef, 2.–4. Mai 2018
Tagungsband

Vorwort

Seit 1984 veranstaltet die *GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"* regelmäßig im Frühjahr ein Arbeitstreffen im Physikzentrum Bad Honnef. Das Treffen dient in erster Linie dem gegenseitigen Kennenlernen, dem Erfahrungsaustausch, der Diskussion und der Vertiefung gegenseitiger Kontakte. In diesem jährlichen Forum werden Vorträge und Demonstrationen zu sowohl bereits abgeschlossenen als auch noch laufenden Arbeiten vorgestellt, unter anderem 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

Dieser Technische Bericht ist dem Treffen des Jahres 2018 gewidmet, dem 35ten Workshop aus der Reihe, welcher vom 2. bis 4. Mai stattfand. Daneben wurden einzelne Beiträge aus den Jahren 2017 und 2016 aufgenommen. Diese Jahr gab es speziell Sitzungen mit Vorträgen zu den Themen Tests & Verifikation, Java, Domänenspezifisches, Programmstrukturen und Curry. Neben der traditionellen Wanderung in die Umgebung, in diesem Jahr auf den Drachenfels, war der kulturelle Höhepunkt 2018 die Uraufführung der *8ten Symphonie (für synthetisches Orchester)*, op. 42a von Markus Lepper, durch Abspielen der in Netz veröffentlichten Realisation.

Allen Teilnehmenden gilt Dank, daß sie durch ihre Vorträge, Papiere und Diskussion den jährlichen Workshop zu einem interessanten und anregenden Ereignis machen. Besonderer Dank auch den Mitarbeitern und Mitarbeiterinnen des Physikzentrums Bad Honnef, die, wie immer, durch ihre umfassende Betreuung für eine angenehme und anregende Atmosphäre gesorgt haben.

Jens Knoop, Martin Steffen und Baltasar Trancón y Widemann, Juli 2018

Foreword

The *GI Working Group"Programming Languages and Computing Concepts"* arranges since 1984 each spring a workshop in the the "Physikzentrum Bad Honnef" (a convention centre of the German Physical Society). The meeting serves to foster exchange of ideas and experience, discussions, and to build up new connections or cultivate existent ones. The technical program of the workshop offers a forum for finished work as well as work-in-progress, in the form of presentations and demonstrations. The topics include items from the following (non-exclusive) list:

- languages, language paradigms
- correctness of design and implementation
- tools
- software- and hardware architectures
- specification and design
- validation and verification
- implementation and integration
- safety and security
- embedded systems
- hardware-close programming

This report collects contributions from the meeting in 2018, the 35th workshop in the series, having taken place May 2nd to 4th. Besides that, a few papers presented 2017 and 2016 are additionally included in these proceedings. This year, there were presentation sessions covering the topics testing & verification, Java, domain-specific languages, programm structures, and Curry. Besides the traditional hike into the surroundings of Bad Honnef, this year climbing the Drachenfels or "Dragon's Stone", a specific cultural highlight of 2018 was the premiere of the *8th Symphony (for synthetic orchestra),* op. 42a, by Markus Lepper, by replaying the version available publically on the Internet.

Thanks to all participants for their presentations, papers, and discussions, that make the annual workshop an interesting and inspiring event. Special thanks also and in particular to the staff and employees at the hosting convention centre, the *Physikzentrums Bad Honnef*, who took care of the arrangment and, with their support and assistance, contributed, as always, to an inspiring and enjoyable atmosphere.

Jens Knoop, Martin Steffen, and Baltasar Trancón y Widemann, Juli 2018

Inhaltsverzeichnis

Generating Tests for Regular Expression Engines

Gabriel Radanne and Peter Thiemann

University of Freiburg, Germany
{radanne,thiemann}@informatik.uni-freiburg.de

Abstract. Regular expressions are part of every programmer's toolbox. They are used for a wide variety of language-related tasks and there are many algorithms for manipulating them. In particular, matching algorithms that detect whether a word belongs to the language described by a regular expression are well explored, yet new algorithms appear frequently. However, there is no satisfactory methodology for testing such matchers.
We propose a testing methodology which is based on generating positive as well as negative examples of words in the language. To this end, we present a new algorithm to generate the language described by a generalized regular expression with intersection and complement operators. The complement operator allows us to generate both positive and negative example words from a given regular expression. We implement our generator in Haskell and OCaml and show that its performance is more than adequate for testing.

1 Introduction

Regular languages are everywhere. Due to their apparent simplicity and their concise representability in the form of regular expressions, regular languages are used for many text processing applications, reaching from text editors [6] to extracting data from web pages.

Consequently, there are many algorithms and libraries that implement parsing for regular expressions. Some of them are based on Thompson's translation from regular expressions to nondeterministic finite automata and then apply the powerset construction to obtain a deterministic automaton. Others are based on Brzozowski's derivatives [2] and map a regular expression directly to a deterministic automaton. Antimirov's partial derivatives [1] yield another transformation into a nondeterministic automaton. An implementation based on Glushkov automata has been proposed [4] with decent performance. Russ Cox's webpage gives a good overview of efficient implementations of regular expression search. It includes a discussion of his implementation of Google's RE2 [3].

Some of the algorithms for regular expression matching are rather intricate and the natural question arises how to test these algorithms. While there online repositories with reams of real life regular expressions [5], there are no satisfactory generators for test inputs. It is not too hard to come up with generators for strings that match a given regular expression, but that is only one side of

the medal. On the other hand, the algorithm should reject strings that do not match the regular expression, so it is equally important to come up with strings that do **not** match.

This work presents generator algorithms for extended regular expressions that contain intersection and complement beyond the regular operators. The presence of the complement operator enables the algorithms to generate strings that certainly do not match a given (extended) regular expression.

Our implementations are useful in practice. They are guaranteed to be productive and produce total outputs. That is, a user can gauge the string size as well as the number of generated strings without risking partiality.

Even though the implementations are not tuned for efficiency they generate languages at a rate between $1.3 \cdot 10^3$ and $1.4 \cdot 10^6$ strings per second, for Haskell, and up to $3.6 \cdot 10^6$ strings per second, for OCaml. The generation rate depends on the density of the language.

- Web app available at `https://regex-generate.github.io/regenerate/`
- OCaml code available at `https://github.com/regex-generate/regenerate`
- Haskell code available at `https://github.com/peterthiemann/re-generate`

References

1. Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
2. Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
3. Russ Cox. Regular expression matching in the wild. `https://swtch.com/~rsc/regexp/regexp3.html`, March 2010.
4. Sebastian Fischer, Frank Huch, and Thomas Wilke. A play on regular expressions: functional pearl. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 357–368. ACM, 2010.
5. Regular expression library. `http://www.regexlib.com/`.
6. Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

Schritte zu einer zertifizierten Taint-Analyse

Thomas S. Heinze

Institut für Informatik
Friedrich-Schiller-Universität Jena
t.heinze@uni-jena.de

Zusammenfassung. In diesem Vortrag wird eine statische Datenflussanalyse für das Problem des Taint-Checking vorgestellt. Die Analyse beruht auf dem Verfahren der Zeigeranalyse und wurde mit Hilfe des Beweisassistenten Coq auf einer einfachen While-Sprache formal definiert, so dass der Nachweis der Analysekorrektheit und -terminierung als maschinenprüfbarer Beweis zur Verfügung steht. Der Beitrag kann somit als ein Schritt zu einer zertifizierten Taint-Analyse verstanden werden. Daneben wird ein Ausblick auf aktuelle Arbeiten und Herausforderungen für die Anwendung der Analyse auf realistische Programme gegeben.

Towards automatic verification of Matlab State Charts — Transforming state charts with requirements into SMT-LIB*

Marcellus Siegburg

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

Abstract

The joint project HPSV aims at improving intended behaviour of software. Among testing, verification is one of the used methods. A highly parallel SMT solver is being developed in the context of the project. That is why, it is desired to generate first-order logic formulas in SMT-LIB format specifying verification problems.

In the context of model driven development Matlab Simulink is a commonly chosen developing tool. There is a language for requirements called MARS which allows for describing the behaviour of a model. Requirements of Matlab State Charts, which are a subset of Matlab Simulink, shall be verified.

Developed as language for property specifications, the purpose of MARS is mainly the automatic generation of test cases. But every requirement can also be used as a proof obligation statement. The state chart can easily be transformed into first-order logic using its denotational semantics. The formalisation of MARS is the foundation for translating requirements into first order logic. A prototypical implementation of a translation of Matlab State Charts and accompanying MARS requirements allows for automatic generation of SMT-LIB formulas.

*This work is supported by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH15006B, joint project „Hochparallele Software-Verifikation nebenläufiger Anwendungen in der Automobilindustrie“ (HPSV)

Optimization of the Java type unification

Martin Plümicke

Duale Hochschule Baden-Württemberg, Campus Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract. The global type inference problem in Java can be reduced to the type Java unification problem. In a former article we have presented a Java type unification algorithm. The algorithm works fine for small examples. But for larger examples the used memory and the runtime is not acceptable.
In this paper we give a short summary of our algorithm and present some optimizations. We present two kinds of optimizations. The first kind do not reduce the solutions. The second kind restricts the solutions which means that we have guarantee that the algorithm determine the relevant solutions.
We present as a benchmark the matrix multiplication. The types cannot be determined with the original algorithm because of memory and runtime lack. But with the optimizations the types of the matrix multiplication can be determined with ordinary computer in acceptable runtime.

1 Introduction

Global type inference enables to write Java programs without any type annotation. The type inference algorithm introduces the types during compile time. This allows to omit the types without loosing the static type property. Such an algorithm is developed in the Java-TX project [Plü15,PS17] .
The type inference problem can be reduced to a type unification problem. One of the major problems is the runtime of the type unification algorithm. The type unification problem is NP-hard. The algorithm described in [Plü09,SP18] iterates all elements of a cartesian product of all possible type annotations. This leads to an exponentiell runtime in comparison to the number of omitted types. There are examples, where the algorithm terminates after hours.
In the worst-case this is probably unchangeable, but in many cases the runtime can be reduced enormously.
In this paper we present after a brief summary of the main part of the type unification algorithm in Section 2 and a large example in Section 3, which we use as a benchmark, different optimizations in Section 4:

Early filtering of not solved forms: After the reduction steps the sets which includes error-pairs are erased.

Reduced evaluation of cartesian product elements: This leads to enormous reduction of memory use.
Evaluation of only one inequation each recursion iteration: This reduces the number of incorrect pairs.
Considering dependent mappings of error-pairs: Many branches in the backtracking, which results in errors, are not considered.
Considering only relevant solutions: This leads to an enormous reduction of the number of calculated solutions.

We close with a summary and an outlook.

2 The type unification algorithm

In this section we give a brief summary od the type unification algorithm. For more details we refer to the [Plü09,SP18].
In the following let $\theta\ \theta'$, θ_i are Java types and $\leq^*$ the subtyping relation of Java types.
The type unification problem is given as: For a set of type term pairs $\{ (\theta_1, \theta'_1), , \ldots, (\theta_n, \theta'_n) \}$ a substitution σ is demanded, such that

$$\sigma(\theta_1) \leq^* \sigma(\theta'_1), \ldots, \sigma(\theta_n) \leq^* \sigma(\theta'_n)$$

Some more notations are used in the following:

- $\theta\ \theta'$, θ_i are Java types.
- The subtyping relation is denoted by $\leq^*$.
- $\theta \doteq \theta'$ means that θ and θ' should be unified such that $\theta = \theta'$.
- $\theta \lessdot \theta'$ means that θ and θ' should be unified such that $\theta \leq^* \theta'$.
- $\theta \lessdot_? \theta'$ means that θ and θ' should be unified such that θ and θ' are in subtype relation in argument position.
- We write `A<?B>` for `A<? extends B>`.
- We write `C<?D>` for `C<? super D>`.
- A set of equations is in *solved form*, if all pairs $a_i \doteq \theta_i$ where a_i are pairwise different type variables and for all i, j a_i does not occur in θ_j and all pairs $a \lessdot b$, $a \lessdot_? b$ consists only of type variables.

The type unification algorithm is given as:

Input: Set of equations $Eq = \{ \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \}$
Precondition: θ_i, θ'_i are Java types.
Output: Set of all general type unifiers $Uni = \{ \sigma_1, \ldots, \sigma_m \}$
Postcondition: For all $1 \leqslant j \leqslant m$ and for all $1 \leqslant i \leqslant n$ holds $(\sigma_j(\theta_i) \leq^* \sigma_j(\theta'_i))$.

1. Repeated application of reduction rules, such that either pairs consists of at least one type variable or the pairs are unsolvable.
2. For each pair $a \lessdot \theta$ and $a \lessdot_? \theta$ for all subtypes $\overline{\theta}$ of θ pairs $a \doteq \overline{\theta}$ are built and for each pair $\theta \lessdot a$ and $\theta \lessdot_? a$ for all supertypes θ' of θ pairs $a \doteq \theta'$ are built.

3. The cartesian product of the sets from step 2 is built.
4. Application of the subst rule which replaces for all pairs $a \doteq \theta$ in all types all occurring a by θ.
5. For all changed sets of type terms start again with step 1.
6. Filter all results in solved form (either pairs of the form $a \doteq \theta$, $a \lessdot b$, or $a \lessdot_? b$, where a and b are type variables) and unite them.

Let us consider an example.

Example 1. In this example we use the standard Java types `Number`, `Integer`, `Stack`, `Vector`, `AbstractList`, and `List`. It holds $\texttt{Integer} \leq^* \texttt{Number}$ and $\texttt{Stack<a>} \leq^* \texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}$.
As a start configuration we use

$$\{\, (\texttt{Stack<a>} \lessdot \texttt{Vector<}{}_?\texttt{Number>}),\ (\texttt{AbstractList<Integer>} \lessdot \texttt{List<a>}) \,\}.$$

In the first step the reduction rules are applied twice:

$$\{\, \texttt{a} \lessdot_? {}_?\texttt{Number},\ \texttt{Integer} \lessdot_? \texttt{a} \,\}$$

With the second step we receive in step three:

$$\begin{aligned}\{\ &\{\, \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\ &\{\, \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\ &\{\, \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\ &\{\, \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\ &\{\, \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\ &\{\, \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\ &\{\, \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\ &\{\, \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}\{\, \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \,\}\ \}\end{aligned}$$

In the forth step the rule *subst* is applied:

$$\begin{aligned}\{\ &\{\, \texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq \texttt{Integer} \,\}, \underline{\{\, {}_?\texttt{Number} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \,\}},\\ &\{\, {}_?\texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, {}^?\texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\ &\{\, \texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, {}_?\texttt{Number} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\ &\{\, {}_?\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, {}^?\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\ &\{\, \texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, {}_?\texttt{Number} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\ &\underline{\{\, {}_?\texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}}, \{\, {}^?\texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\ &\underline{\{\, \texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \,\}}, \{\, {}_?\texttt{Number} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\ &\{\, {}_?\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}\{\, {}^?\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \,\}\ \}\end{aligned}$$

The underlined sets of type term pairs lead to unifiers.
Now we have to continue with the first step (step 5). With the application of reduction rule (step 1) and step 6, we get three general type unifiers:

$$\{\, \{\, a \mapsto {}_?\texttt{Number} \,\}, \{\, a \mapsto {}_?\texttt{Integer} \,\}, \{\, a \mapsto \texttt{Integer} \,\}\ \}.$$

3 Benchmark example

Let us have a lock at another example which we use as benchmark for our optimizations. The class `Matrix` (Fig. 1) with the method `mul` implements the

```
class Matrix extends Vector<Vector<Integer>> {
    mul(m) {
        var ret = new Matrix();
        var i = 0;
        while(i < size()) {
            var v1 = this.elementAt(i);
            var v2 = new Vector<Integer>();
            var j = 0;
            while(j < v1.size()) {
                var erg = 0;
                var k = 0;
                while(k < v1.size()) {
                    erg = erg + v1.elementAt(k)
                          * m.elementAt(k).elementAt(j);
                    k++;
                }
                v2.addElement(new Integer(erg));
                j++;
            }
            ret.addElement(v2);
            i++; }
        return ret;
    }
}
```

Fig. 1. Matrix multiplication

[(`Matrix` ⩽ E), (O ⩽ `java.lang.Integer`), (`java.lang.Boolean` ≐ W),
(W ≐ `java.lang.Boolean`), (S ⩽ `java.lang.Integer`),
(`java.lang.Integer` ⩽ `java.lang.Integer`), (Z ⩽ `Matrix`), (V ⩽ U), (Z ≐ `Matrix`),
(`Matrix` ⩽ `Matrix`), (F ⩽ `java.lang.Integer`), (T ⩽ S), (J ⩽ `java.lang.Integer`),
(AVJ ≐ M), (P ⩽ O), (R ⩽ `java.lang.Integer`), (`java.lang.Integer` ≐ AE),
(`java.util.Vector<java.lang.Integer>` ⩽ N),
(`java.util.Vector<java.lang.Integer>`⩽`java.util.Vector<java.lang.Integer>`),
(AA ⩽ `java.lang.Integer`), (`java.lang.Integer` ≐ AD), (AXP ≐ AB),
(U ≐ AF), (D ⩽ `java.util.Vector<AXP>`), (L ⩽ `java.util.Vector<AVJ>`),
(L ≐ `Matrix`), (U ⩽ `java.lang.Integer`), (AE ⩽ S), (`java.lang.Boolean` ≐ Q),
(Q ≐ `java.lang.Boolean`), (void ≐ AI), (M ⩽ K), (`java.lang.Integer` ⩽ AXR),
(void ≐ AM), (F ≐ AN), (AD ⩽ `java.lang.Integer`), (N ⩽ AXT),
(E ⩽ `java.util.Vector<AXU>`), (E ⩽ C), (N ⩽ `java.util.Vector<AXS>`),
(X ⩽ `java.lang.Integer`), (Y ⩽ `Matrix`), (AB ⩽ `java.util.Vector<AXQ>`),
(AC ⩽ `java.lang.Integer`), (H ≐ `java.lang.Boolean`),
(`java.lang.Boolean` ≐ H), (O ≐ AJ), (G ⩽ F), (K ⩽ `java.util.Vector<AXO>`),
(AXQ ≐ AC), (Y ≐ `Matrix`), (AXO ≐ AA)]

Fig. 2. Type unification input from `Matrix`

matrices multiplication in Java-TX. The method `mul` has no declared argument type and no declared return type. The type inference algorithm shall determine them. The type inference problem is reduced to the type unification problem with the input given in Fig. 2.
We will use this example as a benchmark for the optimizations. Our measurement we have done with an eclipse Oxygen on Mac OS X with standard configurations. If we apply the algorithm to the input of Fig. 2 after many hours the program crashes with out of memory.

4 Optimizations

4.1 Early filtering of not solved forms

After step 1 pairs $\theta \lessdot \theta'$ or $\theta \lessdot_? \theta'$ where θ and θ' is no type variable would let to errors. All sets of pairs which consists such pairs are erased.

4.2 Reduced evaluation of cartesian product elements

In Fig. 3 the cartesian product of step 3 is visualized. It is obvious that the number of pair sets grows enormously. E.g. the number of pairs in Fig. 2 is about 50. If foreach pair average 10 solutions are given, there are 10^{50} sets.
We reduce the number of sets in memory, as we evaluate for each pair only one element at the same time. This means during the calculation of the unification algorithm only one set is in memory. If the set is unsolvable the set is erased. If it has a solution the solution is stored. In Fig. 4 we illustrate the optimization. This leads to an enormous reduction of used memory.

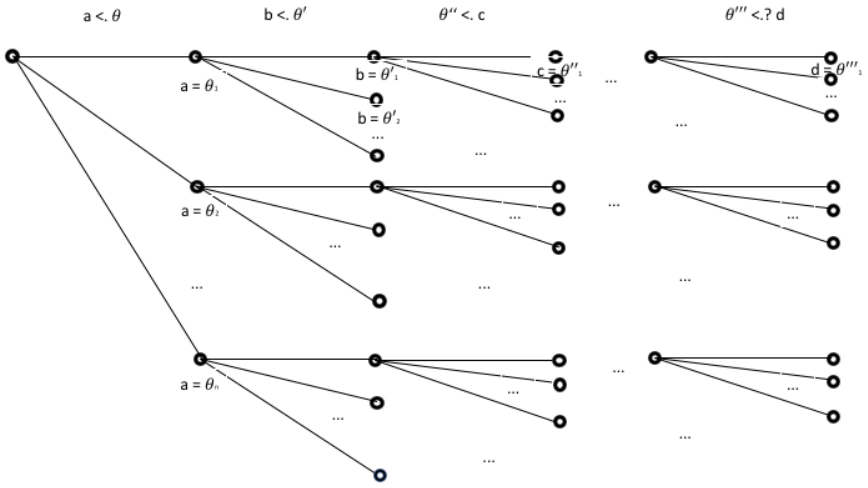


Fig. 3. Unreduced evaluation of step 2

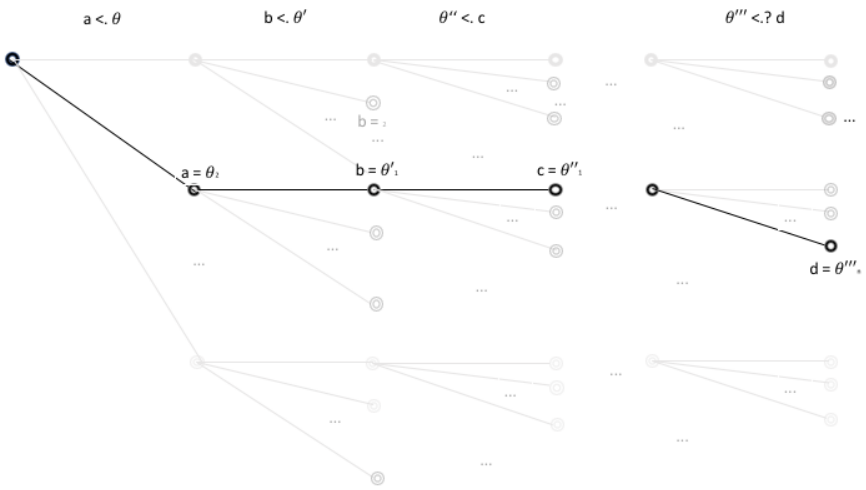


Fig. 4. Reduced evaluation in step 2

4.3 Evaluation of only one inequation each recursion iteration

We change step 2 of the algorithm such that only one element $a \lessdot \theta$ is evaluated to a set $\{ a \doteq \theta_1, \ldots, a \doteq \theta_n \}$. All other pairs remain untouched. They will be evaluated in a later recursion iteration. This means the cartesian product in step 3 has only n elements (number of solutions of $a \lessdot \theta$). It is visualized in Fig. 5.

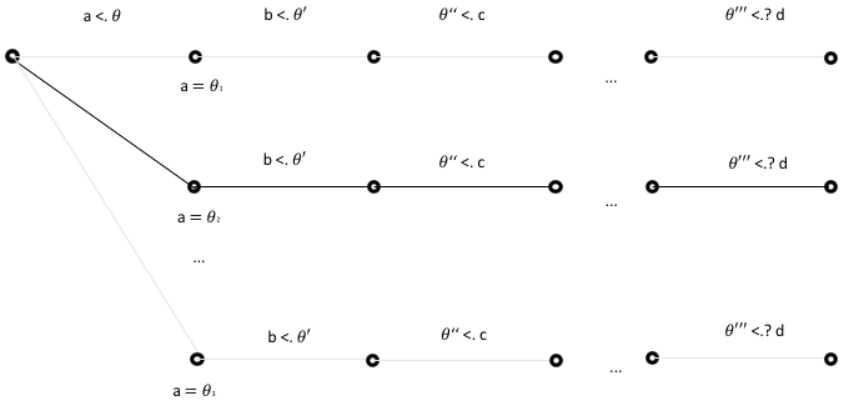


Fig. 5. Evaluation of only one inequation

This leads to a reduction of the of the generated elements, as for all pairs $a \doteq \theta_j$, which leads to errors, the respective element is removed after step 1 (cp. Section 4.1). This means that for this element for the remaining pairs no cartesian product is built.

4.4 Considering dependent substitutions of error-pairs

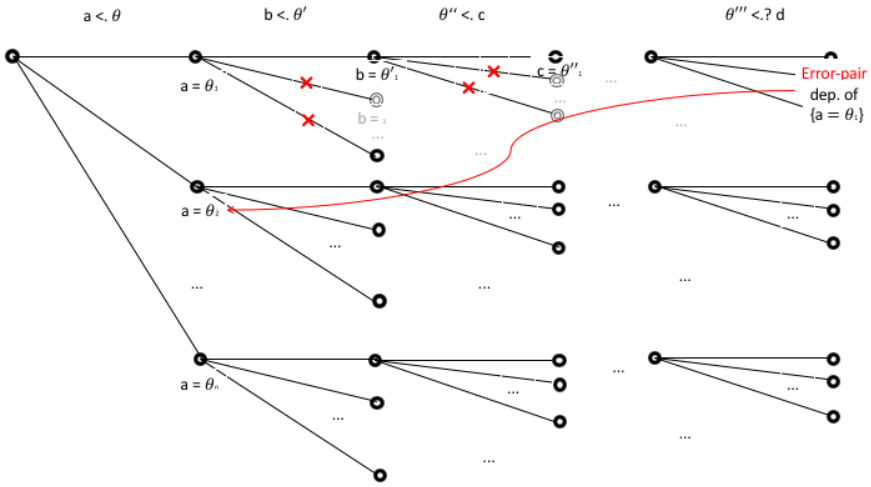


Fig. 6. Dependent substitutions

We change the data-structure of the pairs $\theta \lessdot \theta'$, such all substitutions which are applied to the respective pair are stored in the pairs.
If an error occur after step 1 this pair would lead to an error as long as no dependent substitution of the pair is changed. This allows to go back until a dependent substitution is changed. All backtracking steps on the way back need not to be considered. The optimization is visualized in Fig. 6.

These four optimizations (Section 4.1 - 4.4) reduce runtime and used memory without reducing the number of solutions. Nevertheless, all these optimizations do not lead to satisfying results of runtime and memory use.

4.5 Considering relevant solutions

The following two approaches determines normally only one and in some case a few solutions. This means that many solutions are not determined. We have guarantee that the *relevant* solutions are calculated.

One random solution The naive approach to get only one solution is to stop the backtracking if one solution is reached.
The runtime of this solution is very good. We get for the benchmark example a runtime of less than 10 seconds. The results are indeed correct but random and not unique.

Covariance and contravariance of method argument and return types Before we will present the idea of *relevant* solutions let us have a look at the definition of principal type in this type system. In [Plü07] we made a proposal for a principal type definition.
Damas and Milner [DM82] define for ML-like languages a principal type: *A type-scheme for a declaration is a* principal type-scheme*, if any other type-scheme for the declaration is a generic instance of it.*
A generalization to the Java type system could be: *An intersection type-scheme for a declaration is a* principal type-scheme*, if any (non–intersection) type-scheme for the declaration is a subtype of a generic instance of one element of the intersection type-scheme.*
This idea leads to the formal definition:

Definition 1. *An intersection type of a method* `m`

$$\texttt{m} : ((\theta_{1,1}, \ldots, \theta_{1,n}) \to \theta_1) \& \ldots \& ((\theta_{m,1}, \ldots, \theta_{m,n},) \to \theta_m)$$

is called principal *if for any correct type annotated method declaration*

$$\texttt{rty m(ty1 a1}, \ldots, \texttt{ty1 an) \{ ... \}}$$

there is an element $((\theta_{i,1}, \ldots, \theta_{i,n},) \to \theta_i)$ *of the intersection type and there is a substitution* σ*, such that*

$$\sigma(\theta_i) \leq^* \texttt{rty}, \texttt{ty1} \leq^* \sigma(\theta_{i,1}), \ldots, \texttt{tyn} \leq^* \sigma(\theta_{i,n}).$$

This means that the argument types are contravariant and the return types are covariant. Therefore we make the following assumptions:

- Determine maximal (wrt. the subtyping relation) correct types for the arguments
- Determine minimal (wrt. the subtyping relation) correct types for the results.
- For all other types take the first solution.

Fig. 7 visualize this approach. This approach leads to suitable results. For the matrix benchmark the types

```
Matrix mul(Vector<? extends Vector<? extends Integer>>) { ... }
```

are determined. The runtime of the matrix benchmark is lower that 20 seconds. For the matrix example indeed a principal type is determined. But there are some examples where not all elements of the principal intersection types are calculated.

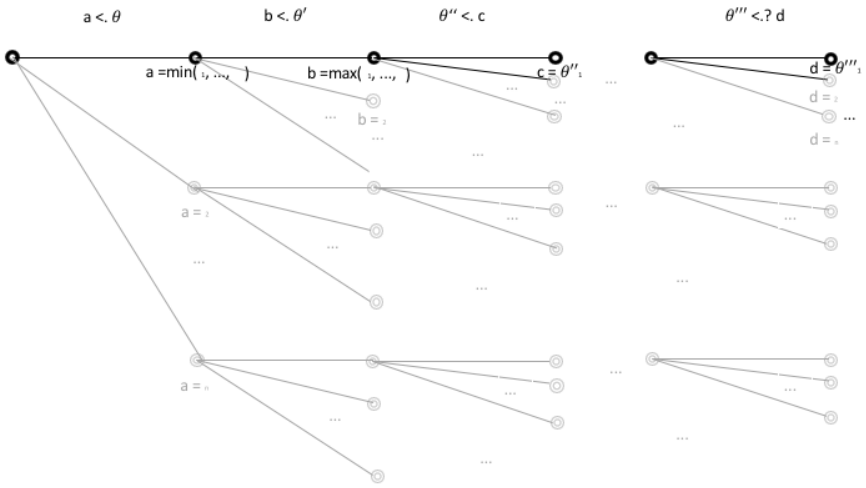


Fig. 7. Maximal and minimal Elements

Let us consider the next example

```
public class Lambda {

  m () {
   var lam1 = (x) -> {
        return x;
   };
   return lam1.apply(1);
}
```

The algorithm determine `Object` as the result type of `m`, although `Integer` would also be correct and it is more general. The reason is that first for `lam1` a type variables α is determined for the argument type and a type variables β is determined for the result type. Furthermore holds $\alpha \lessdot \beta$ and $\beta \lessdot \gamma$, where γ is the return type of `m`.
In `lam1.apply(1)` `Integer` $\lessdot \alpha$ is generated. As α is an argument type the maximum for α is determined. This means α is instantiated by `Object`. Therefore γ is instantiated by `Object`, too.

5 Summary and outlook

We have presented an optimization of our type unification algorithm. The algorithm determines a cartesian product of all possible typings. We reduce the number elements of the cartesian product as we remove wrong elements as early as possible. These optimization do not restrict the solutions. As we do not get

satisfying results by these optimizations, we restrict the solutions which are determined.
We considered the definition of a principal type which we gave in a former article. It is the goal to infer a principal type such that all other types can be derived from the inferred type. The principal type is the maximium of the argument types and the minimum of the result type of a function, as argument types are contravariant and result types are covariant. At the moment the algorithm determine in the most cases a principal type. Only in some cases not all types are determined. With our optimizations we can determine the types of matrix multiplication is acceptable runtime.
In future work we plan to optimize the algorithm again. At the moment many sets of sub- and supertypes are calculated often and some part-unification are determined often, too. We look forward to to get speed-ups by using hashtables. Furthermore we will change the optimized algorithm such that the algorithm determines in all cases principal types.

References

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

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

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

Plü15. Martin Plümicke. More type inference in Java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

PS17. Martin Plümicke and Andreas Stadelmeier. Introducing Scala-like function types into Java-TX. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 23–34, New York, NY, USA, 2017. ACM.

SP18. Florian Steurer and Martin Plümicke. Erweiterung und Neuimplementierung der Java Typunifikation. 2018. (in german).

Laufzeitoptimierung von Java Typinferenz

Andreas Stadelmeier

Duale Hochschule Baden-Württemberg, Campus Horb

1 Globale Typinferenz für Java

Globale Typinferenz für Java ermöglicht es Java vollständig ohne Typen zu schreiben. Zur Compilezeit setzt dann ein Typinferenzalgorithmus die fehlenden Typen in das Programm ein und führt den für Java üblichen statischen Typcheck durch. Das erlaubt dem Programmierer sämtliche Typen in seinem Programmcode auszulassen ohne dabei auf die Vorteile einer statisch typisierten objektorientierten Programmiersprache verzichten zu müssen.
Ein solcher Algorithmus wird im Zuge unseres JavaTX Projektes entwickelt.

2 Laufzeitverhalten

Eines der größten aktuellen Probleme ist die Laufzeit des Typinferenzalgorithmus. Denn Globale Typinferenz für Java ist NP-Schwer. Das lässt sich mittels einer Reduktion des SAT-Problems auf unseren Typinferenzalgorithmus zeigen. Der momentan vom Projekt verwendete Algorithmus iteriert über alle in Frage kommenden Typisierungen. Dadurch verhält sich dessen Laufzeit exponentiell zu der Anzahl an ausgelassenen Typen. So kann es schon bei kleineren Beispielen passieren, dass der Algorithmus erst nach mehreren Minuten terminiert. Im Normalfall sollte es allerdings nicht nötig sein sämtliche Typisierungen durchzuprobieren, so unsere Vermutung.

3 Laufzeitoptimierung

Die Idee ist es mithilfe von Verfahren aus dem Bereich des SAT-Solvings den Algorithmus zu beschleunigen. Meine Masterarbeit beschäftigt sich damit den Kern des Typinferenzalgorithmus in ein aussagenlogisches Problem umzuwandeln. Dieses kann anschließend von einem SAT-Solver gelöst werden.
Der Einfachheit halber programmiere ich den Algorithmus zunächst als Answer Set Program. Dieses kann ein Compiler wie clingo zum Teil in ein SAT Problem, also eine Aussagenlogische Formel, umwandeln und mittels eines SAT Solvers lösen. Dadurch kann möglicherweise bereits eine Laufzeitverbesserung erreicht werden, die mit der direkten Anwendung eines SAT Solvers vergleichbar ist.
Mein Vortrag stellt den aktuellen Stand meiner Arbeit vor und gibt zudem einen Ausblick auf zukünftige Erweiterungen, welche die Anwendung modernen SAT Solver ermöglichen.

Bytecode generation in Java-TX

Fayez Abu Alia

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
abualia@hb.dhbw-stuttgart.de

Abstract. Java-TX is an extension of Java. The main new features are global type inference and real function types for lambda expressions. The type inference algorithm infers usually more than one correct typing. In the old version of Java-TX, the user must select a result to use for bytecode generation. In the future the selection should be made automatically. To achieve this, all correct typings must be used in the bytecode generation. In the official standard of Java methods with generic data types can not be overloaded. This problem should be solved in Java-TX by coding the descriptors of parameterized types into valid class names. For each parameterized type, an empty class must be generated which should has the new name and inherit from the actual class so that it can perform all functions of that class.

1 Bytecode Generation

The Java compiler does not translate Java code directly into a machine code, but into a bytecode. The generated bytecode is stored in a binary class file and then can be executed with the Java Virtual Machine (JVM).
Bytecode generation is the last component of the Java-TX project, in which the Abstract Syntax Tree (`AST`) that is generated by parser is translated into bytecode using the result of the type inference.
The following example should illustrate the compile process in Java-TX project.

Example 1. The class Example has one method that takes a parameter a. The parameter a is assigned to a string value and will be returned:

```
class Example {
        m(a) {
                a = "result";
                return a;
        }
}
```

Figure 1 shows the generated `AST` of the Java program in Example 1.

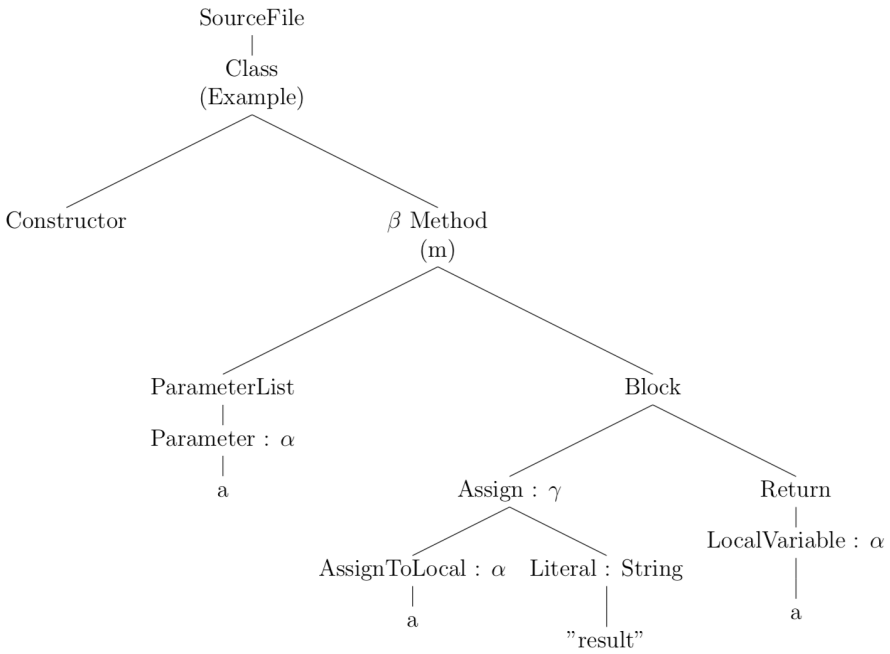


Fig. 1. The generated AST of the Java program

The type annotations are given as type placeholder (greek letters). The following typings are determined by type inference:

$$[[\alpha \mapsto \texttt{String}, \beta \mapsto \texttt{String}, \gamma \mapsto \texttt{String}]].$$

As a next step, bytecode is to be generated. In bytecode generation each node of the `AST` will be visited starting from the root node to the leaves. In each node the necessary informations will be translated into bytecode and the result obtained from the type inference will compensate for the missing types.
In the beginning of bytecode generation the root node `SourceFile` is visited. This node does not contain any necessary informations for bytecode so we do nothing in this node, then the child node `Class` is visited. It includes the access flags, the name, the superclass and the implemented interfaces of the class. We translate all these informations into bytecode. The node `Class` has two child nodes which are `Consturctor` and `Method`. The node `Constructor` is visited first. It is the default constructor which is automatically inserted in the bytecode by the compiler. In this node the following bytecode is generated:

```
public Example();
descriptor: ()V
flags: (0x0001) ACC_PUBLIC
Code:
```

```
stack=1, locals=1, args_size=1
0: aload_0
1: invokespecial #9 //Method java/lang/Object."<init>":()V
4: return
```

The default constructor is public and takes no arguments. The opcode `aload_0` pushes the `this`-reference onto the operand stack. The opcode `invokespecial` invokes the constructor of this class's superclass, in this case it is `Object`.

Afterwards, the node `Method` is visited. It contains the necessary informations for bytecode generation which are the modifiers, the name and the descriptor of the method `m`. First we generate the method descriptor using the return type of the method and the type of the parameter `a` which are calculated by type inference. The following descriptor is generated:

```
(Ljava/lang/String;)Ljava/lang/String;
```

We translate all the above informations into bytecode.

Thereafter the node `ParameterList` is visited which contains just one parameter. The position of the parameter in the array of local variables, and the type of parameter are stored for later use.

Then the node `Block` is visited. It has two child nodes which are `Assign` and `Return`. First we visit the node `Assign` in which there is no necessary informations for bytecode generation, then we visit the node `Literal`. It is a string literal, and it has the value `"result"`. This value has to be pushed onto the operand stack using the bytecode instruction `ldc`. Afterwards the node `AssignToLocal` is visited. In this node the value on the stack should be popped and then stored in position `1` (the position of parameter `a`) in the array of the local variables. This step could be done using the bytecode instruction `astore_1`.

Finally the node `return` is visited in which the parameter `a` has to be returned. For this the patemeter `a` must be pushed from the array of the local variables onto the operand stack using the bytecode instruction `aload_1`, then it has to be returned using the bytecode instruction `areturn`.

When all nodes in the `AST` have been visited and all necessary informations have been translated into bytecode, then it will be stored in a binary class file.

The following cutout of the translation of the class `Example` shows the generated bytecode of the method `m`:

```
java.lang.String m(java.lang.String);
descriptor: (Ljava/lang/String;)Ljava/lang/String;
flags: (0x0000)
Code:
stack=1, locals=2, args_size=2
0: ldc #2 // String result
2: astore_1
```

```
3: aload_1
4: areturn
```

Example 2. The translated bytecode class corresponds to

```
class Example {
    String m(String a) {
        a = "result";
        return a;
    }
}
```

2 Using all typings

Usually the type inference does not only calculate one set of results, but also a lot of results, such as class `Add` in Example 3.

Example 3. The class Add illustrates intersection types caused by overloading.

```
class Add {
        m(a) {
                return a + a;
        }
}
```

In the method `add` the plus operator is performed on the parameter `a`. As mentioned in Example 3, the plus operator can be an addition, as well as a concatenation in java. The addition can be performed on the numeric types which include `Integer`, `Double`, `Long` and `Float`, whereas the concatenation can be preformed on `String`.
According to the above, under the assumptions that the type placeholder α stands for the argument type and the type placeholder β stands for the return type of `add`, the following typings are determined by type inference:
$[[\alpha \mapsto \texttt{Integer}, \beta \mapsto \texttt{Integer}], [\alpha \mapsto \texttt{Double}, \beta \mapsto \texttt{Double}],$
$[\alpha \mapsto \texttt{Long}, \beta \mapsto \texttt{Long}], [\alpha \mapsto \texttt{Float}, \beta \mapsto \texttt{Float}],$
$[\alpha \mapsto \texttt{String}, \beta \mapsto \texttt{String}]]$
In the old version of Java-TX, the user must select a result to use for bytecode generation. However, choosing a result restricts the functionality of the class.
At present we aim to use all possible results of the type inference in order to generate a general class as possible. The user should not have to select a result for bytecode generation; the selection should be made automatically. In order to obtain the most general class, we generate a method for each typing in the bytecode so that it is overloaded. The bytecode corresponds to the following Java class:

```
class Add {
  Integer add(Integer a) { return a + a; }

  Double add(Double a) { return a + a; }

  Long add(Long a) { return a + a; }

  Float add(Float a) { return a + a; }

  String add(String a) { return a + a; }
}
```

This is a correct Java program.
Summarized the following Java-TX program is correct:

```
class Add {
    add(a) { return a + a; }
}

class Main {
  public static void main(String[] args) {
    Add a = new Add();
    a.add(1);
    a.add("xxx");
  }
}
```

3 Generics in function types

Another problem is overloading methods with generic parameters. The generic type information is not available at runtime because the generic implementation uses `type erasure`. This problem arises especially with our function types.
Let us consider the following example:

Example 4. Let the following method be given:

```
class Generics {
  public String m(Fun0<String> f) {
    ...
  }
}
```

The following is an excerpt of the translation of the class `Generics`:

```
...
public java.lang.String m(Fun0<java.lang.String>);
```

```
descriptor: (LFun0;)Ljava/lang/String;
flags: ACC_PUBLIC
⋮
Signature:#108 //(LFun0<Ljava/lang/String;>;)Ljava/lang/String;
...
```

If we take a look the method descriptor in the generated bytecode, we will see that all the informations between < and > are removed by type erasure. Therefore, the information about the parameters are not available at runtime. We want to overload the method in the above example, so we define a new method with the same name but where its parameter has a different type. Let us consider the following example:

Example 5. Let the following Java code be given:

```
class Generics {
  m(f) {
    var ret = f.apply() + f.apply();
  }
}
```

The type inference infers the types `Fun0<Integer>`, ..., `Fun0<String>` for `f`. This would lead to the following program:

```
class Generics {
  Integer m(Fun0<Integer> f) {...}
  ...

  String m(Fun0<String> f) {...}
}
```

This is not a correct Java program. The compiler will return the following error message:

```
name clash: m(Fun0<Integer>) and m(Fun0<String>) have the same
erasure
```

This error message will be returned because both methods have the same descriptor. In order to prevent this problem, the Java Runtime Environment (JRE) should be able to distinguish the descriptors of the overloaded methods with generic parameters. Let us consider the signatures of parameterized types in Example 5:

```
LFun0<Ljava/lang/Integer;>;
LFun0<Ljava/lang/String;>;
```

As we can see above, the signatures of the parameterized types are not the same, but we cannot use this syntax in the descriptor because the JVM has a checker which checks that no parameterized types are used. Our approach in resolving this problem is to change the syntax of the signature by replacing the separator characters / by $$ and the characters < and > by $$$. Subsequently that we use the new syntax in the descriptors of the overloaded methods with a parameterized type as follows:

```
LFun0$$$Ljava$$lang$$Integer$$$;
LFun0$$$Ljava$$lang$$String$$$;
```

This means that the parameterized type is changed into a class name. Each parameterized type is presented as a separate class. Therefore, they are distinguishable from one another. Since these types are used in the descriptors of the overloaded methods in Example 5, they are also distinguishable from one another. The following are the new descriptors of both methods in Example 5:

```
(LFun0$$$Ljava$$lang$$Integer$$$;)Ljava/lang/Integer;
(LFun0$$$Ljava$$lang$$String$$$;)Ljava/lang/String;
```

Now, the JRE can distinguish the descriptors of the overloaded methods with parameterized types but it does not know the types. In order for the JRE to know the types, we have to generate a new class for each type that has the new name. These generated classes are empty, they do not have any variables or methods and they should inherit from the actual class so that they can perform all functions of that class.

4 Conclusion and future work

In this paper we presented the bytecode generation, which generates a method element in the bytecode for each inferred element of each method's intersection type, such that all inferred overloadings can be used. Furthermore, we described the possibility of using generics for function types in bytecode.
we plan to consider the *type erasure* in bytecode. In similar fashion as for the function types, the parameters should be used in bytecode for all parameterized types. There are indeed a number of challenges. A first approach could to generate a corresponding class for each type instantiation.

Muli: Constraint-Programmierung in Java auf symbolischer JVM

Jan C. Dageförde und Herbert Kuchen

ERCIS, Leonardo-Campus 3, D-48149 Münster, Germany
{dagefoerde,kuchen}@uni-muenster.de

Zusammenfassung. Die Münster Logic-Imperative Language (Muli) ergänzt die Programmiersprache Java um Möglichkeiten zur integrierten Lösung von Suchproblemen. Muli beherrscht logische Variablen und eingekapselte Suche mit nichtdeterministischer Ausführung. Durch Java als Grundlage bietet Muli auch bei der Implementierung von Suchproblemen gewohnte und geschätzte Features objektorientierter Programmiersprachen. Die zugehörige Laufzeitumgebung adaptiert Konzepte der Warren Abstract Machine, insbesondere Choice Points und Trail, für die objektorientierte Programmierung. Dies ermöglicht eine enge Integration von objektorientierter und logischer Programmierung innerhalb eines Programms. Der verwendete Constraint Solver stellt sicher, dass nichtdeterministische Ausführung nur auf diejenigen Ausführungspfade beschränkt ist, deren Pfadbedingung erfüllbar ist. Die Auslagerung von nichtdeterministischer Ausführung und Suche an externe Programme, beispielsweise an spezialisierte Prolog-Programme, entfällt vollständig.[1]

1 Motivation und Einordnung

In der Praxis hat sich objektorientierte Programmierung mit Sprachen wie Java an vielen Stellen durchgesetzt, u. a. durch Features wie Kapselung von Struktur und Verhalten sowie Vererbung. Java ist allerdings nur bedingt zur Lösung von Suchproblemen, wie man sie beispielsweise bei der Personaleinsatzplanung oder der Produktionsplanung vorfindet, geeignet. Zwar lassen sich Constraint Solver als Bibliotheken einbinden; jedoch sind diese wenig intuitiv in Programme einzubetten und nicht standardisiert.[2] Auch sind Constraint Solving-Abschnitte von der übrigen objektorientierten Programmierung sprachlich getrennt. Gegebenenfalls wird die Lösungsfindung sogar an Prolog-Programme mit CLP(FD) [Tri12] ausgelagert, wodurch die Integration in bestehende Java-Programme zusätzlich erschwert wird.

Vor diesem Hintergrund wird die *Münster Logic-Imperative Language (Muli)* vorgestellt, welche darauf abzielt, objektorientierte und logische Programmierung

[1] Extended Abstract des Beitrags, der am 10.05.2017 auf dem 34. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte vorgestellt wurde.

[2] Bestrebungen zur Schnittstellenstandardisierung wie JSR 331 scheinen nicht mehr aktiv verfolgt zu werden, vgl. https://jcp.org/en/jsr/detail?id=331.

in integrierter Form zu ermöglichen. Sie ist insbesondere zur Implementierung von Suchproblemen geeignet, die zur Laufzeit dynamisch spezifiziert werden. Der Entwicklung von Muli werden mehrere Prinzipien zugrunde gelegt. Erstens, dass Suchprobleme auf Ebene der Laufzeitumgebung gelöst werden, sodass die Suche den Zustand der Laufzeitumgebung berücksichtigt, aber auch beeinflussen kann. Zweitens soll Suche eingekapselt sein, sodass nichtdeterministische Ausführung (und damit Suche) nur gezielt stattfindet, während übrige Programmteile deterministisch und ohne Overhead ggü. Java funktionieren. Drittens soll so wenig Syntax wie möglich ergänzt werden; stattdessen soll bestehende Syntax von Java übernommen und geeignet adaptiert werden, z. B. zur Definition von Constraints. Viertens soll die Ausführung von Muli-Programmen analog zu der von Java-Programmen nicht lazy sein. Abschließend stellt Muli eine Erweiterung zu Java dar, wodurch impliziert wird, dass die Funktionalität von Java-Features und -Syntaxelementen in deterministischen Teilen von Muli-Programmen identisch zu denen in Java-Programmen sein soll.

In Abschnitt 2 wird Muli vorgestellt, insbesondere hinsichtlich der Sprachfeatures und der Laufzeitumgebung. Abschnitt 3 umreißt verwandte Arbeiten. Anschließend fasst Abschnitt 4 die Ergebnisse zusammen und gibt einen Ausblick auf Forschungsperspektiven.

2 Münster Logic-Imperative Language

Muli ist eine Programmiersprache für (constraint-) logische objektorientierte Programmierung. Dazu ergänzt sie die Programmiersprache Java um Features, die der vereinfachten Lösung von Suchproblemen dienen. Muli beherrscht den Umgang mit *freien Variablen*, d. h. logischen Variablen, die nicht gebunden sind. In Muli-Programmen werden sie deklariert und definiert indem das **`free`**-Schlüsselwort verwendet wird:

```
int x free;
```

und stehen nachfolgend dem Programm zur Verfügung [DK18].

Suchprobleme werden in Form von *Constraints* über freien und übrigen Variablen definiert. Dabei verzichtet Muli zunächst auf besondere Operatoren oder andere Syntax, um Constraints zu formulieren. Stattdessen entstehen Constraints wenn Kontrollstrukturen während der Suche nichtdeterministisch ausgeführt werden, also wenn ihre Bedingungen Ausdrücke enthalten, die nicht eindeutig entschieden werden können [DK17], wie z. B.

```
while (x > 5) {...}
```

mit der freien Variable `x`. Die Bedingung wird symbolisch ausgewertet und kann sowohl wahr als auch falsch sein, abhängig von konkreten Werten für `x`. Für den weiteren Programmverlauf werden systematisch beide Optionen ausgewertet; analog zur Warren Abstract Machine (vgl. [War83]) wird ein *Choice Point* erzeugt. Zunächst wird eine Option angenommen und ein zugehöriger Constraint dem bestehenden Constraint-System hinzugefügt, z. B. $x > 5$ für die Annahme, dass die Schleifenbedingung erfüllt ist [DK17]. Anschließend wird die

Kontrollstruktur passend ausgeführt, bis eine Lösung erreicht wird. Es entsteht ein symbolischer Ausführungsbaum (vgl. [Kin76]), dessen Blätter den Lösungen der Suche entsprechen. Anschließend findet *Backtracking* zu einem Choice Point statt, der eine weitere Option ermöglicht.

Die Beschreibung der Suchprobleme findet in Methoden statt, sogenannten *Suchbereichen* [DK18]. Sie können als Lambda-Ausdrücke oder in Form von Methoden definiert werden. Letztere werden über Methodenreferenzen zur Ausführung gebracht und können so wiederverwendet werden. Suchbereiche definieren anhand des **`return`**-Schlüsselworts, welche Werte von ihnen als Lösungen zurückgegeben werden. Der Rückgabetyp ist beliebig wählbar und kann daher auch ein Array oder ein Objekt sein, welches die Werte mehrerer Variablen zurückgibt. Darüber hinaus gelten auch (nicht gefangene) Exceptions als Lösungen, da auch sie zu Blättern des symbolischen Ausführungsbaums führen. Ob diese als gleichberechtigte Lösungen behandelt oder schlicht ignoriert werden, bleibt dem ausführenden Programm überlassen. Beispielsweise sind Exceptions für die Implementierung eines Generators von JUnit-Testfällen relevante Lösungen, die zu entsprechenden Testfällen führen müssen. Sollen einzelne Lösungen gezielt ausgeschlossen werden, stellt Muli die Methode `Muli.`**`fail`**`()` zur Verfügung, die innerhalb eines Suchbereichs ausgeführt werden kann, was zu sofortigem Backtracking führt [DK18].

Um die Effekte nichtdeterministischer Suche für Java-Programmierer verständlich zu machen und die Suche kontrolliert durchzuführen, ist Nichtdeterminismus nur innerhalb von *eingekapselter Suche* erlaubt [DK18]. Zur Steuerung eingekapselter Suche stellt Muli eine Schnittstelle aus mehreren Methoden bereit, welche einen Suchbereich annehmen. Diese führen den Suchbereich nichtdeterministisch aus und geben dem aufrufenden Programm die gefundenen Lösungen zurück: `Muli.`**`getAllSolutions`**`()` nimmt einen Suchbereich an und wertet diesen aus, bis alle Lösungen errechnet wurden, also der gesamte symbolische Ausführungsbaum des Suchbereichs abgedeckt ist. Die Lösungen werden in `Solution`-Objekten kodiert und gesammelt als Array zurückgegeben. Analog dazu wertet `Muli.`**`getOneSolution`**`()` den Suchbereich aus und gibt die erste gefundene Lösung zurück. Listing 1.1 demonstriert anhand eines einfachen Muli-Programms beispielhaft die Kombination der beschriebenen Konzepte.

```
public static void powersOf2() {
  Solution[] powers = Muli.getAllSolutions(() -> {
    int x free;
    return pow(2, x); });
  }
```

Listing 1.1. Suchbereich zur Bestimmung von 2^x für alle ganzzahligen x unter Annahme einer Methode `pow()`, welche eine Potenz a^b berechnet.

Compiler Ein Compiler für Muli wurde mit Hilfe des Compilerframeworks ExtendJ entwickelt, welches Implementierungen von Java-Compilern enthält und

deren inkrementelle Erweiterung vereinfacht [EH07]. Der Muli-Compiler basiert auf dem ExtendJ-Compiler für Java 8 und fügt das **`free`**-Schlüsselwort hinzu. Lauten die Java-EBNF-Regeln zur Deklaration eines Felds (angelehnt an [GJS+15]) wie folgt:

FieldDeclaration ::= FieldModifier Type VariableDeclarator (, VariableDeclarator)*;*

VariableDeclarator ::= VariableDeclaratorId (= VariableInitializer)?;,

so kann **`free`** als Alternative ergänzt werden, indem die `VariableDeclarator`-Regel modifiziert wird:

VariableDeclarator ::= VariableDeclaratorId (free | (= VariableInitializer))?;.

Wenn der Parser anhand dieser Regel ein freies Feld findet, entsteht im abstrakten Syntaxbaum ein `FreeFieldDeclarator`-Objekt, anstelle eines Objekts vom Typ `FieldDeclarator` für konventionelle Felder. Analog dazu werden freie lokale Variablen in Methoden ermöglicht, die im abstrakten Syntaxbaum als `FreeVariableDeclarator`-Objekte repräsentiert werden. Der vom Muli-Compiler erzeugte Bytecode ist JVM-kompatibel (vgl. [LYBB15]); die Information über freie Variablen und Felder bildet der Compiler in den erweiterbaren `method_info`- bzw. `field_info`-Strukturen des Bytecodes ab [DK18].

Die darüber hinausgehende Schnittstelle stellt keine Spracherweiterung dar, sondern ist in Form einer Laufzeitbibliothek implementiert, welche sowohl zur Kompilierung als auch zur Laufzeit zur Verfügung steht. In dieser stellt die Klasse `de.wwu.muli.Muli` statische Schnittstellenmethoden zur Verfügung, u. a. **`fail`**`()` und **`getAllSolutions`**`()` zur Verfügung. Interaktionen mit der Laufzeitumgebung sind darin als **`native`** abstrakt deklariert und werden von der Laufzeitumgebung implementiert.

Laufzeitumgebung Die Ausführungsumgebung basiert auf der symbolischen JVM (SJVM) des Glassbox-Testfallgenerators Muggl und generalisiert diese [DK18]. Zusätzlich zu den bekannten Datenstrukturen, die den Zustand einer JVM repräsentieren (insbesondere Frame-Stack, Operandenstack und Heap), unterhält die SJVM an die Warren Abstract Machine angelehnte Datenstrukturen (vgl. [War83]), die besonders der nichtdeterministischen Ausführung dienen: Während der Ausführung verzweigender Kontrollstrukturen werden Choice Points erzeugt, die abbilden, an welchen Stellen weitere Auswahlmöglichkeiten bestehen. Diese werden auf einem entsprechenden Stack abgelegt. Außerdem enthält jeder Choice Point den Trail, eine Stack-Struktur, welche Operationen enthält, die notwendig sind, um die SJVM beim Backtracking in denjenigen Zustand zurückzuversetzen, in dem sie eine weitere Auswahlmöglichkeit des Choice Points auswerten kann. Diese integrierte Struktur der SJVM ermöglicht, dass objektorientierte (imperative) Programmierung und logische Programmierung innerhalb desselben Programms eng verzahnt werden können, sodass Suchprobleme und übrige Geschäftslogik in derselben Sprache implementiert werden können. Sie ermöglicht außerdem, dass sich die Suche genau wie objektorientierte Programme auf beliebige Aspekte des SJVM-Zustands beziehen und diese verändern kann (z. B. Felder von Objekten, welche nicht direkt Eingabe- oder Ausgabeparameter eines Suchbereichs sind).

Darüber hinaus verwaltet die Muli SJVM eine Solver-Komponente, welche die Suche nach spezifischen Lösungen übernehmen [DK18]. Außerdem beschleunigt sie die Suche nach gültigen Lösungen indem sie Ausführungspfade frühzeitig ausschließt, deren Constraint-System nicht erfüllbar ist. Die Solver-Komponente ist flexibel erweiterbar, um unterschiedliche Solver-Bibliotheken anzubinden. Derzeit sind zwei Solver implementiert. Einer ist ein SMT-Solver, der für die automatisierte Testfallgenerierung entwickelt wurde [LCMK04]. Der zweite ist der Finite-Domain-Solver JaCoP [Kuc03], der sich durch effiziente Constraint Propagation auszeichnet.

Die Struktur der Laufzeitumgebung und insbesondere die Beziehungen zwischen Komponenten der Muli SJVM werden in [DK18] illustriert und näher beschrieben.

3 Verwandte Arbeiten

Verschiedene Bibliotheken ermöglichen Constraint Solving mit Java, z. B. JaCoP [Kuc03]. Allerdings sind ihre Schnittstellen nicht standardisiert und mit ihnen formulierte Suchprobleme laufen meist getrennt vom restlichen Programm ab. Im Gegensatz dazu können Muli-Suchbereiche den gesamten SJVM-Zustand in die Suche einbeziehen und erlauben somit eine nahtlosere Integration. Logic Java ist ein Ansatz, der bereits versuchte, die SJVM von Muggl zu einer eigenständigen Laufzeitumgebung zu generalisieren [MK11]. Allerdings ist die Sprache in ihren Möglichkeiten stark eingeschränkt, z. B. bei der Deklaration freier Variablen. Alternative Ansätze erweitern eine deklarative Sprache um Features der Objektorientierung, u. a. Prolog++ [Mos94]. Diese erlauben allerdings keinen imperativen Programmierstil und sind daher für Entwickler, die an objektorientierte Programmiersprachen gewöhnt sind, weniger zugänglich.

Die Integration von Features funktionaler Programmierung in objektorientierte Programmierung wird gerne angenommen, z. B. die Java Stream API [UFM14] und LINQ in C# [MBB06]. Gleichermaßen könnte die Integration von (constraint-) logischer Programmierung in die Objektorientierung nützlich sein.

Am ehesten mit Muli vergleichbar ist Curry, eine Programmiersprache, die funktionale und logische Programmierung in Haskell-Syntax miteinander vereint [HKMN+95]. Die Konzepte der eingekapselten Suche und der Constraint-Definition haben Teile von Muli inspiriert, wobei sie sich in ihrer Implementierung grundlegend voneinander unterscheiden.

4 Abschließende Bemerkungen

Durch Muli entfällt die Auslagerung von nichtdeterministischer Ausführung und Suche an externe Programme, beispielsweise an spezialisierte Prolog-Programme, sowie die Einbettung von Constraint Solving-Laufzeitbibliotheken, vollständig. Stattdessen werden diese Aufgaben nahtlos von der virtuellen Maschine übernommen. Als Programmiersprache beherrscht sie die integrierte Implementierung von Suchproblemen neben sonstiger Geschäftslogik. Im Gegensatz zu rein deklarativen

Programmiersprachen baut Muli auf Java auf und beherrscht daher Features wie Vererbung und Kapselung so, wie Entwickler sie von objektorientierten Sprachen erwarten.

Auf Grundlage des hier vorgestellten Entwicklungsstands wurde die operationale Semantik für Muli spezifiziert, welche aufzeigt, an welchen Stellen, unter welchen Umständen und in welcher Form Nichtdeterminismus in eingekapselter Suche zu erwarten ist [DK17]. Dies dient u. a. der Implementierung von Laufzeitumgebungen für Muli. In einer weiteren Veröffentlichung wird die Implementierung einer symbolischen JVM als prototypische Laufzeitumgebung für Muli beschrieben und ihre Performance für mehrere Suchprobleme evaluiert [DK18]. Der Compiler und die symbolische JVM sind freie Software und unter `https://github.com/wwu-pi/muli` zugänglich.

Der derzeitige Stand der Forschung eröffnet spannende Perspektiven. So ist z. B. die Größe des Lösungsraums eines Suchbereichs erst bekannt, nachdem er vollständig durchsucht wurde [Kin76]. Dabei besteht die Möglichkeit, dass der Lösungsraum unendlich groß ist. Schon daher ist es nicht praktikabel, den Suchbereich vollständig zu durchsuchen, bevor einzelne Lösungen betrachtet werden. Daher wird Gegenstand zukünftiger Arbeiten u. a. sein, die Möglichkeit zu schaffen, Suchbereiche zu einem beliebigen Zeitpunkt zur Laufzeit fortzusetzen, sodass die Suche nach einer wahlfreien Anzahl gefundenen Lösungen unterbrochen und u. U. wieder aufgenommen werden kann. Desweiteren ist die Arbeit mit Constraints über Objekten und Arrays interessant.

Literatur

DK17. Dageförde, Jan C. ; Kuchen, Herbert: An Operational Semantics for Constraint-logic Imperative Programming. In: *Proc. Declare 2017*, 2017

DK18. Dageförde, Jan C. ; Kuchen, Herbert: A Constraint-logic Object-oriented Language. In: *SAC 2018*, ACM, 2018. – ISBN 978–1–4503–5191–1/18/04, S. 1185–1194

EH07. Ekman, Torbjörn ; Hedin, Görel: The JastAdd extensible Java compiler. In: *OOPSLA*, 2007. – ISBN 978–1–59593–786–5, S. 1–18

GJS+15. Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad ; Buckley, Alex: *The Java® Language Specification – Java SE 8 Edition*. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`. Version: 2015

HKMN+95. Hanus, M ; Kuchen, H ; Moreno-Navarro, J J. ; Votano, JR ; Parham, M ; Hall, LH: Curry: A Truly Functional Logic Language. In: *ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), S. 95–107

Kin76. King, James C.: Symbolic execution and program testing. In: *Communications of the ACM* 19 (1976), Nr. 7, S. 385–394. `http://dx.doi.org/10.1145/360248.360252`. – DOI 10.1145/360248.360252

Kuc03. Kuchcinski, Krzysztof: Constraints-driven scheduling and resource assignment. In: *ACM Transactions on Design Automation of Electronic Systems* 8 (2003), Nr. 3, S. 355–383. `http://dx.doi.org/10.1145/785411.785416`. – DOI 10.1145/785411.785416. – ISBN 1084–4309

LCMK04. Lembeck, Christoph ; Caballero, Rafael ; Müller, Roger A. ; Kuchen, Herbert: Constraint Solving for Generating Glass-Box Test Cases. In: *Proceedings WFLP*, 2004, S. 19–32

LYBB15. Lindholm, Tim ; Yellin, Frank ; Bracha, Gilad ; Buckley, Alex: *The Java® Virtual Machine Specification – Java SE 8 Edition*. `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`. Version: 2015

MBB06. Meijer, Erik ; Beckman, Brian ; Bierman, Gavin: LINQ: Reconciling Objects, Relations and XML in the .NET Framework. In: *ACM SIGMOD Conference on Management of data*, 2006, S. 706

MK11. Majchrzak, Tim A. ; Kuchen, Herbert: Logic Java: Combining Object-Oriented and Logic Programming. In: *WFLP*, 2011, S. 122–137

Mos94. Moss, Chris: *Prolog++ - the power of object-oriented and logic programming*. Addison-Wesley, 1994 (International series in logic programming). – ISBN 978–0–201–56507–2

Tri12. Triska, Markus: The Finite Domain Constraint Solver of SWI-Prolog. In: *FLOPS* Bd. 7294, 2012 (LNCS), S. 307–316

UFM14. Urma, Raoul-Gabriel ; Fusco, Mario ; Mycroft, Alan: *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. Greenwich, CT : Manning Publications Co., 2014. – ISBN 9781617291999

War83. Warren, David H. D.: An Abstract Prolog Instruction Set / SRI International. Menlo Park, 1983. – Forschungsbericht

Verwendung von Programmiersprachentechniken in der Sozialwahltheorie

Jan Christiansen[1], Sandra Dylus[2], Johannes Hedtrich[3], Christian Henning[3], and Rudolf Berghammer[2]

[1]Fachbereich Information und Kommunikation, Hochschule Flensburg
[2]Institut für Informatik, Christian-Albrechts-Universität zu Kiel
[3]Institut für Agrarökonomie, Christian-Albrechts-Universität zu Kiel

Legislative Bargaining beschreibt einen Prozess, mit dessen Hilfe politische Entscheidungen modelliert werden. Dabei werden immer wieder Vorschläge zur Abstimmung gestellt bis sich eine Mehrheit für einen Vorschlag findet. Ein Beispiel für einen solchen Prozess ist etwa die Entscheidungsfindung in der EU-28.

Die Wahrscheinlichkeit, dass ein Vorschlag, der zur Abstimmung steht, angenommen wird, lässt sich aus den Wahrscheinlichkeiten berechnen, dass die einzelnen Spieler dem Vorschlag zustimmen. Eine solche einzelne Abstimmung kann mit Hilfe eines gewichteten Mehrheitsspiels modelliert werden. Für dieses gewichtete Mehrheitsspiel kann dann die Menge der gewinnenden Koalitionen bestimmt werden, wobei eine Koalition eine Menge von Spielern ist. Eine gewinnende Koalition ist eine Koalition, welche die Abstimmung gewinnt, wenn alle Spieler der Koalition zustimmen.

Um die Wahrscheinlichkeit zu bestimmen, dass ein Vorschlag angenommen wird, müssen die Wahrscheinlichkeiten der einzelnen Spieler jeder gewinnenden Koalition multipliziert und die sich ergebenden Wahrscheinlichkeiten am Ende aufsummiert werden. Eine naive Durchführung dieser Berechnung ist nicht möglich. Das Spiel, das die Abstimmung in der EU-28 modelliert, hat zum Beispiel 7 722 293 166 gewinnende Koalitionen.

Wir zeigen, wie die Annahme-Wahrscheinlichkeit mit Hilfe einer Faltung über eine Baumstruktur wesentlich effizienter berechnet werden kann. An Stelle der 7 722 293 166 Additionen in der naiven Variante, müssen in diesem Fall nur noch 10 268 Additionen durchgeführt werden. Mit Hilfe von Techniken aus dem Bereich der Programmiersprachen erhalten wir bei der Berechnung auch direkt die Garantie, dass die Berechnung korrekt ist. Das heißt, die Berechnung mit 7 722 293 166 Additionen berechnet in allen Fällen das gleiche Ergebnis wie die Berechnung mit 10 268 Additionen.

Sig adLib

Mostly Compositional Clocked Synchronous Data-Flow Programming in Java

Baltasar Trancón y Widemann and Markus Lepper

`<semantics/>` GmbH, Berlin, DE

Abstract. We present Sig adLib, a lightweight embedded domain-specific declarative language for clocked synchronous data-flow programming on the Java host platform. The design demonstrates how to reduce data-flow programs, which are known to have non-compositional properties, to a pair of separate, compositional aspects that specify data and control flow, respectively. Sig adLib can be used direcly as a conservative extension of Java, as a target for domain-specific meta-programming, and as a backend for its companion stand-alone language Sig.

1 Introduction

We present the first results on the design and implementation of Sig adLib, a lightweight embedded domain-specific language (DSL) for clocked synchronous data-flow programming on the Java platform. This prototypic programming tool is a building block of the larger Sig project, which also comprises a semantic framework, a compiled stand-alone declarative language, and application-specific libraries.

1.1 Motivating Example: Kahan Summation

The clocked synchronous data-flow programming paradigm addresses a broad area of computational problems that can be illustrated by a (literally) archetypal example: The short article [4], published more than half a century ago, discusses a pair of algorithms for the summation of a sequence of floating-point values. The respective Fortran implementations are depicted in Figure 1.

They have in common that the sum of successive values (of variable YI) is accumulated (in variable S). The naïve algorithm does so in the straightforward way, whereas Kahan's proposed algorithm additionally tracks a compensation term (in variable S2), to the effect that the numerical error bound does no longer increase with the length of the sequence; confer Table 1.

These short code snippets exemplify both the general properties of the problem class of interest, and the particular expressive shortcomings of the imperative paradigm that are amended by clocked synchronous data-flow programming.

```
1    S  = 0.0

2    DO 4 I = 1, N
3    YI = ···
4    S  = S + YI




5    ···
```

```
1    S  = 0.0
     S2 = 0.0
2    DO 4 i = 1, N
3    YI = ···
13   S2 = S2 + YI
     T  = S + S2
23   S2 = (S − T) + S2
4    S  = T
5    ···
```

Fig. 1. FORTRAN summation algorithms [4] – naïve (left); Kahan (right, emph. added).

n	**naïve**	**Kahan**
1 000	0.999 991	1.000 000
10 000	10.000 411	10.000 000
100 000	99.956 696	100.000 008
1 000 000	991.141 541	1 000.000 061
10 000 000	9 780.204 102	10 000.000 000
100 000 000	32 768.000 000	100 000.007 813

Table 1. Results of summing n times the approximate constant 0.001f

1.2 Common Properties of the Problem Class

1. Programs operate on a moderate number of mutable variables which are each updated frequently, specifically at each clock tick; thus each variable represents a *stream* of successive values, indexed by discrete clock time.
2. In contrast to the variables that serve as operands and results, the operators executed by the program do not change essentially over time; thus a *data-flow network* fully describes the algorithm, except for initial values of mutable variables.
3. Each variable has a single writer, but an unconstrained number of readers.
4. Some variables are *inputs*; their updates are not controlled by the program. It is algorithmically inconsequential whether the program operates *autonomously* on a (necessarily bounded) array, or *reactively* on a (potentially unbounded) real-time stream of input values; thus the essential program structure is just a *loop body*.
5. Some variables are *outputs*; their updates are the purpose of the program. Well-behaved programs implement *causal*[1] stream functions that map inputs to outputs.
6. Any mutable variable that is updated under program control may be observed both *after* and *before* the update; thus dependent operations may appear to operate on the synchronous stream of values or a *delayed* copy, respectively.

[1] A stream function is causal if and only if each output elements is determined by past and present inputs, independently of the future.

7. Data-flow dependencies between variables are dense and typically circular.

In [7] we have argued that these properties are captured adequately by a fully abstract semantic foundation on *coiterative* causal stream functions, namely as final coalgebras in categories of Mealy automata.

1.3 Expressive Shortcomings of the Imperative Paradigm

From the perspective of reasoning about programs, the imperative approach as exemplified above lacks two crucial desirable properties:

1. *Referential transparency.* Distinct occurences of the same symbol in the program, even though they purport to refer to the same semantic entity, have very different meanings.
2. *Compositionality.* The behavior of the whole program can not be expressed clearly in terms of the behavior of its parts.

These shortcomings are by no means a novel topic; they have been criticized famously as the weaknesses of the "von Neumann style" in [2]. There, an alternative style is suggested that constructs complex programs by applying function-level operators to simpler programs. This approach has been demonstrated to be very effective for many problem classes. In particular light-weight solutions, requiring at most cosmetic support from the "conventional" host language, have found widespread practical acceptance.[2]

Unfortunately for the causal stream function type of problems, they tend to lack the property of compositionality in an essential way; not all relevant properties of a program fragment can be determined without considering its context. This poses a serious obstacle to applicative approaches. In the remainder of this paper, we shall argue how the Sig adLib design *almost* restores compositionality, in a sense that is theoretically honest and practically useful.

2 The Sig Approach

Figure 2 depicts the data flow expressed in the Fortran algorithm for Kahan summation from Figure 1, in the style we have used to reason about Sig programs [7]. All lines are understood to denote stream variables. Arithmetic operators apply elementwise. The special operator δ delays the input stream by a single clock tick. The initial output value is omitted in the diagram; here both instances are initialized to zero.

The flow graph corresponding to the algorithm can be found by a static analysis that is mostly equivalent to transforming an imperative program into static single-assignment (SSA) form. Note that the established compilation

[2] Consider computations on finite collections of data, and their implementation in terms of higher-order operations such as *map*, *filter* and *reduce*, in languages such as Java, Scala, Python.

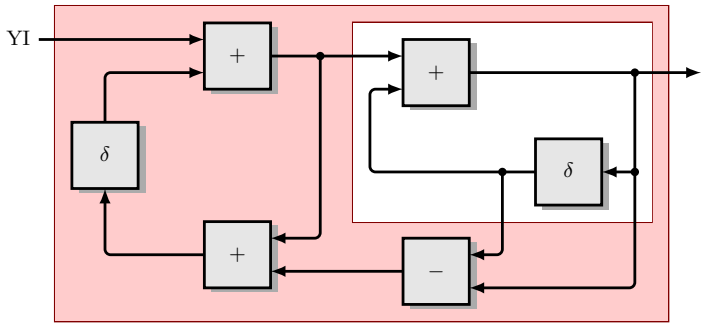


Fig. 2. SIG data-flow diagram – naïve summation (white); Kahan summation (colored).

techniques for clocked synchronous data-flow programs, which we have discussed in the SIG context in [9, 10], are sufficient to recover the imperative implementation.

Even for the simple case of Kahan summation, the data-flow diagram exhibits the characteristic properties posited in section 1.2:

1. The variables that need frequent updating correspond one-to-one to the instances of the δ operator. Temporary variables and anonymous intermediate results correspond to edges. Note that, confusingly, the identifier S2 doubles as an updated variable (line 23), and as a temporary result (line 13) that is used twice. We have indicated the latter role by writing *S2* with emphasis.
2. The remaining arithmetic operators correspond one-to-one to operators in the FORTRAN program code.
3. Bullets indicate edge splits where a variable has more than one reader.
4. Inputs are arranged at the far left of the diagram. No termination condition is specified.
5. Outputs are arranged at the far right of the diagram.
6. The mutable variable corresponding to the rightmost δ node is clearly seen to observed both before and after update; as opposed to the indirect expression of this fact in the FORTRAN code lines 23 and 4, involving T as the post-update revenant of S.
7. The data-flow graph is easily seen to be densely connected, and contain four distinct but overlapping cycles. Note that all cycles contain a δ node, and are hence admissible; instantaneous feedback is generally forbidden in the clocked synchronous paradigm.

Note also how the data-flow diagram clearly depicts in which precise sense the Kahan algorithm is an extension of the naïve one.

2.1 From Streams to Elements And Back Again

In the SIG approach, the change of perspective from stream-based to elementwise calculations is expressed as a simple program transformation.

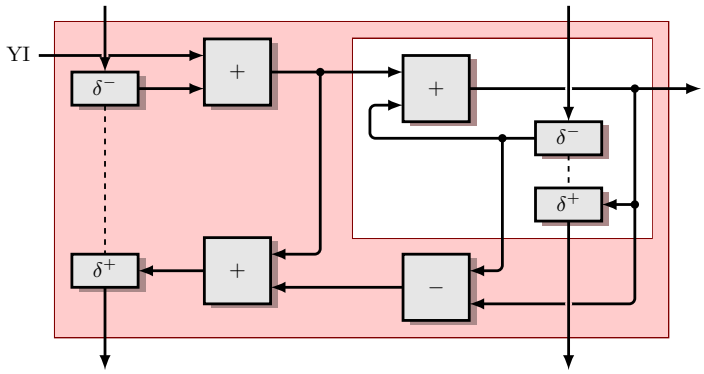


Fig. 3. SIG iteration diagram – naïve summation (white); Kahan summation (colored).

The essential obstacle to viewing a program as operating on single elements is the δ operator, which requires additional state. For each occurence of δ a corresponding pair of variables is added, representing pre-state and post-state, respectively. Then the δ operator splits into two independent identity operators that simply forward pre-state to output and input to post-state, respectively. In Figure 3, pre-state and post-state variables are arranged at the top and bottom of the diagram, respectively, and aligned horizontally for correspondence. We have denoted the split operators as δ^- and δ^+, respectively, and indicated their correspondence by dashed lines.

By this transformation, all admissible cycles of non-instantaneous feedback are broken. The resulting data-flow graph represents one iteration of the program loop in a cycle-free, and hence causally deterministic and effectively computable way. The stream semantics of the program, that is the behavior of the driving loop, follows from this representation by

1. *replicating* the diagram infinitely often downwards, identifying the post-state variables of each iteration with the pre-state variables of the next; and
2. *initializing* the pre-state variables of the first iteration to the specified values, that have been omitted in the data-flow diagram.

This informal operational description appeals directly to the intuition of the programmer, and to experience with low-level computational software structures. But it has also been demonstrated [3, 7] that the scheme is justified formally by the coinductive principle in the theory of Mealy automata.

3 Design of the Sig adLib Language

The design principle underlying the SIG ADLIB language is *compositional reification*: elements of the target program are constructed as a graph of JAVA objects, each carrying enough local information and behavior such that the ensemble is globally executable.

In a heavyweight DSL, host objects would be arranged to represent the abstract syntax of the target program, subject to further processing. By contrast, in a lightweight approach such as the one discussed here, there is no manifest syntax, and the target program may be constructed directly in any convenient way, most notably

1. in simple cases by borrowing the expression syntax of the host language;
2. by ad-hoc meta-programming in the host language;
3. by compilation from a frontend language;

Thus the task of designing a lighweight DSL is closely related to designing the runtime system of a compiled language.

3.1 The Control–Data Dilemma

Achieving compositional reification is not easy, neither in general nor in the particular case of clocked synchronous data-flow programs. An evident problem, and the one that we have addressed in this work, is that the *local* control flow of a clocked synchronous computation depends on its *global* data flow.

Clearly, the *macro-steps* of a computational component, that is the ones observable at its interface, are tied to clocks and hence controlled externally. By contrast, a complex computation may take several *micro-steps* with intermediate results in order to produce a single output element. Synchronicity mandates that a program must behave globally as if micro-steps take no time at all, whereas locally each subcomputation can only be performed when its inputs are available. That is, any schedule for micro-steps must be consistent with (a topological sorting of) the partial order of *causality*, that is the transitive closure of the data-flow graph, which in general is a non-compositional property.

In practice, one can therefore not expect to be able to construct both the data-flow network of a clocked synchronous computation, and a sequential schedule of its operators for execution on a sequential host platform, simultaneously by composition from building blocks.

In declarative data-flow languages, the programmer focuses on the former, and the compiler is expected to deduce the latter automatically by global static analysis; or otherwise the runtime system is trusted to schedule dynamically, see for instance [6]. However, in a DSL intended for manual and ad-hoc meta-programming of low-level software systems, neither is a helpful solution.

The key idea of the Sig adLib design is dialectical, resolving the dilemma by:

1. keeping the aspects of control and data flow separate in the language core, and trusting their consistent use to the programmer; but
2. allowing the programmer to express specific knowledge of joint compositionality, in terms of the host-language facilities for encapsulation.

3.2 Core API

Figure 4 depicts the basic APIs of the data-flow and the control-flow aspects of Sig adLib.

Data Flow From the imperative viewpoint of the Java host environment, variables in data-flow programs have a single writer and arbtrarily many readers. A natural representation in the object-oriented world is to encapsulate a variable as "owned by" its writer, and exposing a public signal source interface accessible for reading by anyone who holds a reference. In the Java standard library, this concept is embodied in the Supplier family of functional interfaces.

The data-flow core API of Sig adLib consist of Supplier-like interfaces for various value types. Figure 4 shows only the **float** case, other types are supported analogously.[3] A data stream is obtained by repeatedly invoking the get method. The interface does not specify whether the variable is actual and simply read from, or virtual and its computed on the fly. It follows that there is no guarantee that multiple concurrent readers will not cause redundant computation. Furthermore, the interface does not specify when updates may occur. Both asynchronous and synchronous data-flow models are supported at this level of abstraction.

Using the lambda notation of Java, computations can be denoted very concisely. For example, a stream of constant value can be denoted as `() -> 42`, and the elementwise sum of two streams `a` and `b` as `() -> a.getAsFloat() + b.getAsFloat()`.

While of great didactical value, these constructs do not benefit from some additional features of the Sig adLib language, to be described in section 5 below. Therefore, our core library also defines factories such that the examples can also be denoted as `constant(42)` and `a.add(b)`, respectively.

Control Flow All control flow in Sig adLib programs is routed through a single interface Realtime that specializes the standard interface Runnable, shown in Figure 4. Synchronous program components that implement this interface may in general update their state only during invocations of the run method; no spontaneous asynchronous mutations are allowed.

A complex component may require micro-step activities to update its observable state. Thus its implementation of run may involve recursive invocations on its building blocks, in a pre-scheduled sequential order.

As for the data flow aspect, Java lambdas can express Sig adLib control flow constructs very concisely. For example, a trivial user output component can be denoted as `() -> System.out.print(".")`, and the sequential scheduling of two components `p` and `q` as `() -> { p.run(); q.run(); }`. Our core library provides factories for many constructs, including the infix `p.andThen(q)` and variadic `sequence(p, q)` for the latter.

Flow Synthesis The synthesis of the dialectical aspects of flow is achieved, in the true spirit of object-orientation, by either multiple inheritance or aggregation from both parts of the core API.

The canonical example is the clocked signal source, depicted in Figure 5 for the case of value type **float**. This abstract class defines the output variable to have

[3] Idiosyncratically, the standard library package java.util.function defines interfaces IntSupplier, LongSupplier and DoubleSupplier which we can extend, but there is no FloatSupplier.

actual storage in the field `out`. Concrete subclasses need to implement the method `run` to update `out`. Thus multiple readers can observe the value without causing redundant computations. On the other hand, there is no *laziness*; computations are performed logically independently of their observers. There is a factory method `clocked()` that wraps any signal source in a clocked proxy.

Note that the clocked synchronicity of SIG ADLIB is not generally *thread-safe*. Since synchronous data-flow programming aims at deterministic, in particular race-free parallelism, it is generally incompatible with shared-memory concurrency as implemented by JAVA threads. On the upside, the synchronous-parallel composition of data-flow computations can be implemented with no significant context-switching overhead, and thus extremely fine-grained interleaving and low latency.

As discussed, SIG ADLIB programs are constructed as loop bodies. A main loop that drives the computation has the general form as depicted in Figure 6. Any technical means of the JAVA host platform for communication and synchronization with the environment may be used.

Delay A slightly more complex, very important case of synthesis of control and data flow is depicted in Figure 7. The implementation of the single-step delay operator builds on the clocked signal source with an output variable, adding an input and a mutable state variable. The class defines the common control flow API, as well as a custom variant consisting of two separate phases `pre` and `post`. The implementation is completely straightforward, and corresponds exactly to the splitting of δ operators as discussed in section 2.1, with `run`, `pre` and `post` implementing δ, δ^- and δ^+, respectively.

A delay operator requires initialization in two senses. In the control-flow world, the state variable requires an initial value, as discussed in section 2.1. In the data-flow world, it needs to be connected to an input. Unlike for arithmetic operators, this is not in general a combinatorial matter, because delay operators are more often than not used in conjunction with circular data flow, and hence the graphs cannot be constructed by successive application of constructors and factories alone; some recursive language construct is required.

The core library provides a setter method to connect a delay object to its input *after* its construction. This should better not be used to modify data-flow graphs destructively, and ruin the declarative properties of the SIG ADLIB language; but it is a very handy basis for implementing safe and expressive resursive-let-like factories. A shorthand for simple cycles as the fixed point of a construction function is provided, as well as a shorthand for the acyclic case; see Figure 8.

4 Example Revisited

4.1 Basic Summation Component

The data flow of the naïve summation of a stream `x`, as depicted in Figure 2, can be expressed with little effort as the fixpoint of a recursive construct `s = new FloatDelay(0, d -> d.add(x))`. However, the devil is in the details:

```
public interface FloatSignalSource {
    public float getAsFloat();
}
```

```
public interface Realtime extends Runnable {
    @Override public void run();
}
```

Fig. 4. SIG ADLIB core API aspects – data flow (left); control flow (right).

```
public abstract class FloatClockedSignalSource implements FloatSignalSource, Realtime {
    protected float out;
    public final float getAsFloat() { return out; }
}
```

Fig. 5. SIG ADLIB core API basic synthesis

```
  while (!terminationCondition()) {
      controlFlow.run();
      react(output1.getAsFloat(), ..., outputN.getAsSomething());
  }
```

Fig. 6. SIG ADLIB main loop (schematic)

```
public class FloatDelay extends FloatClockedSignalSource {
    protected FloatSignalSource source;
    protected float state;

    public FloatSignalSource getSource() { return source; }
    public void run() {
        pre();
        post();
    }
    public void pre() { out = state; }
    public void post() { state = source.getAsFloat(); }
    // ... see below
}
```

Fig. 7. SIG ADLIB single-step delay operator (control flow fragment).

```
public class FloatDelay extends FloatClockedSignalSource {
    // ... see above
    public FloatDelay(float init) { state = init; }
    public void setSource(FloatSignalSource source) { this.source = source; }

    public FloatDelay(float init, UnaryOperator<FloatSignalSource> fun) {
        this( init );
        setSource(fun.apply(this));
    }
    public FloatDelay(float init, FloatSignalSource source) {
        this( init , ___ -> source);
    }
}
```

Fig. 8. SIG ADLIB single-step delay operator (initialization fragment).

```
public class FloatSeries implements Realtime {
    private FloatClockedSignalSource incl;
    private FloatDelay excl;

    public FloatSeries() { excl = new FloatDelay(0); }
    public void setSource(FloatSignalSource source) {
      incl = source.add(excl).clocked();
      excl.setSource(incl);
    }

    public FloatSeries(UnaryOperator<FloatSignalSource> fun) {
        this();
        setSource(fun.apply(excl));
    }
    public FloatSeries(FloatSignalSource source) {
        this(___ -> source);
    }

    public FloatClockedSignalSource getInclusiveSum() { return incl; }
    public FloatClockedSignalSource getExclusiveSum() { return excl; }

    public void run() {
        excl.pre();
        incl.run();
        excl.post();
    }
}
```

Fig. 9. SIG ADLIB summation component

1. When the resulting object is used as the output, the computed sum is *exclusive* rather than *inclusive*, that is the current input value is not (yet) added. The inclusive sum is provided by `s.getSource()`.
2. Since the latter is not clocked, then redundant computation of both the input and the addition operator may ensue, because the value is used both as the input to the delay and as the global output. Thus it is preferrable to write `new FloatDelay(0, d -> d.add(x).clocked())`.
3. The necessary control flow for operating this component is merely `s` in the unclocked case, but more complicated in the clocked case: the micro-step for the clocked addition needs to be sandwiched between the two phases of the delay operator; confer Figure 3.

A reusable component that embodies these design considerations is depicted in Figure 9. It implements composite control flow directly, and for data flow aggregates two signal sources yielding the inclusive and exclusive sum, respectively.

In comparison to the underlying delay, another recursive twist is supported; the term to be added in each step may depend causally on the current exclusive sum, that is the preceding inclusive sum. The API for recursive constructs is analogous to Figure 8.

```
yi = ···;
s = new FloatSeries(yi);




controlFlow = sequence(yi, s);

output1 = s.getInclusiveSum();
```

```
yi = ···;
s = new FloatSeries();
s2 = new FloatDelay(0);
s3 = yi.add(s2).clocked();
s.setSource(s3);
s2.setSource(s3.add(s.getExclusiveSum()
                    .sub(s.getInclusiveSum())));
controlFlow =
    sequence(yi, s2::pre, s3, s, s2::post);
output1 = s.getInclusiveSum();
```

Fig. 10. Sig adLib summation algorithms – naïve (left); Kahan (right).

$$
\begin{aligned}
M_k &= M_{k-1} + (x_k - M_{k-1}) \,/\, k \\
S_k &= S_{k-1} + (x_k - M_{k-1}) * (x_k - M_k) \\
V_k &= S_k \,/\, (k-1)
\end{aligned}
$$

Fig. 11. Variance algorithm from [5]

4.2 Construction of Summation Algorithms

Figure 10 shows constructions for Sig adLib programs corresponding to the Fortran code depicted in Figure 1. Types have been omitted from variable declarations, for simplicity and to make the analogy more recognizable. Note however two crucial differences:

1. The Sig adLib versions are meta-programs that, upon execution, construct the actual algorithm, to be run as depicted in Figure 6. Since the examples are trivial, static meta-programs, they appear as nothing more than a pedantic notation for the abstract syntax of the target program. But the same basic constructions, when combined with generic software-engineering tools such as reuse, parametrization and meta-algorithmics, have a very different scaling behavior than first-order programming.
2. Sig adLib meta-programming are largely applicative, except for the circular wiring expressed using setSource. The two dialectical aspects, control flow and data flow, are constructed separately by applying higher-order operations to building blocks. Sig adLib meta-programming is also compositional; the properties of complex components are no different than the properties of their building blocks.

4.3 Extended Example: Variance

Figure 11 depicts the abstract presentation of a well-known algorithm for online calculation of the variance V of a stream x, using an adaptive mean M in the process. Streams are defined elementwise by a system of simultaneous equations. Again, the computation is made up entirely of arithmetic and delay operators,

```
x = ···;
k = new FloatSeries(constant(1));
m = new FloatSeries(m0 −> x.sub(m0).div(k.getInclusiveSum()));
s = new FloatSeries(x.sub(m.getExclusiveSum())
                    .mult(x.sub(m.getInclusiveSum())));
v = s.getInclusiveSum().div(k.getExclusiveSum()).clocked();
controlFlow = sequence(k, m, s, v);
output1 = v;
```

Fig. 12. SIG ADLIB variance algorithm

the latter indicated by the subscript $k-1$. Thus the formulae also denote a clocked synchronous data-flow program.

The only obstacle to a straightforward translation into the SIG format is the use of the index k not only as a time parameter, but also as a value (the non-subscript occurences). This detail can be fixed by adding a *counter* component, that is a summation of constant one.

Figure 12 depicts an implementation in the SIG ADLIB style. Except for the use of a lambda expression for the recursive definition of m, the construction is completely straightforward and purely applicative. Its apparent triviality demonstrates that a sparse and orderly data-flow graph translates directly into concise compositional constructions.

5 Compilation

In an executable compositional reification approach, each object carries the operational semantics of the computational concept it represents, such that the object graph is both the representation and a (distributed) interpreter of the program. Obviously, interpretation has substantial runtime overhead compared to generated code. On a standard JAVA virtual machine (JVM) platform, just-in-time (JIT) compilation can be expected to compensate some, but surely not all of this overhead. On dedicated real-time platforms, where applications of clocked synchronous data-flow programming abound in the form of controlling and signal processing, JIT compilation may be severely restricted, or even not available at all. In any case, it would be useful if SIG ADLIB had a compiler.

Fortunately, data-flow operations are very different from imperative machine instructions in nature. By the very principle of the paradigm, the context on which they operate is made explicit. This not only makes it possible to distribute the interpreter over the program object graph, without requiring global state to be lugged around. It also allows for compositional code generation.

Besides the interpretation APIs that we have discussed in section 3.2, SIG ADLIB provides support for compilation to JVM bytecode instructions. The resulting custom classes implement the interpretation APIs, and can hence interoperate transparently with interpreted constructs. The code generator uses our own LLJAVA [13] bytecode object model and assembler, and can be used for both static and dynamic meta-programming.

Compilation is organized as an "opt-in" service. The core APIs declare methods for compilation analogous to those for interpretation, and provide default implementations which generate stub code to fall back into interpreted mode. Predefined library components and combinators override these methods to provide actual compilation to bytecode instructions. User-defined components may either rely on compositional compilation, or define their own coding schemes.

5.1 Compilation API

Two interfaces CompilableDataFlow and CompilableControlFlow can be implemented by classes that support compilation of the respective aspect. These interfaces are inherited by the core APIs, with default implementations for stub code generation, such that applications defined without regard to compilation remain usable in compiled settings. See Figures 13 and 14, respectively, and Figure 4.

The workhorse of the compilation API is an interface CompilationContext, not to be described here in detail. It declares actions for management of bytecode instructions, blocks and contexts, state and temporary variables, environment references, and the public API of a class to be generated. An instance is passed to any compilation method. The class LLJavaCompilationContext provides the binding to the LLJava backend.

A class implementing one or more core APIs can provide local compilation support, simply by overriding the compilation method in addition to implementing the corresponding interpretation method. Unfortunately, Java lambda expressions can not be used to achieve this.

Figure 15 depicts a compilable factory for constant streams. Upon compilation, it generates bytecode instructions to store to the 0th output variable of the current data-flow subgraph context the constant value that has been loaded onto the operand stack. In fact, the stack-based nature of the JVM is essential for compositional code generation techniques to work.

Figure 16 depicts a compilable factory for control flow sequences. Upon compilation, it simply emits the compiled code of the steps in order. This example demonstrates the great power of the compilation step to compensate the overhead of fine-grained compositional reification: The interpreted implementation can be criticized as naïve and inefficient, because it is defined generically for any number of steps, and requires a loop to traverse them at runtime. By contrast, this is of no importance when compiling, because the loop is executed ahead of time, and thus unfolded. Since JVM bytecode is inherently sequential and flat, no trace of sequencing operators remains.[4]

5.2 Compiler Usage

Figure 17 depicts the use of the Sig adLib to JVM bytecode compiler. A code generator backend is created, the control flow and data flow aspects are reg-

[4] For more advanced control flow operators, such as 'execute only on every n-th iteration' for oversampling, some extra bytecode will have to be generated; but these are out of scope here.

```
public interface CompilableDataFlow {
    public void compileDataFlow(CompilationContext context);
}

public interface FloatSignalSource extends CompilableDataFlow {
    public float getAsFloat();
    public default void compileDataFlow(CompilationContext context) {
        // generate call to this.getAsFloat();
    }
}
```

Fig. 13. SIG ADLIB data-flow compilation API

```
public interface CompilableControlFlow {
    public void compileControlFlow(CompilationContext context);
}

public interface Realtime extends CompilableControlFlow {
    @Override public void run();
    public default void compileControlFlow(CompilationContext context) {
        // generate call to this.run();
    }
}
```

Fig. 14. SIG ADLIB control-flow compilation API

```
public static FloatSignalSource constant(float value) {
    return new FloatSignalSource() {
        public float getAsFloat() {
            return value;
        }
        @Override public void compileDataFlow(CompilationContext context) {
            context.storeOutput(0, () -> context.load(value));
        }
    };
}
```

Fig. 15. Compilable constant stream

```
public static Realtime sequence(Realtime... steps) {
    return new Realtime() {
        @Override public void run() {
            for (Realtime step : steps)
                step.run();
        }
        @Override public void compileControlFlow(CompilationContext context) {
            for (Realtime step : steps)
                step.compileControlFlow(context);
        }
    };
}
```

Fig. 16. Compilable sequential control flow

```
cc = new LLJavaCompilationContext();
cc.addMainRealtime(controlFlow);
cc.addMainFloatSignalSource(output1);
cc.compile();

controlFlow = (Realtime)cc.instantiate();
output1 = (FloatSignalSource)controlFlow;
```

Fig. 17. Sig adLib bytecode compiler call

istered, and the compilation process is executed. The result is a *closure*, that is the bytecode defining a class, together with a list of values captured during compilation from the environment. For instance, the stub code generated by the default compilation methods depicted in Figures 13 and 14 captures the **this** reference of the owner object. The compiler instantiation service loads the bytecode into the running JVM, creates an object of the new class, passing the captured values to its constructor.

The resulting object supports the core APIs, either through direct implementation or aggregation, depending on the way functionality has been registered for compilation. In the example, both control flow and a single output of **float** type are implemented directly, such that the compiled object can replace its sources transparently.

5.3 Evaluation

Table 2 shows the results of a simple benchmark of interpreted and compiled versions of the two example algorithms discussed above. Measurements have been obtained on a Core i7-5600U CPU at 2.6 GHz, using a single core, running CentOS Linux 7 64-bit and the OpenJDK 1.8.0-45 HotSpot JVM. Algorithms have been run for 10^7 iterations of a tight loop, with constant input, measuring wall-clock running times. The table gives mean values and standard deviations for 10 consecutive repetitions of each run, after a single warm-up round.[5]

For these examples of comparable complexity, the speedup is consistent and significant, and competes well with other programming technology (estimated 10 and 21 CPU cycles per iteration, respectively). The quality of generated code is

[5] The programs run on constant memory, hence garbage collection pauses are not an issue.

	interpreted	compiled	speedup	class size
Kahan Sum	62.71 ± 0.41	3.92 ± 0.20	16	589
Variance	146.84 ± 0.21	8.16 ± 0.19	18	797
	(ns /iteration)			(bytes)

Table 2. Evaluation of Sig adLib programs

quite satisfactory, considering that compilation is mostly compositional, with only a few cleanup optimizations applied to the complete code sequence. It remains to be investigated how the gains of compilation scale with program size. But since

1. the bottleneck of interpreted computation is the fine-grained routing of compositional data and control flow through costly interface method invocations; and
2. the transparent replacement of interpreted by compiled components is compositional as well;

we expect that a sweet spot of compilation granularity can be found easily in practice.

```
vmovss  0x14(%rsi), %xmm1
; S2
vmovss  0x20(%rsi), %xmm0
; S
vaddss  -111(%rip), %xmm1, %xmm2
; YI
vaddss  %xmm0, %xmm2, %xmm1
vmovss  %xmm1, 0x20(%rsi)
; S
vsubss  %xmm1, %xmm0, %xmm1
vaddss  %xmm2, %xmm1, %xmm0
vmovss  %xmm0, 0x14(%rsi)
; S2
```

```
vmovss  0x30(%rsi), %xmm1
; S
vmovss  0x24(%rsi), %xmm0
; M
vmovss  0x10(%rsi), %xmm3
; x
vmovss  0x18(%rsi), %xmm2
; k
vsubss  %xmm0, %xmm3, %xmm5
vaddss  -128(%rip), %xmm2, %xmm4
; 1
vmovss  %xmm4, 0x18(%rsi)
; k
vdivss  %xmm4, %xmm5, %xmm6
vaddss  %xmm0, %xmm6, %xmm0
vmovss  %xmm0, 0x24(%rsi)
; M
vsubss  %xmm0, %xmm3, %xmm3
vmulss  %xmm5, %xmm3, %xmm0
vaddss  %xmm1, %xmm0, %xmm1
vmovss  %xmm1, 0x30(%rsi)
; S
vdivss  %xmm2, %xmm1, %xmm0
vmovss  %xmm0, 0x38(%rsi)
; V
```

Fig. 18. Machine code for SigadLib programs – Kahan (left); variance (right).

Figure 18 depicts the machine code that the JVM JIT compiler generated in the benchmark runs from our bytecode. Only the loop bodies are shown. The result is exactly as expected. Note that code such as the displayed can not be generated compositionally, due to the global constraints of register allocation.

It is easily seen that the machine instructions (from the AVX instruction set chosen by the JIT compiler) for the Kahan summation algorithm (left) correspond one-to-one to the operators in Figure 3.

For the variance algorithm (right), we have not given the flat Sig data-flow diagram. However, it is also fairly easy to map machine instructions to the operators of the formulae in Figure 11. The JIT compiler has even spotted that the subexpression $(x_k - M_{k-1})$ occurs twice, and eliminated the redundant computation.

Note that such optimizations are not supported at the Sig adLib level directly, since they would break compositionality. The programmer is of course free to do so manually, and share the outputs of a component explicitly.

6 Conclusion

We have presented Sig adLib, a lightweight embedded DSL for clocked synchronous data-flow programming on the Java platform. Its design and core APIs embody the semantic principles of the Sig project and stand-alone DSL, but allow the prospective data-flow programmer to tap directly into the resources of the Java ecosystem, as well as serving as a backend to the Sig frontend language. Sig adLib objects can be executed directly, forming a distributed interpreter, or compiled to JVM bytecode for direct access to the JVM.

The Sig adLib DSL is a new realization of ideas that we have discussed before. For thorough comparisons to related work, see the Sig papers [7, 9, 10, 12]. The relative virtues of lightweight and heavyweight DSLs for signal processing have been demonstrated lucidly in the context of the Feldspar language [1].

The distinguishing innovative feature of Sig adLib is its solution of the quest for compositionality. Sig adLib building blocks are compositional, that is descriptively self-sufficient and locally executable, in each of the two complementary aspects of data and control flow. This separation of concerns is both necessary, since their combination is not compositional for simple theoretical reasons, and very effective. It allows data-flow programs to be constructed in a reusable and modular way, leveraging the well-understood encapsulation means of the underlying object-oriented host language.

Compositionality is not merely a theoretical nicety and end in itself. It gives the language a "democratic" power: The only privileged part of the language is its collection of core APIs; all implementations, from predefined primitive operations and combinators to complex and application-specific user-defined components, are technically on par. This holds for direct interpretation as well as for compilation.

Sig adLib does not yet encompass all the potential features that we have investigated in the context of the Sig semantical framework. Of particular interest for future work are advanced control flow mechanisms, such as pattern-based case distinctions [8], higher-order streams [11], and multi-rate clocks [10].

References

[1] E. Axelsson et al. "Feldspar: A domain specific language for digital signal processing algorithms". In: *MEMOCODE*. IEEE Computer Society, 2010, pp. 169–178. DOI: 10.1109/MEMCOD.2010.5558637.

[2] J. Backus. "Can Programming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs". In: *Comm. ACM* 21.8 (1978), pp. 613–641.

[3] P. Caspi and M. Pouzet. "A Co-iterative Characterization of Synchronous Stream Functions". In: *Electronic Notes in Theoretical Computer Science* 11 (1998), pp. 1–21. DOI: 10.1016/S1571-0661(04)00050-7.

[4] W. Kahan. "Further remarks on reducing truncation errors". In: *Communications of the ACM* 8.1 (Jan. 1965), p. 40. DOI: 10.1145/363707.363723.

[5] D. Knuth. *The Art of Computer Programming*. 3rd Edition. Vol. 2. Addison–Wesley, 1997. ISBN: 978-0-201-89684-2.

[6] H. Nilsson, A. Courtney, and J. Peterson. "Functional Reactive Programming, Continued". In: *Proc. Haskell Workshop*. ACM, 2002, pp. 51–64. DOI: 10.1145/581690.581695.

[7] B. Trancón y Widemann and M. Lepper. "Foundations of Total Functional Data-Flow Programming". In: *Proc. MSFP*. Vol. 154. Electronic Proceedings in Theoretical Computer Science. 2014, pp. 143–167. DOI: 10.4204/EPTCS.153.10.

[8] B. Trancón y Widemann and M. Lepper. "Sound and Soundness: Practical Total Functional Data-flow Programming". In: *Proceedings 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*. FARM '14. Demo abstract. ACM, 2014, pp. 35–36. DOI: 10.1145/2633638.2633644.

[9] B. Trancón y Widemann and M. Lepper. "Towards Execution of the Synchronous Functional Data-Flow Language Sig". In: *Draft Proceedings 26th International Symposium on Implementation and Applications of Functional Languages (IFL 2014)*. Ed. by S. Tobin-Hochstadt. University of Indiana, 2014, pp. 71–80. URL: http://ifl2014.github.io/ifl-pre-proceedings.pdf.

[10] B. Trancón y Widemann and M. Lepper. "Laminar Data Flow: On the Role of Slicing in Functional Data-Flow Programming". In: *Proceedings Trends in Functional Programming (TFP 2015)*. Vol. 9547. Lecture Notes in Computer Science. 2015. ISBN: 978-3-319-39109-0. DOI: 10.1007/978-3-319-39110-6_5.

[11] B. Trancón y Widemann and M. Lepper. "The Shepard Tone and Higher-Order Multi-Rate Synchronous Data-Flow Programming in Sig". In: *Proceedings 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design (FARM 2015)*. ACM, 2015, pp. 6–14. DOI: 10.1145/2808083.2808086.

[12] B. Trancón y Widemann and M. Lepper. "Higher-Order Causal Stream Functions in Sig from First Principles". In: *Software Engineering (Workshops)*. Vol. 1559. CEUR Workshops. 2016, pp. 25–39. URL: http://ceur-ws.org/Vol-1559/paper03.pdf.

[13] B. Trancón y Widemann and M. Lepper. "LLJava: Minimalist Structured Programming on the Java Virtual Machine". In: *Proc. Principles and Practices of Programming on the Java Platform (PPPJ 2016)*. ACM, 2016. DOI: 10.1145/2972206.2972218.

Operatoren mit optionalen und wiederholten Syntaxteilen in MOSTflexiPL

Christian Heinlein
Studiengang Informatik
Hochschule Aalen – Technik und Wirtschaft

Die Syntax der Programmiersprache MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language, http://flexipl.info) kann von ihren Benutzern nahezu beliebig erweitert und angepasst werden, zum Beispiel:

```
maximum of <x:int> and <y:int> : int
{ if x > y then x else y end }
```

Hier wird ein Operator mit der Syntax `maximum of • and •` definiert (die Punkte repräsentieren die Positionen der Operanden), der das Maximum seiner beiden Operanden ermittelt und anschließend z. B. in der Form `maximum of 2 and 3` verwendet werden kann.

Um die Flexibilität der Sprache noch weiter zu erhöhen, können Syntaxteile optional deklariert werden, indem man sie (wie bei EBNF) in eckige Klammern setzt, zum Beispiel:

```
maximum [of] <x:int> [and] <y:int> : int
{ if x > y then x else y end }
```

Damit sind jetzt auch Verwendungen wie z. B. `maximum 2 3` oder auch `maximum of 2 3` möglich.

Ein Maximum von mehr als zwei Werten kann natürlich durch verschachtelte Anwendungen des Operators ermittelt werden, z. B. `maximum of 2 and maximum of 3 and 4`, was jedoch relativ umständlich ist. Wesentlich einfacher und natürlicher wäre z. B. `maximum of 2 and 3 and 4`, d. h. der Syntaxteil `and•` sollte beliebig oft wiederholt werden können. Dies lässt sich, wiederum in Anlehnung an EBNF, mit geschweiften Klammern ausdrücken, die auch beliebig mit eckigen Klammern kombiniert werden können:

```
maximum [of] <x:int> { [and] <y:int> } : int
{
    max : int?; max != x;
    repeat
        if y > ?max then max != y end
    end;
    ?max
}
```

Zur Implementierung der Funktionalität wird eine Variable `max` mit Typ `int?` definiert (Variablentypen werden durch ein Postfix-Fragezeichen ausgedrückt) und mit dem Wert des Parameters `x` initialisiert (der Zuweisungsoperator hat die Syntax `•!=•`). Anschließend wird mit dem vordefinierten Operator `repeat•end` die Liste der Werte des wiederholten Parameters `y` durchlaufen, wobei `y` in jedem Durchlauf einen dieser Werte besitzt. Wenn der Wert von `y` im aktuellen Durchlauf größer als der momentane Wert der Variablen `max` ist (der Wert einer Variablen wird durch ein Präfix-Fragezeichen abgefragt), wird deren Wert entsprechend angepasst. Abschließend wird der Wert der Variablen zurückgegeben.

Weitere interessante Anwendungen für optionale und (bei Bedarf auch verschachtelte) wiederholte Syntaxteile, die im Vortrag gezeigt werden sollen, sind Kontrollstrukturen wie z. B. `if • then • { elseif • then • } [ else • ] end` und `branch • { if • { or • } then •} [ else • ] end`.

Rewriting For Parametrization
(Bad Honnef 2018)

Markus Lepper[1] and Baltasar Trancón y Widemann[2]

[1] <semantics/> GmbH, Berlin
post@markuslepper.eu
[2] baltasar@trancon.de

Abstract. With a modular architectures of source text objects, a method of free rewriting can be more adequate than pre-wired parameters: each module import can be accompanied with a set of rewriting rules, mapping references to expressions and thus adapting the imported definitions to the needs of the importer. Such a mechanism is described in this paper; it has been implemented in the d2d system applied to document types, but seems applicable also for other architectures with a "glass box" view to modules.
There are two fundamentally different use cases: rewriting (a) to algebraic semantics, e.g. for controlling a parsing process, and (b) reifying the rewritten results, for documentation, statistics, automated user dialogue, etc. The second case needs much more effort than the first. For both cases an algorithm is presented, which is designed for optimized performance, and which reflects details rising from experiences in practical applications.
The transfer to other languages is supported by a division of the second algorithm into a generic and a specific part.

1 Introduction

1.1 The Principle of Rewriting for Parametrization

The method explored in this paper, called *rewriting for parametrization*, has been developed in one particular context, but is applicable to others with the same basic settings. These are:

1. Given is a collection of *definitions*; each assigns an *expression* in some well-defined language to an *identifier*. These expressions refer to (other) definitions by quoting their identifiers. The language is called *underlying domain language* in the following.
2. These definitions are grouped in source text *modules*. One module can *import* one or more modules, for referring the definitions contained therein.
3. Every imported module can be subject to *parametrization*. This alters the imported definitions slightly, according to the needs of the importer.

It is obvious that these points are met by many and very diverse source text architectures: computer progam sources, document type definitions, rule sets for inference systems or description logic, etc. The original context was *d2d*, a front-end for writing down XML encoded texts in the flow of authoring, and the underlying domain language defines content models for XML element types and related front-end representation character parsers.

The practical requirements in this project made it impossible to foresee every possibly later parametrization needs, already when defining a library module for later re-use. Instead, a general mechanism was implemented which allows to *freely rewrite any definition of an imported module*, modifying them to the needs of the importer without restrictions. This immediately implies a further criterion on the overall architecture:

4. The modules and their contents must be known to the importer, following a *glass box discipline*: The internal structure of the imported module must be known to be sensibly rewritten.

This last condition certainly cuts down the field of applicability for rewriting significantly. But with any system of document type definitions it is fulfilled: The user wants to construct text models which adhere to the structure defined in the module, so they must know its internal structure anyhow. Similar is the situation with the diverse rule sets mentioned above.

Coupling rewrite rules with module imports allows to adopt a whole collection of definitions in an atomic and consistent way. Basically, this mechanism is easy to understand, straight-forward to implement and sufficiently expressive, as our tool has proven in production.

The intended data structure is always the result of *resolving the rewriting rules* and inlining the resulting definitions. Technically, two cases must be distinguished: *First*, the results are used with *algebraic semantics*, e.g. to steer the parsing process, as in the case of d2d. This is relatively easy to achieve: All unfolding can be done in a "naive" way, because (a) the input to parse naturally limits the expansion process and prevents explosion, and (b) creating different copies with the same algebraic behaviour does in no way influence the results.

Much more complicated is the *second case*, when the results of rewriting shall not only be applied, but also *reified* and *identified*. This is necessary e.g. for automated generation of *user documentation*, (a central issue in each project which aims at domain experts not necessarily language experts) and of execution statistics. In these cases, an explosion of equivalent copies due to different but irrelevant import statements must be avoided, and an ergonomically sensible and computable notion of *equivalence* of the parametrized definitions is required. This case is much harder to deal with; our algorithm establishes a clean separation of its generic part and of the specifics of the underlying domain language.

The contribution of this article is a *practical* one, in two concerns: First, for both cases, the algebraic and the reified, we give operational semantics, which are immediately executable and designed for optimized performance. Second, many design decision have come from concrete needs in daily practice and will be mentioned as such, whenever relevant.

1.2 Structure of the Article

Section 2 constructs a mathematical model of definitions and modules, and defines the rewriting process in its pure algebraic form, as needed e.g. for parsing control. Section 3 discusses the additional problems of reification, and enhances the algorithm accordingly. Section 4 presents future and related work.

2 Rewriting for Algebraic Application

2.1 Definitions and Modules

The underlying domain language in the case of d2d is a collection of definitions which assign XML content models (plus possibly character parsing rules) to types and tags of XML elements. The language is given as the data type T in Table 1: Only the first two lines are significant for the rewriting process; the further lines, printed in gray, are just for illustration how an underlying domain language can look like. The content models given by these definitions steer the d2d parsing process. More precisely: The *expanded* versions do so, after resolving all parametrization rewritings of module imports.

Every *(source text) module* maps *identifiers* to *expressions*, see the schema types *Module* and *Definition* in Table 1.[3][4] The basic atoms of the expressions are *references* from R. These are either simple identifiers from $\mathbb{I}_T$, and refer to the definitions contained in the very same source module as the expression, or prefixed with one or more *import keys* from $\mathbb{I}_I$. Then they refer to a definition in an *imported module*, see the following example.

2.2 Import and Parametrization

Beside definitions, every module can contain import clauses, modeled by the schema type *Import* in Table 1. For each of these (1) an import key is defined (from $\mathbb{I}_I$), to be used as a prefix for referring to the definitions in the imported module, (2) its source text is identified by an absolute module address from $\mathbb{I}_A$, and (3) three different kinds of *rewrite rules* may be applied to this source:

Global rewritings replace every occurrence of a particular reference everywhere in the imported module by the given *replacement expression*. This applies to both kinds of expressions in the imported module, namely in element contents definitions and in replacement expressions in import clauses. The replacement expression will be evaluated in the context of the importing module. The reference to be replaced is interpreted verbatim, as a source text component.

[3] The mathematical notation applied is summarized in Appendix A.

[4] The data type "Definition" appears here as a superfluous wrapper around "Expression", but in a practical implementation further information may be attached to its instance objects.

Local rewritings only affect one single definition in the imported module, as indicated by its identifier. In its expression, the given reference is replaced by the replacement expression in the same way as with a global rewriting. A local rewriting overrides the global rewriting for the same reference.

Import rewritings replace the whole import clause in the imported module with the given import key by an import clause defined in the importing module.

In the d2d format this looks like

```
1  module uses_other_modules
2    import M from module_to_import
3           ^( (M.a, M.J.L.b, c)* / MODULE_PARAMETER )
4           ^( N / H )
5    import N from another_module
6           in a ^( (N.a, N.a) / a )
7           in b ^( a / a )
```

This example module imports a first module (the source of which is found under the name "module_to_import") and replaces in its source, according to the rewrite rule in line 3, all references to "MODULE_PARAMETER" by the given expression. This in turn contains a reference "M:a" to the imported and parametrized module itself. Please note that "MODULE_PARAMETER" is syntactically the same as a simple identifier "a"; its appearance has been chosen only to indicate to the user that it is *foreseen* to be rewritten: Parametrization in the narrow sense is subsumed as a special case of rewriting.[5]

The second import rewrites the regular expressions of the definitions "a" and "b" in the imported module. In "a" it rewrites every reference to itself to a sequence of two of these (line 6). In "b" it rewrites all references to "a" to a definition named "a" in the importing module (line 7): In "a/a" the second term is the reference to be replaced, as it stands unevaluated in the source representation of the imported module; the first term is the replacement expression, evaluated in the context of the importing module. (Which therefore must contain a definition for "a" somewhere, or "a" must be re-written to something defined, when uses_other_modules is imported itself.)

In most concrete fields of application, *implicit re-exports* turned out to be indispensable: In line 3 above "M.J.L.b" does refer to a definition in an imported module of an imported module of an imported module; another aspect of the glass box approach. E.g. in the context of our LaTeX-like d2d_gp architecture, an instantiation of the general purpose article for a particular technical documentation will distribute the required additional in-line elements (= technical terms, abbreviations, symbols, trade marks, etc.) not only to the flow text of paragraphs, but also into the content models of captions of tables and figures, into titles, hypertext anchors, footnotes, etc. All these are contained in imported modules. Whenever further modification of the resulting top level

[5] The d2d implementation allows to declare a definition as "#GENERIC", so that all references to it *must* be rewritten.

$$
\begin{array}{l}
T ::= R \mid \texttt{@}\,T \mid T\,\texttt{\^{}}\,(\,T\,/\,R\,) \mid \#\texttt{none} \\
\quad \mid (\,T\,) \mid T\,\texttt{,}\,T \mid T\,\&\,T \mid T\,|\,T \mid T\,\texttt{?} \mid T\,\star \mid T\,\texttt{+} \\
\quad \mid T\,\tilde{}\,T \mid T\,\tilde{}\,\star \mid T\,\tilde{}\,\texttt{+} \mid [\,\mathbb{I}_T\,T\,] \mid (\texttt{>}\,T\,) \\
\quad \mid \texttt{'}\,\mathit{chars}\,\texttt{'} \mid \texttt{"}\,\mathit{chars}\,\texttt{"} \mid \texttt{0x}\,\mathit{hhhh} \\
\quad \mid T \cup T \mid T \mathbb{A} T \mid T - T \mid T\,\texttt{..}\,T
\end{array}
$$

$\mathbb{I}_I$ // *ids used as import key. Examples use "K", "L", etc.*
$\mathbb{I}_A$ // *ids used as module source location. Examples use "my_mod", etc.*
$\mathbb{I}_T$ // *names and tags of definitions. Examples use "a", "b", etc.*

$R = \mathbb{I}_I{}^\star \times \mathbb{I}_T$ // *Relative reference, as appearing in source text.*
$\overline{S} = \mathbb{I}_I{}^\star$ // *Completely expanded, "dynamic" path, relative to* top
$S = \mathbb{I}_I{}^\star \times \mathbb{I}_T$ // *idem.*

$$
\begin{array}{lll}
\mathit{Module} & \triangleq & [\mathit{imps} : \mathbb{I}_I \rightharpoonup \mathit{Import} \;;\; \mathit{defs} : \mathbb{I}_T \rightharpoonup \mathit{Definition}\,] \\
\mathit{Import} & \triangleq & [\mathit{addr} : \mathbb{I}_A\,; \\
 & & \quad \mathit{globalRews} : R \rightharpoonup T\,; \mathit{localRews} : \mathbb{I}_T \rightharpoonup R \rightharpoonup T\,; \mathit{impRews} : \mathbb{I}_I \rightharpoonup \mathbb{I}_I\,] \\
\mathit{Definition} & \triangleq & [\mathit{tag} : \mathbb{I}_T \;;\; \mathit{regExp} : T\,]
\end{array}
$$

Table 1. Grammar and Data Types for d2d Text Structure Definitions.

format is necessary, all these now correctly instantiated substructures shall also be immediately addressable for reuse and rearrangement. This is achieved by implicit re-export.

The third kind of rewriting is that of a whole import clause. It has been introduced especially for multi-lingual support: All structure definitions e.g. for calendaric date and time, or for personal names, or for postal addresses, are exchanged together, by one single rewrite rule, switching between whole modules. Currently this feature is only defined between "direct aunt and nephew": The common ancestor imports two modules and replaces in the first one (in the example "M") one of its internal module imports ("H") by its own second import ("N", see line 4). This could of course be easily extended to a free compositional device, which is always desirable in the theoretical perspective, but does not seem to make much sense in practice, – it soon gets too confusing.

2.3 Resolving Imports and Parametrizations

Table 2 shows the two start values which are kept constant for the evaluation process: The library of modules module maps module source adresses to data objects representing source text modules; the value top gives the type of the root element as a product of the source adress of a module and the identifier of a definition therein.

Every module import involved in the resolution process is contained in the "*imps*" table of the top level module, or in that of another module, directly or indirectly imported by the top module. So every module import (=instantiated

$$
\begin{array}{rll}
\multicolumn{3}{l}{\textit{// global constants, fixed per run:}} \\
\mathsf{module} : \mathbb{I}_A \rightharpoonup \mathit{Module} & // & \textit{Library of all module source texts.} \\
\mathsf{top} : \mathbb{I}_A \times \mathbb{I}_T & // & \textit{Top-most text structure element.}
\end{array}
$$

$$
\begin{array}{rll}
\multicolumn{3}{l}{\textit{// navigating sources, relative to } \mathsf{top}\textit{:}} \\
\mathit{import} : \overline{S} \rightharpoonup \mathit{Import} & // & \textit{Source text import statement for dynamic path.} \\
\mathsf{module}' : \overline{S} \rightharpoonup \mathit{Module} & // & \textit{Complete module source text for dynamic path.} \\
\mathit{sourceAddr} : S \rightharpoonup (\mathbb{I}_A \times \mathbb{I}_T) & // & \textit{Source text address for dynamic path.} \\
\mathsf{def} : S \rightharpoonup \mathit{Definition} & // & \textit{Defintion source text for dynamic path.} \\
\mathit{base} : \overline{S} \rightharpoonup \overline{S} & // & \textit{Shorten dynamic path.}
\end{array}
$$

$$\mathit{import}(\langle\rangle) = (\mathsf{top}.1, \{\}, \{\}, \{\})$$

$$\frac{p, q : \mathbb{I}_I \qquad \mathit{import}(\pi \blacktriangleleft p) = (a, \_, \_, i)}{\mathit{import}(\pi \blacktriangleleft p \blacktriangleleft q) = \begin{cases} \mathit{import}(\pi \blacktriangleleft q') & \textbf{if} (q \mapsto q') \in i \\ \mathsf{module}(a).\mathit{imps}(q) & \textbf{otherwise} \end{cases}}$$

$$
\begin{aligned}
\mathsf{module}'(i) &= \mathsf{module}(\mathit{import}(i).\mathit{addr}) \\
\mathit{sourceAddr}(i, t) &= (\mathit{import}(i).\mathit{addr}, t) \\
\mathsf{def}(i, t) &= \mathsf{module}'(i).\mathit{defs}(t) \\
\mathit{base}(J \blacktriangleleft j) &= J
\end{aligned}
$$

Table 2. The Two Top-Level Constants and Auxiliary Navigation Functions.

module) can be identified by the sequence of import keys from $\mathbb{I}_I{}^*$, which adresses it from the viewpoint of the top module. We call this its *dynamic path*. It is modelled by the type $\overline{S}$ from Table 1. [6] An instantiated definition in this module is identified by this dynamic path plus its local identifier from $\mathbb{I}_T$, the data type S.

Static we call all facts which follow from the source text of each module alone, per se. *Dynamic* are those which arise after selecting top, by following all imports and references from this starting point, accumulating all rewriting rules and applying them.

Dynamic paths may become *infinite* in case of cyclic imports. In most cases, such an infinity does not disturb the resolution process, since a fix point w.r.t. the resolution of whole modules is never required, but only for *reachable definitions*, which is nearly always found, as discussed in Section 3.4.

The auxiliary navigation function *import*(..) resolves a dynamic path to the corresponding *Import* object and resolves the import rewritings as follows: The import clause $(a, \_, \_, i)$ has been found for the dynamic path $\pi \blacktriangleleft p$. If this clause contains a rewriting for q in the import rewritings i, then we follow this other import clause instead; else we take verbatim the import clause named q in

[6] This type is formal identical with R. But R stands for a reference relative to a source text context, while S gets its meaning globally, starting from top.

// Stack of currently active local expression level rewritings:
$K_1 = (R \times T)^\star$
// stack of frozen inlining (="@") contexts:
$K_2 = ((K_1 \times \overline{S} \times \mathsf{opt}\,\mathbb{I}_T) \cup \{\nabla\})^\star$

$\mathsf{visit}[E] : K_1 \times \overline{S} \times \mathsf{opt}\,\mathbb{I}_T \times K_2 \times T \times E \to (E \cup \{\texttt{Error}\ldots\})$
$\mathsf{action}_\square[E] : \overline{S} \times \mathbb{I}_T \times E \to (E \cup \{\texttt{Error}\ldots\})$

$\mathsf{visit}(k, J, d, m, r_1\ \texttt{\^{}}\ (\ r_2\ /\ j\ )\ , e) = \mathsf{visit}(k \blacktriangleleft (j \mapsto r_2), J, d, m, r_1, e)$
$\mathsf{visit}(k, J, d, m,\ \texttt{@}\ x, e) = \mathsf{visit}(k, J, d,\ (m \blacktriangleleft (k, J, d) \blacktriangleleft \nabla)\ , x, e)$
$\mathsf{visit}(k, J, d, m, (i,t) : R, e)$

$$= \begin{cases} \mathsf{visit}(k', J, d, m, x, e) \\ \qquad \textbf{if } k = k' \frown \langle (i,t) \mapsto x \rangle \frown k'' \wedge \quad k'' \neq \ldots \frown \langle (i,t) \mapsto \_ \rangle \frown \ldots \\ \mathsf{visit}(\langle\rangle, \mathit{base}(J), \bot, m, x, e) \\ \qquad \textbf{elseif } ((i,t) \mapsto x) \in (\mathit{import}(J).\mathit{globalRews} \oplus \mathit{import}(J).\mathit{localRews}(d)) \\ \mathsf{visit}(\langle\rangle, J \frown i, t, m', \mathsf{def}(J \frown i, t).\mathit{regExp}, e) \\ \qquad \textbf{elseif } m = m' \blacktriangleleft \nabla \\ \mathsf{visit}(k', J', d', m', (i,t), e) \\ \qquad \textbf{elseif } m = m' \blacktriangleleft (k', J', d') \\ \mathsf{action}_\square(J \frown i, t, e) \qquad \textbf{otherwise} \end{cases}$$

$$\boxed{\mathsf{action}_A(i,t,e)} = \begin{cases} X & \textbf{if } X = \mathsf{check}(i,t) \neq \mathit{true} \\ \mathsf{visit}(\langle\rangle, i, t, \langle\rangle, \mathsf{def}(i,t).\mathit{regExp}, e) & \textbf{otherwise} \end{cases}$$

$\mathsf{check}(i,t)$

$$= \begin{cases} \texttt{Error, no import key } i & \textbf{if } i \notin \mathsf{dom}\ \mathit{import} \\ \texttt{Error, no module at } a & \textbf{if } a = \mathit{import}(i).\mathit{addr} \notin \mathsf{dom}\ \mathsf{module} \\ \texttt{Error, no definition at } (a,t) & \textbf{if } t \notin \mathsf{module}'(i).\mathit{defs} \\ \mathit{true} & \textbf{otherwise} \end{cases}$$

$$\frac{\square \in \{\ \texttt{,}\ ,\ \texttt{|}\ ,\ \texttt{\&}\ ,\ \texttt{\~{}}\ ,\ \mathbb{U}\ ,\ \mathbb{A}\ ,\ -\ ,\ \texttt{..}\ \} \qquad \mathsf{visit}(k, J, d, m, r_1, e) = e_1}{\mathsf{visit}(k, J, d, m, r_1\ \square\ r_2, e) = \mathsf{visit}(k, J, d, m, r_2, e_1)}$$
$$\mathsf{visit}(k, J, d, m, (\alpha \square \#\mathtt{none} \square \beta), e) = \mathsf{visit}(k, J, d, m, (\alpha\ \square\ \beta), e)$$

$$\frac{\bigtriangleup \in \{\ \star\ ,\ +\ ,\ ?\ ,\ \texttt{\~{}}\star\ ,\ \texttt{\~{}}+\ \} \qquad t \in \mathbb{I}_T}{\mathsf{visit}(k, J, d, m, r\ \bigtriangleup, e) = \mathsf{visit}(k, J, d, m,\ [\ t\ r\ ]\ , e)}$$
$$= \mathsf{visit}(k, J, d, m, >\ r, e) = \mathsf{visit}(k, J, d, m, r, e)$$

$$\frac{\gamma \in \{\ \texttt{"}\,\alpha\,\texttt{"}\ ,\ \texttt{'}\,\alpha\,\texttt{'}\ ,\ 0\texttt{x}\mathit{hhhh}, \#\mathtt{none}\}}{\mathsf{visit}(k, J, d, m, \gamma, e) = e}$$

Table 3. Rewriting of Source Definitions With Mere Algebraic Semantics.

the source text indicated by a. This function is employed by $\mathsf{module}'(..)$ to find the source text for a dynamic path, and by $\mathsf{def}(..)$ to find the source text of a definition for a dynamic path plus a local identifier.

In the expression language T from Table 1, the construct `@` x means the *inlining* or *flattening* of the content models of all definitions referred to in x. This mechanism allows the same definition to be used as XML element content model, or only as a constant with a regular expressions as its value, or in both roles. All substitutions must be applied to the expression x before flattening, and to all expressions inserted by flattening.[7]

The construct e `^ ( ` f ` / ` r `)` is an explicit rewriting on expression level, of all identifiers r in the expression e by the replacement expression f. This is useful especially in combination with the insertion operator, as in

```
(@a,@b) ^ ( (x,y*) / c )
```

or as a shorthand notation for repetition, as in

```
(x,x,x) ^ ( (a,(b|c)*,d) / x )
```

The rewriting to $f = \#\texttt{none}$ mimics "physically erasing" the visual representation of the reference r in the input source text, together with the adjacent operators. Its semantics are context sensitive: E.g. when rewriting a grammar regular expression, in a sequence context it corresponds to "ε", the empty sequence which is always accepted and delivers nothing, while in an alternative it corresponds to "*fail()*", a parser which never matches.[8]

The top module is not rewritten, and the definition adressed by top is taken verbatim as the start point of resolution. The resolution process is modelled in Table 3 as a *visitor process*. By calling $\mathsf{visit}(\langle\rangle, \langle\rangle, \bot, \langle\rangle, (\langle\rangle, \mathsf{top}.2), \_)$, all reachable definitions are expanded for further processing with algebraic semantics by the function $\mathsf{visit}(k, J, d, m, x, e)$. The arguments in this function signature are "vistior registers", which keep track of the rewriting rules which are accumulated in descending. They stand from left to right with decreasing priority:
k is a storage for local rewritings on expression level;
J is the dynamic path of the containing instantiated module;
d is the identifier of the currently visited definition, or $\bot$ if x lives on module level (= in a replacement expression of an import clause);
m is a further stack for storing the situation of all three preceding stacks, for to apply them twice, before and after inlining;
x is the expression to expand and visit (for e see below).

The lower rules in Table 3 are specific for the underlying domain language and show how expressions with binary and unary operators are visited by simply visiting their argument(s) and feeding through the parameter e.

[7] Please note that together with unlimited cylic usage, this `@` operator causes the transition from regular to context free languages, and thus should be used with care. Not all problems arising from unlimited usage have currently been solved in our implementation.

[8] This design is highly "irregular" under pure algebraic view and has again been chosen for practical reasons: Whenever all apperances of a reference shall be erased, this is inteded (in most cases) in alternatives and sequences simultanuously.

The top part of the Table shows the rules of general interest: Descending into an explicit expression level rewriting $r_1 \;\hat{}\; (\, r_2 \,/\, j \,)$ appends the pair of identifier and replacement expression to the end of stack k and then descends into r_1;
descending into a flattening expression $@\, x$ stores the whole resolution context plus a marker symbol ∇ at the end of stack m and then descends into the expression x.

The central rule is that which visits a reference $(i, t) : R$, i.e. a local identifier plus possibly a prefix of import indentifiers. If there is a replacement for this reference in the stack of expression level rewritings, than this stack is shortened and the replacement expression is visited.
Otherwise: if there is a rewriting statement for this reference found in the import clause of the current module, then visit the replacement expression in the context of the importing module. (A specific rewriting expression for the current definition has priority over a global one, see the operator $\oplus$; only here the parameter d is needed.)
Otherwise: if there is a marker ∇ at the end of the K_2 stack then remove it and visit the expression of the definition with name t in the module found by the import prefix list i. This realizes the inlining, required by the operator "@".
Otherwise: if the K_2 stack is not empty, then pop one entry, reconstruct from it the resolution contexts k, J and d, and visit again the reference.
The last two rules co-operate to apply every substitution twice: if an identifier stands under a flatten operator @, first all accumulated substitutions are applied to it, as such. If none are left, the targeted definition is inlined and the marker is removed from the stack. If now a identifier from the inlined expression does not have a local rewriting on it, the original source context of the @ expression is reconstructed from the stack, and all rewritings are applied again, to the reference in the inlined expression.[9]

Finally, if no single rewriting is applicable, the reference is resolved to a definition by concatenating the local import prefix with the dynamic path of the current module J, and $\mathsf{action}_{\square}(..)$ is called. Here the code can be plugged in which is to apply to the reference after all rewritings have been resolved.

[9] This solution comes again from practical experiences: Assume a content model definition like "`tags article = ..., @ table, ...`". Then these two kinds of rewritings are both sensible:

- "use the content model of `article`, replacing (all references to and) all insertions of `table` by those of `mytable`;"
- "use `article`, but in the inserted contents of `table` replace `caption` by `#none`."

It seems that separate language elements for both kinds of rewriting requests either become very complicated or must give up compositionality. Furthermore, we did not find a use case which required to execute a replacement in one phase, and to suppress it explicitly in the other. So we could unify both meanings and keep the language front-end small.

The standard implementation $\mathsf{action}_A(i,t,e)$ simply checks whether the combination of dynamic path and local identifier leads to a definition, and, if yes, enters visit(..) for its expression recursively. Thus it performs a context check for the whole structure starting at top.[10]

In all these functions, the argument e of the generic parameter type E is some additional accumulator parameter fed through all visits, easily implemented as additional visitor field variables. It is not needed in the pure algebraic usage of the visitor in this stage and simply passed through, but will be employed in the context of the next section.

The collection of function definitions from Table 3 is complete, covers common static error cases as detected by check() and thus defines a precise operational semantics for the rewriting process. This is totally independent from further properties of the underlying domain language, here: T.

When for $\mathsf{action}_\square$ some simple variant of action_A is selected, then this is sufficient for the purpose of controlling a parsing process in the original d2d context: Whenever it is called, an input token is tested for consumption, a new XML result element is possibly opened or closed, etc.[11]

Two problems do not arise: (A) cycles and thus infinite structures are not an issue, because the parsing process matches them against a *finite* input, or at least against an always finite prefix of the input, which limits all access operations in a natural way. (B) duplications are not an issue: As usual in evaluation systems of this kind, only the algebraic semantics are exploited. One and the same definition may be unfolded and instantiated arbitrarily often, because the effect of all these copies will implode again when the result, here: the parse tree, is finally delivered.

3 Rewriting for Reification

The situation changes fundamentally as soon as the expanded definitions shall be "reified" and treated as identifiable objects, as it is frequently necessary in practice for error diagnosis, user documentation, collection of statistic data, etc.

For instance, the current implementation of d2d administers polyglot documentation for every XML element of a text structure in form of a hyper text, using d2d recursively to document itself. These texts naturally are defined for the unexpanded sources, and a computer scientist or language expert can possibly profit and "link" them mentally. But the addressed domain experts are much more helped with a documentation of the *fully expanded* structure definition, which tells them directly where certain tags are allowed, required, forbidden, etc., and what their role is in this particular context.

[10] This can lead to infinite recursion; in the general case, cylce detection must be provided.

[11] Of course the different constructors in the underlying domain language (see the grayed-out lines in the definition of T as an example) must also be treated specially, each according to its semantics, when reaching the corresponding visit() function. This is left out here and described in detail in [2].

In all these cases, in contrast to the expansion process as described in the preceding section, the above-mentioned issues arise: (A) the expansion is not limited by the input data, therefore infinite structures *must* not occur, and (B) multiple equivalent expansions of the same source definition *should* not occur, for not confusing the user. Consequently, the properties of the underlying domain language, here: T, can no longer be totally abstracted from, as it was possible in the preceding section. The finiteness of the recursion depends on the definition of equality, and any notion of equivalence is related to the intended semantics and to the pragmatic requirements from a user's point of view.

Nevertheless, this influence is limited and can be isolated. For this purpose, the following algorithm is given as a framework of two layers: The upper one steps through definitions, using a generic strategy, and is presented in Tables 4 and 5. It is independent of the properties of the underlying domain language. These come into play on the lower level, which compares expressions, see Table 6.

3.1 Comparing Definitions

Problem (B) is easier to deal with, and a typical unification problem. We search for the greatest fixpoint: All different instantiations of the same source text definition for which no counterexample is found are put into the same equivalence class. A counterexample is any sub-structure which shows a difference. These equivalence classes are constructed by recursive descent, similar to the visiting process above, enriched by very simple backtracking. In contrast to the algebraic case, here two (2) definitions or expressions are visited in parallel. During this descent, hypotheses of equivalence are collected.

This process is simplified significantly by two properties of the overall equation system: First, there is no negation, i.e. the addition of hypotheses is monotone. Second, there are not many alternatives, i.e. whenever the hypothesis of two instantiations being equivalent is added, there is only one single way to prove it, namely to show the equivalence of their complete instantiated substructures in a one-to-one fashion – there are no choice points with a finer granularity than complete definition instantiations and their equivalence class, save those induced by properties of the underlying domain language, see next section.

For easier discussion and implementation, the visitor code from the preceding section is used for "cloning", i.e. for generating a static copy of the visiting results, which are all traversed definition and expression objects. In this "expanded copy", all reference values are from S, not from R, which means that they point directly to the instantiated copies of the rewritten definitions. (The grammar definition from Table 1 must be adjusted accordingly.) This decouples the process of replacement resolution, as described above, from the classification process described in this section, and allows to discuss this by direct functions, which otherwise must be "folded by" the visiting and rewriting resolution process from the preceding section.[12]

[12] This approach requires that no cycle of refences contains the "`@`" operator, since cycle detection is not yet covered.

$$
\begin{aligned}
state &= \{\texttt{hypo}, \texttt{own}, \texttt{equ}, \texttt{failed}\} \\
J &\triangleq [\ state : State \ \times \ other : \mathsf{opt}\,\overline{S} \\
&\qquad \times\ different : \mathbb{F}\,\overline{S} \ \times \ depending : \mathbb{F}\,S\ ] \\
Js &= (\mathbb{I}_A \times \mathbb{I}_T) \to \overline{S} \nrightarrow J \\
E_J &\triangleq [\ classes : Js \quad \times \quad pending : \mathbb{F}\,S \quad \times \quad hypo : S\ ]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{setState} &: E_J \times S \times State \times \mathsf{opt}\,\overline{S} \to E_J \\
\mathsf{getEntry} &: E_J \times S \to \mathsf{opt}\,J \\
\mathsf{isUnbound} &: E_J \times S \to boolean \\
\mathsf{foundClasses} &: E_J \times S \to \mathbb{F}\,\overline{S} \\
\mathsf{getRepr} &: E_J \times S \to \overline{S} \\
\mathsf{linkHypos} &: E_J \times \overline{S} \times \overline{S} \times \mathbb{I}_T \to E_J
\end{aligned}
$$

$$
\frac{\begin{array}{c} c = e.classes(sourceAddr(i,t)) \\ c(i) = (\_, o, m_1, m_2) \\ H = \begin{cases} (i,t) & \textbf{if } s = \texttt{hypo} \\ e.hypo & \textbf{otherwise} \end{cases} \quad M = \begin{cases} m_1 \cup \{o\} & \textbf{if } s = \texttt{failed} \\ m_1 & \textbf{otherwise} \end{cases} \\ P = \begin{cases} e.pending \cup \{(i,t)\} & \textbf{if } s = \texttt{failed} \\ e.pending & \textbf{otherwise} \end{cases} \end{array}}{\begin{array}{c} \mathsf{setState}(e, (i,t), s, j) \\ = (\ e.classes \oplus (sourceAddr(i,t) \mapsto (c \oplus \{i \mapsto (s, j, M, m_2)\})),\ P,\ H) \end{array}}
$$

$$
\frac{c = e.classes(sourceAddr(i,t))(i)}{\mathsf{getEntry}(e,(i,t)) = c}
$$

$$
\mathsf{isUnbound}(e,(i,t)) = (c = \bot)\ \vee\ (c.state = \texttt{failed})
$$

$$
\mathsf{getRepr}(e,(i,t)) = \begin{cases} i & \textbf{if } c.state = \texttt{own} \\ c.other & \textbf{otherwise} \end{cases}
$$

$$
\mathsf{foundClasses}(e,(i,t)) = \{j | (j \mapsto c) \in e.classes(sourceAddr(i,t))\ \wedge\ c.state = \texttt{own}\ \}
$$

$$
\frac{\begin{array}{c} X = sourceAddr(i,t) \\ H_\Box = \begin{cases} \{e_3\} & \textbf{if } \mathsf{getEntry}(e,(\Box,t)).state = \texttt{hypo} \\ \{\} & \textbf{otherwise} \end{cases} \\ e\ =\ (\{\ldots, X \mapsto \{\ldots, (i \mapsto (i_1,i_2,i_3,i_4)), \quad (j \mapsto (j_1,j_2,j_3,j_)) \quad \ldots\}, \ldots\}, e_2, e_3\ ) \\ e'\ =\ (\{\ldots, X \mapsto \{\ldots, (i \mapsto (i_1,i_2,i_3,i_4 \cup H_i)), (j \mapsto (j_1,j_2,j_3,j_\cup H_j))\ldots\}, \ldots\}, e_2, e_3\ ) \end{array}}{\mathsf{linkHypos}(e,i,j,t) = e'}
$$

Table 4. Reification I, Auxiliary Data Types and Functions

$$
\begin{array}{ll}
\mathsf{cmpExp} & : T \times T \times E_J \to (E_J \times \mathit{boolean}) \\
\mathsf{cmpDef} & : \overline{S} \times \mathbb{I}_T \times E_J \times \overline{S} \to (E_J \times \mathit{boolean}) \\
\mathsf{action}_B(i,t,(e_1,e_2,e_3)) & = (e_1, e_2 \cup \{(i,t)\}, e_3) \\
\mathsf{classifyAll}((e_1,\{\},e_3)) & = (e_1,\{\},e_3) \\
\mathsf{classifyAll}((e_1,e_2 \cup \{(i,t)\},e_3)) & = \mathsf{classifyAll}(\mathsf{classify}(i,t,(e_1,e_2,e_3)))
\end{array}
$$

$$
\mathsf{classify}(i,t,e) = \begin{cases} e & \textbf{if } \neg\,\mathsf{isUnbound}(e,(i,t)) \\ \mathsf{ta}(i,t,e,\mathsf{foundClasses}(e,i,t)) & \textbf{otherwise} \end{cases}
$$

$$
\mathsf{ta}(i,t,e,\{j\} \cup \sigma) = \begin{cases} \mathsf{ta}(i,t,e,\sigma) & \textbf{if } j \in \mathsf{getEntry}(e,(i,t)).\mathit{different} \\ e' & \textbf{if } \mathsf{cmpDef}(i,t,e,j) = (e',\mathit{true}) \\ \mathsf{ta}(i,t,e',\sigma) & \textbf{otherwise} \end{cases}
$$

$$
\mathsf{ta}(i,t,e,\{\}) = \mathsf{classifyAll}(\mathsf{setState}(e,(i,t),\mathtt{own},\bot))
$$

$$
\frac{\mathsf{cmpExp}(\mathsf{def}(i,t).\mathit{regExp}, \mathsf{def}(j,t).\mathit{regExp}, \mathsf{setState}(e,(i,t),\mathtt{hypo},j)) = (e',b)}{\mathsf{cmpDef}(i,t,e,j) = \begin{cases} e' & \textbf{if } b = \mathit{true} \\ \mathsf{propagateFail}(i,t,e') & \textbf{otherwise} \end{cases}}
$$

$$
\frac{e' = (\ \mathsf{setState}(e,(i,t),\mathtt{failed})).\mathit{classes}, e.\mathit{pending} \cup \{(i,t)\}, e.\mathit{hypo}\ )}{\mathsf{propagateFail}(i,t,e) = \mathsf{propagateFail}'(e', \mathsf{getEntry}(e,i,t).\mathit{depending})}
$$

$$
\begin{array}{l}
\mathsf{propagateFail}'(e,\{\}) = e \\
\mathsf{propagateFail}'(e,\{(j,u)\} \cup \sigma) = \mathsf{propagateFail}'(\mathsf{propagateFail}(j,u,e),\sigma)
\end{array}
$$

$$
\mathsf{cmpExp}((i,t),(j,u),e) = \begin{cases} (e,\mathit{false}) & \textbf{if } \mathit{sourceAddr}(i,t) \neq \mathit{sourceAddr}(j,u) \\ \mathsf{cmpExp}((i,t),(j,t),\mathsf{classify}(i,t,e)) & \textbf{elseif } \mathsf{isUnbound}(e,(i,t)) \\ \mathsf{cmpExp}((i,t),(j,t),\mathsf{classify}(j,t,e)) & \textbf{elseif } \mathsf{isUnbound}(e,(j,t)) \\ (\mathsf{linkHypos}(e,i,j,t),\mathit{true}) & \textbf{elseif } \mathsf{getRepr}(e,i,t) = \mathsf{getRepr}(e,j,t) \\ (e,\mathit{false}) & \textbf{otherwise} \end{cases}
$$

Table 5. Reification II, Testing for Equivalence of Rewriting Results

First $\mathsf{visit}(\langle\rangle,\langle\rangle,\bot,\langle\rangle,(\langle\rangle,\mathsf{top}.2),e = (\{\},\{\},\bot))$ is called, as defined above, with $\mathsf{action}_\square(i,t,e)$ set to $\mathsf{action}_B(i,t,e)$. This simply fills the set $e.\mathit{pending}$, which is then emptied by calling $\mathsf{classifyAll}(e)$.

The operational context of the algorithm is defined by E_J in Table 4. Its fields are *.classes* (= for every source level definition: a map from all dynamic paths under which it is reachable, to an equivalence class modelled by the data type J), *.pending* (a set of all reference values which are still waiting for) and *.hypo* (= the most recent opened hypothesis).

The entries of type J contain *.state* (a current state), *.other* (a reference), *.different* (a set of dynamic paths for which are proven to be *not* equivalent) and *.depending* (a set of hypotheses which rely on this entry as a correct hypothesis).

In the current implementation, both phases are executed by interleaving.

The states have the following meaning: `hypo` means that the instantiation is hypothetically set equivalent with the instantiation given by the field *.other*; `equ` indicates that this equivalence has been proven; `failed` means the hypothesis has failed; `own` means the instantiation makes its own equivalence class and is itself the representative.

At the end of the process, each such entry is either `own` or `equ`. For each equivalence class exists one *representative*, as given by the function getRepr(..). Only this representative will appear in the generated documentation, statistics, etc.

The function classifyAll(..) steps through *e.pending* until all reachable references are classified by classify(..). First it tests whether a hypothesis is already active, then nothing needs to be done. Otherwise the test function ta(..) steps through all already established equivalence classes and tries cmpDef(..). If this succeeds, the identity is stored; if all candidates are exhausted, a new equivalence class is set up.

The function cmpDef(i, t, e, j) checks whether the definition with the identifier t in the instantiations of the same source text module, reachable by the dynamic paths i and j, can be made equivalent under the hypotheses in e, possibly by adding further hypotheses: First the hypotheses are stored in e by the function setState(..). This is defined in Table 5 and always updates the *.state* and *.other* field. (When state changes to `failed`, the non-identity with *.other* in memorized in *.different*, to prevent future futile reattempts, and the reference's need for classification is re-established; when it goes to `hypo`, this instance is stored as the "current hypothesis".)

With this new context the (expanded) expression values of the two definitions are compared calling cmpExp(..). If this fails, the fact of failure is propagated to all other hypotheses which rely on the current one.

The classification of expressions of the underlying domain language is discussed in the next section. But when two reference values (from S) are reached, the last rule in Table 5 applies: To be in the same equivalence class, two reference values must (of course) point to the same source text definition. If either of them is still unclassified, classify(..) is called on it; finally the classes (identified by the representatives from getRepr(..)) are simply compared.

The possible calls to classify(..) will re-enter the whole process recursively, and new hypotheses will be constructed trying to fullfil the current one. This can lead to arbitrarily complex graphs. When the described comparison of two (expanded) references succeeds, all currently just hypothetical classifications are linked to the current hypothesis *e.hypo* (= the last entered cmpDef(..)) for later failure propagation: due to possible cycles in the call graph, *e.hypo* may be a sub-hypothesis of one of its sub-hypotheses, which is not yet fully verified, but will fail when later visiting a "nephew" of *e.hypo*.[13]

[13] The current implementation of linkHypos(..) additionally memorizes whether any hypothesis has been used for the current hypothesis in *e.hypo* (in a further field of the corresponding J structure), and cmpDef(..) changes the *.state* to `equ` otherwise. This is a mere optimization and left out in this discussion, for brevity.

$$\frac{c_\_ \in \{\,\texttt{"}\,\alpha\,\texttt{"}\,,\ \texttt{'}\,\alpha\,\texttt{'}\,,\ 0\texttt{x}\,hhhh, \#\texttt{none}\}}{\mathsf{cmpExp}(c_1, c_2, \_, \_, \_, e) = (e, c_1 \stackrel{?}{=} c_2)}$$

$$\frac{\bigtriangleup_\_ \in \{\,\star\,,\ +\,,\ ?\,,\ \tilde{}\,\star\,,\ \tilde{}\,+\,\}}{\mathsf{cmpExp}(r_1 \bigtriangleup_1, r_2 \bigtriangleup_2, e) = \begin{cases} \mathsf{cmpExp}(r_1, r_2, e) & \textbf{if } \bigtriangleup_1 = \bigtriangleup_2 \\ (e, \mathit{false}) & \textbf{otherwise} \end{cases}}$$

$$\mathsf{cmpExp}(>\ r_1, >\ r_2, e) = \mathsf{cmpExp}(\texttt{@}\ r_1, \texttt{@}\ r_2, e) = \mathsf{cmpExp}(r_1, r_2, e)$$

$$\mathsf{cmpExp}(\ [\ t_1\, r_1\ ]\ ,\ [\ t_2\, r_2\ ]\ , e) = \begin{cases} \mathsf{cmpExp}(r_1, r_2, e) & \textbf{if } t_1 = t_2 \\ (e, \mathit{false}) & \textbf{otherwise} \end{cases}$$

$$\frac{\begin{array}{c}\square_\_ \in \{\,\texttt{,}\ ,\ \texttt{|}\ ,\ \&\ ,\ \tilde{}\ , \mathbb{U}\ , \mathbb{A}\ , -\ ,\ \texttt{..}\,\} \\ \mathsf{cmpExp}(r_1, r_2, e) = (e', X)\end{array}}{\mathsf{cmpExp}(r_1 \square_1 s_1, r_2 \square_2 s_2, e) = \begin{cases} (e, \mathit{false}) & \textbf{if } \square_1 \neq \square_2 \\ (e, \mathit{false}) & \textbf{elseif } X = \mathit{false} \\ \mathsf{cmpExp}(s_1, s_2, e') & \textbf{otherwise} \end{cases}}$$

Table 6. Reification III, Comparison of Expressions, Specific for the Underlying Domain Language.

3.2 Comparing Expressions

The functions in Table 6 check whether two (expanded) expressions are equivalent under the hypotheses in e.

If the constructors of both expressions differ, i.e. no rule in Table 6 matches, the unification fails by an implicit default rule, which indicates a *structural difference* of constructor application. The rule which compares two (expanded) reference values leads back recursively to the comparison of definitions and is thus defined already in Table 5 and discussed above.

All other rules are *specific* for the underlying domain language: On the level of expression alternatives may arise for equivalences, when the intended semantics re taken into account, e.g. coming from associative and commutative constructors. It may be appropriate to treat structural differences as context-sensitive and healable: in T the expressions "`(a,b)^(#none/b)`" and "`(a|b)^(#none/b)`" could be considered equivalent. These must be modeled as additional inference rules in Table 6 accordingly. The `d2d` implementation eliminates the need for these further choice points by transforming expressions into a normal form, based on a "complete order" of constructors and primitive scalars.

3.3 Using the Result

Once *e.pending* is emptied, all hypotheses which have survived as such are considered proven, since no counterexample has been found. This means to find

the *greatest* fixpoint, as mentioned above. All (expanded) references can be redirected to the one representative of their equivalence class. The resulting data structure can be used for algebraic purposes (here: parsing control) as well as for reified use, like statistics, documentation or user help functions: Only one and the same representative per equivalence group will always be visited.

3.4 Termination

The above-mentioned question (A) of termination is much harder to deal with, and again related to the intended semantics of the underlying domain language. It is only related to cyclic imports of modules, since cyclic references in the expression language can be safely ignored, as described above. A *sufficient* condition for termination is that every reference which appears directly or indirectly in a replacement expression for itself, due to a circular import, does so without any constructor application from T, i.e. is totally *identical* with this expression, modulo all renamings occurring on the circle. This condition prevents the import parameter from accumulating complexity with every turn of the unrolling of the import cycle, i.e. growing beyond all limits. Then our algorithm obviously terminates: There are statically only finitely many rewritten parameters and replacement expressions; all recursion goes through the functions `ta`() and `tb`() and both are consumptive by always removing one element from some finite set valued parameter.

4 Outlook

4.1 Future Work

In the context of our tool, following topics seem worth further research and development:

The condition for the termination of reification may be too strict for practical needs. Instead of forbidding all constructor applications on closed cycles, it may be more appropriate to allow that *idempotent constructors* may be applied. So this is again a point where properties of the underlying domain language come in play. In case of `d2d`, a module "`myA`" which contains a definition of and references to "`a`" could well import itself by "`import A from myA ^(a?/a)`". Taking the conventional semantics of T into account, it would have a well-defined and easy to find fixpoint, simply because of the idempotency of the constructor: "`a?=a??=a???=...`"

For one particular and fixed reference b, the application of "$\_ \mid b$", seen as one atomic transformation function, seems idempotent too: One and the same alternative can only be added once. But this case is more complicated since a further rewriting of b itself can interfere and cancel the assumption of a "fixed reference b".

To find an appropriate abstraction for these kinds of transformations is ongoing research; meanwhile the more restrictive termination criterion from above is used.

Furthermore, the expression language T has a very simple type system.[14] Type checking on the expanded definitions is trivial, but type checking of import statements in advance is a challenge.

In some contexts it may be useful to rewrite whole expressions, not only simple references. This means introducing some kind of pattern matching on the "b" side in the rewriting rules "a/b". We assume that beyond well-known standard problems like precedence, overlap, sequential order, this will not add any complexity to the subject of this article.

4.2 Related Work

The problem of rewriting in general is an important discipline on its own, with a wide range of theories, tools and applications. Nearly all of them differ from our approach since they deal with automated transformations, or preservations of properties, or semantical identity, etc. Contrarily, in our context rewriting is a mere practical means to derive new structures which are intended not to be equivalent.

There are only few cases when rewriting has been applied to grammars: Already in 2001 Lämmel described a collection of grammar transforming operators and their properties [3]. The motivating context is, similar to ours, a practical one, namely to reconstruct the originally intended semantics of ill-formed artefacts. Cyclic definitions and reification are not an issue. His operators and combinators live one and two levels above our simple substitution, respectively, and their influence on properties is exhaustively discussed. It seems promising to integrate some of these results into our future work; same holds for the fundamental considerations in [4].

RelaxNG [1] has an module import mechanism based on a notion of "file", which is defined by URLs. Thus mechanism supports replacement of definitions. The file contents is inserted on the front-end level, and the general incremental definition operators can be applied, not regarding the provenience. Our approach differs substantially: Since we support replacement of references, (1) the original definitions stay accessible, and (2) the rewriting takes place for a whole group of definitions in a consistent way.

The W3C XSD schema language has methods for *type derivation*. [5] Derived types can be used to define "substitution groups" which allow to replace one element class by another. But since type derivation can both extend and restrict types, the construction of precise semantics seems infeasible.

References

1. Clark and Murata. *Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG.* ISO/IEC, http://standards.iso.org/ittf/PubliclyAvailableStandards/c052348_ISO_IEC_19757-2_2008(E).zip, 2008.

[14] A hierarchy of singleton character sets, character sets, character sequences, character parsers and tag parsers; each constructor with one or more signatures.

2. Markus Lepper and Baltasar Trancón y Widemann. d2d — a robust front-end for prototyping, authoring and maintaining XML encoded documents by domain experts. In Joaquim Filipe and J.G.Dietz, editors, *Proceedings of the International Conference on Knowledge Engineering and Ontology Delelopgment, KEOD 2011*, pages 449–456, Lisboa, 2011. SciTePress.
3. Ralf Lämmel. Grammar adaptation. In *PROC. FORMAL METHODS EUROPE (FME) 2001, VOLUME 2021 OF LNCS*, pages 550–570. Springer-Verlag, 2001.
4. Ralf Lämmel and Wolfgang Lohmann. Format evolution, 2001.
5. Schema Working Group. *XML Schema*. W3C Candidate Recommendation, `http://www.w3.org/XML/Schema`, 2004.
6. J.M. Spivey. *The Z Notation: a reference manual*. Prentice Hall, 1988.

A Mathematical Notation

The employed mathematical notation is fairly standard, inspired by the Z notation [6]. But for leaner notation, we add some overloading. The following table lists some details:

$\mathbb{F}\ A$	Finite power set, the type of all *finite* subsets of the set A.
$A \times B$	The product type of two sets A and B, i.e. all pairs $\{c = (a, b) \mid a \in A \wedge b \in B\}$. We write $c.1$ and $c.2$ for the components a and b.
$A \to B$	The type of the *total* functions from A to B.
$A \nrightarrow B$	The type of the *partial* functions from A to B.
$\mathsf{dom}\, a, \mathsf{ran}\, a$	Domain and range of a function or relation.
$r \oplus s$	Overriding of function or relation r by s. Pairs from r are shadowed by pairs from s: $r \oplus s = \big(r \setminus (\mathsf{dom}\, s \times \mathsf{ran}\, r)\big) \cup s$
$r \oplus (a \mapsto b)$	Overriding of function or relation r by a single maplet.
A^*	The type of all final lists of elements from A.
$\alpha \frown \beta$	Concatenation of two lists.
$\langle\rangle$	An empty list.
$\alpha \blacktriangleleft b$	A list(/stack) with element b preceded by a prefix α.
$a \stackrel{?}{=} b$	The Boolean result of comparing for equality.
$T \stackrel{\triangle}{=} [a : A; b : B]$	Definition of a schema, i.e. a product type with named projections. Equivalent to $T = A \times B$, with $\pi_1 = .a$ and $\pi_2 = .b$

Functions are considered as special relations, i.e. as sets of pairs, thus can be used like "$f \cup g$".

Beweise über Programme mit call-time-choice-Semantik in Coq

Sandra Dylus
CAU Kiel
sad@informatik.uni-kiel.de

Jan Christiansen
Fachhochschule Flensburg
jan.christiansen@fh-flensburg.de

Finn Teegen
CAU Kiel
fte@informatik.uni-kiel.de

Um Beweise über Curry-Programme in einem Beweissystem wie Coq zu führen, müssen wir diese Curry-Programme in Coq-Programme transformieren. Es ist nicht möglich das Programm eins zu eins zu transformieren. Coq-Programme müssen total sein, während Curry-Programme nichtdeterministisch, und damit auch partiell, sein können. Diese Einschränkung von Coq hat zur Folge, dass der auftretende Nichtdeterminismus explizit modelliert werden muss. Ein natürliches Modell für Nichtdeterminismus ist die Repräsentation als Baumstruktur, wobei die nichtdeterministische Auswahl durch eine Verzweigung, eine purer Wert durch ein Blatt und eine Fehlschlag durch den leeren Baum modelliert werden. Es ist weiterhin erstrebenswert die Programme in Coq generisch zu modellieren, anstatt den Nichtdeterminismus als vorherrschenden Effekt fest einzubauen. Mit einem generischen Ansatz kann der Effekt bei der Verwendung explizit gewählt werden. Da die Effekte die wir verwenden wollen, meist auf Monaden zurückzuführen sind, ist es naheliegend die sogenannte *Freie Monade* zur Modellierung zu verwenden.

Es ist interessant sich Curry-Programme in diesem Ansatz anzuschauen, da der verwendete Effekt mehrere Eigenschaften erfüllen muss. Der Nichtdeterminismus in Curry folgt nämlich der *call-time-choice*-Semantik. Die Idee von dieser Semantik ist, dass nichtdeterministische Berechnung bei mehrfacher Verwendung, immer das gleiche Ergebnis liefern. Ein Ansatz m diese Semantik umzusetzen, ist allen Verzweigungen im Baum eindeutige Kennung zu geben. Bei der Auswertung nichtdeterministischer Berechnungen werden die Entscheidung für jede dieser Verzweigung in einem Zustand festgehalten, um bei wiederholten Auftreten der gleichen Kennung, konsistente Entscheidungen zu treffen. Um allen Verzweigungen eindeutige Kennungen zu vergeben, muss der Effekt zur Modellierung von Curry-Programmen neben dem Nichtdeterminismus auch ein Zustand für die vergebenen Kennungen abbilden.

Combining Static and Dynamic Contract Checking for Curry

Michael Hanus and Marius Rasch

Christian-Albrechts-Universität zu Kiel

Abstract. Static type systems are usually not sufficient to express all requirements on function calls. Hence, contracts with pre- and postconditions can be used to express more complex constraints on operations. Contracts can be checked at run time to ensure that operations are only invoked with reasonable arguments and return intended results. Although such dynamic contract checking provides more reliable program execution, it requires execution time and could lead to program crashes that might be detected with more advanced methods at compile time. To improve this situation for declarative languages, we present an approach to combine static and dynamic contract checking for the functional logic language Curry. Based on a formal model of contract checking for functional logic programming, we propose an automatic method to verify contracts at compile time. If a contract is successfully verified, dynamic checking of it can be omitted. This method decreases execution time without degrading reliable program execution. In the best case, when all contracts are statically verified, it provides trust in the software since crashes due to contract violations cannot occur during program execution.

Resource-aware Data Parallel Array Processing

Clemens Grelck[1] and Cédric Blom[2]

[1] University of Amsterdam
Amsterdam, Netherlands
C.Grelck@uva.nl
[2] Delft University of Technology
Delft, Netherlands
CedricBlom@outlook.com

Abstract. Malleable applications may run with varying numbers of threads, and thus on varying numbers of cores, while the exact number of threads/cores is irrelevant for the program logic. Malleability is a common property in data-parallel array processing, and with continuously growing core counts we are increasingly faced with the problem of how to choose the best number of threads/cores.
We propose a compiler-directed, almost automatic tuning approach for the functional array processing language SaC. Our approach consists of an offline training phase during which compiler-instrumented application code systematically explores the design space and accumulates a persistent database of profiling data. When generating production code our compiler consults this database and augments each data-parallel operation with a recommendation table. Based on these recommendation tables the runtime system chooses the number of threads individually for each data-parallel operation.
With energy/power efficiency becoming an ever greater concern, we explicitly distinguish between two application scenarios: aiming at best possible performance or aiming at a beneficial trade-off between performance and resource investment.

1 Introduction

SaC (aka Single Assignment C) is a purely functional, data-parallel array language [17, 18, 14] with a C-like syntax (hence the name). SaC features homogeneous, multi-dimensional, immutable arrays and supports both shape- and rank-generic programming: SaC functions may not only abstract from the concrete shapes of argument and result arrays, but even from their ranks (i.e. the number of dimensions). A key motivation for functional array programming is fully compiler-directed parallelization for various architectures starting from exactly the same one source. Currently, the SaC compiler supports general-purpose multi-processor and multi-core systems [13], CUDA-enabled GPGPUs [19], heterogeneous combinations thereof [8], the Amsterdam MicroGrid general-purpose many-core processor [15] or, most recently, clusters of workstations [25].

For the purpose of this paper we focus on general-purpose multi-core systems with cache-coherent shared memory. One of the advantages of a fully compiler-directed approach to parallel execution is that compiler and runtime system are technically free to choose any number of threads for execution, and by design the choice cannot interfere with the program logic. We call this characteristic property *malleability*. Malleability raises the central question of this paper: what would be the best number of threads to choose for the execution of a data-parallel operation? This choice depends on a number of factors, including but not limited to

- the number of array elements to compute,
- the computational complexity per array element and
- architecture characteristics of the compute system used.

For a large array of computationally challenging values making use of all available cores is a rather trivial choice on almost any machine. However, smaller arrays or less computational complexity or both inevitably leads to the observation illustrated in Fig. 1. While for a small number of threads/cores we often achieve almost linear speedup, the additional benefit of using more of them increasingly diminishes until some (near-)plateau is reached. Beyond this plateau using even more cores often shows a detrimental effect on performance.

This common behaviour [26, 28, 29, 27] can be attributed to essentially two independent effects. First, on any given system off-chip memory bandwidth is limited, and some number of actively working cores is bound to saturate the available bandwidth. Second, the organisational overhead for synchronisation, communication workload scheduling, etc, typically grows super-linearly in practice.

We can distinguish two scenarios for choosing the number of threads. From a pure performance perspective we would aim at the number of threads that yield the highest speedup. In the example of Fig. 1 that would be 16 threads. However, we typically observe a performance plateau around that optimal number. In the given example we can observe that from 12 to 20 threads the speedup obtained only marginally changes. Hence, finding a sub-optimal, but sufficiently near-optimal, number of threads suffices in practice.

However, as soon as computing resources are not considered free of charge, it does indeed make a big difference if we use 12 cores or 20 cores to obtain extremely similar performance. The 8 additional cores clearly fail to deliver any additional performance in the example of Fig. 1. Thus, they could more productively be used for other tasks. In the absence of any useful work, they could be powered down to save energy. This observation leaves us with two possible usage scenarios:

- aiming at best possible performance, the traditional HPC view, or
- aiming at a favourable trade-off between resource consumption and performance delivered.

Outside extreme high performance computing (HPC) the latter policy becomes more and more relevant. Here, we are looking at the gradient of the speedup

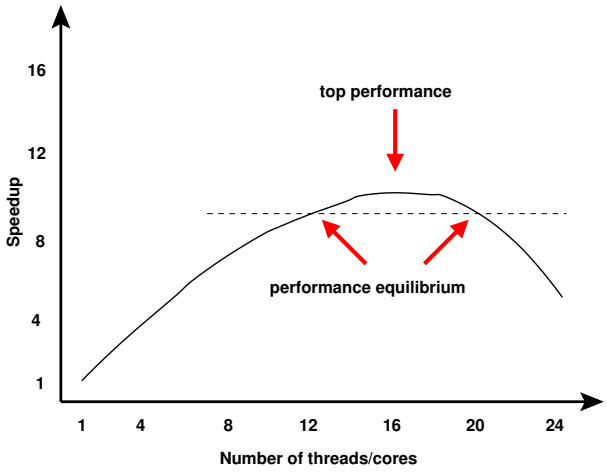


Fig. 1. Typical speedup graph observed for multi-core execution

curve. If the additional performance benefit of using one more core/thread drops below a certain threshold, we constitute that we have reached the optimal (with respect to the chosen policy) number of threads. Where exactly this threshold lies, is highly application- and situation-dependent.

In classical, high performance oriented parallel computing our issues have hardly been addressed because in this area users have typically strived for solving the largest possible problem size that still fits the constraints of the computing system used. In today's ubiquitous parallel computing [3], however, the situation has completely changed, and problem sizes are much more often determined by problem characteristics than machine constraints. But even in high performance computing some problem classes inevitably run into the described issues: multi-scale methods. Here, the same function(s) is/are applied to arrays of systematically varied shape and size. We illustrate multi-scale methods in Fig. 2 based on the example of the NAS benchmark MG (*multigrid*) [2, 11]. In this so-called *vcycle* algorithm (A glimpse at Fig. 2 should suffice to understand the motivation behind this name.) we start the computational process with a 3-dimensional array of large size and then systematically reduce the size by half in each dimension. This process continues until some predefined minimum size is reached, and then the process is sort of inverted and array sizes now double in each dimension until the original size is reached again. The whole process is repeated many times until some form of convergence is reached.

Since the array's size determines the break-even point of parallel execution in a data parallel array comprehension, it is clear that the optimal number of threads is different on the various levels of the vcycle. Regardless of the overall

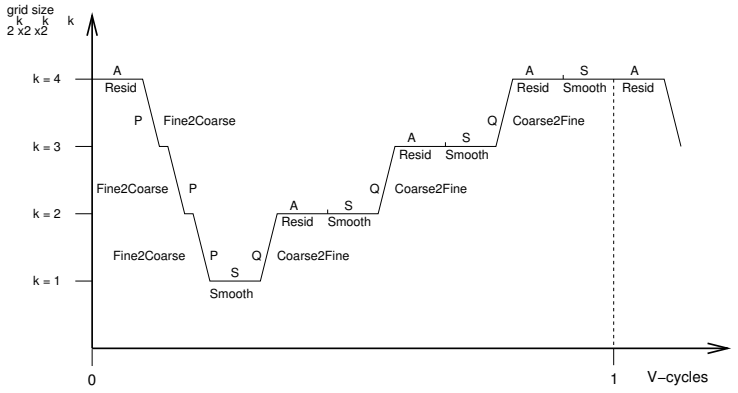


Fig. 2. Algorithmic vcycle structure of NAS benchmark MG as a representative of multi-scale methods, reproduced from [11]

problem size started with, we will always reach problem sizes where using varying fractions of the total number of threads will yield the best possible performance before for very small data sets purely sequential execution is the best solution.

All the above examples and discussions lead to one insight: in most non-trivial applications we cannot expect to find the one number of threads that is best across all data-parallel operations. This is the motivation for our proposed *smart decision tool* that aims at selecting the right number of threads for execution on the basis of individual data-parallel operations and user-configurable general policy in line with the two usage scenarios sketched out above. The smart decision tool is meant to replace a much more coarse-grained solution that SaC shares with many other high-level parallel languages, namely that based on heuristic methods some data-parallel operations in an application program may be run entirely sequential.

Our smart decision tool is based on the assumption that for many data-parallel operations the effectively best choice in either usage scenario neither is to use all cores for parallel execution nor to only use a single core for completely sequential execution. We follow a two-phase approach that distinguishes between offline training runs and online production runs of the same application. In training mode compilation our compiler instruments the generated code to produce an individual performance profile for each individual data-parallel operation. In production mode compilation we associate each data-parallel operation with an oracle that based on the performance profiles gathered offline chooses the number of threads based on the array sizes encountered at application production runtime.

The distinction between training and production mode has the disadvantage that users need to explicitly and consciously use the smart decision tool for man-

ual optimisation. One could think of a more seamless and transparent integration where applications as silently as continuously produce profiling data stored in a database and dynamically consult this database to make educated decisions. We rejected such an approach because of its adverse effects on production runtime performance. In contrast, our proposed approach inflicts minimal runtime overhead in production mode.

The remainder of the paper is organised as follows. Section 2 provides some background information on our functional array language SaC while Section 3 sketches out compilation into multithreaded code and the SaC runtime system for multi-core machines. In Sections 4 and 5 we describe our proposed smart decision tool in detail: training mode and production mode, respectively. In Section 6 we outline necessary modifications of SaC's runtime system. Some preliminary experimental evaluation is discussed in Section 7. Finally, we sketch out related work in Section 8 before we draw conclusions in Section 9.

2 Introducing SAC

As the name "Single Assignment C" suggests, SaC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This is meant to facilitate adoption in compute-intensive application domains, where imperative concepts prevail. Core SaC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions; details can be found in [17].

Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SaC code while the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation.

On top of this language kernel SaC provides genuine support for processing truly multidimensional and truly stateless/functional arrays using a shape-generic style of programming. Any SaC expression evaluates to an array. Arrays may be passed between functions without restrictions. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]`, and arrays of any rank, e.g. `int[*]`. The latter include scalars, which we consider rank-0 arrays with an empty shape vector. For convenience and equivalence with C we use `int` rather than the equivalent `int[]` as a type notation for scalars. The hierarchy of array types induces a subtype relationship, and SaC supports function overloading with respect to subtyping.

SaC only features a very small set of built-in array operations, among others to query for rank and shape, or to select individual arry elements. Aggregate array operations are specified in SaC itself using WITH-loop array comprehensions:

```
with {
  ( lower_bound <= idxvec < upper_bound) : expr;
   ...
  ( lower_bound <= idxvec < upper_bound) : expr;
}: genarray( shape, default)
```

Here, the keyword `genarray` characterises the WITH-loop as an array comprehension that defines an array of shape *shape*. The default element value is *default*, but we may deviate from this default by defining one or more index partitions between the keywords `with` and `genarray`.

Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to loop variables in FOR-loops. Unlike FOR-loops, we deliberately do not define any order on these index sets. We call the specification of such an index set a *generator* and associate it with some potentially complex SAC expression that is in the scope of *idxvec* and thus may access the current index location. As an example, consider the WITH-loop

```
A = with {
      ([1,1]<=iv<[4,5]): 10*iv[0]+iv[1];
      ([4,0]<=iv<[5,5]): 42;
    }: genarray( [5,5], 99);
```

that defines the 5×5 matrix

$$A = \begin{pmatrix} 99 & 99 & 99 & 99 & 99 \\ 99 & 11 & 12 & 13 & 14 \\ 99 & 21 & 22 & 23 & 24 \\ 99 & 31 & 32 & 33 & 34 \\ 42 & 42 & 42 & 42 & 42 \end{pmatrix}$$

WITH-loops in SAC are extremely versatile. In addition to the dense rectangular index partitions show above SAC supports also strided generators. In addition to the `genarray`-variant used here, SAC features further variants, among others for reduction operations. Furthermore, a single WITH-loop may define multiple arrays or combine multiple array comprehensions with further reduction operations, etc. For a complete, tutorial-style introduction to SAC as a programming language we refer the interested reader to [14].

3 Compiling SAC

Compiling SAC programs into efficiently executable code for a variety of parallel architectures is a non-trivial undertaking. While this clearly is not the place to explain the compilation process in any detail, we still sketch out a few areas most relevant for our current work.

It probably comes at no surprise to the experienced reader that WITH-loops, as introduced in the previous section, play a central role in the compilation of SAC programs. Any aggregate array operation in SAC, in one way or another, is expressed by means of WITH-loops, may it be explicitly in the application program or implicitly through composition of basic operations from the SAC standard array library. Thus, we can expect that almost any SAC program spends almost all execution time in WITH-loops.

Many of our optimisations are geared towards the composition of multiple WITH-loops into one [16]. These compiler transformations systematically improve the ratio between productive computing and organisational overhead. Consequently, when it comes to generating multithreaded code for parallel execution on multi-core system, we can focus on individual WITH-loops. WITH-loops are data-parallel by design. Thus, any WITH-loop can be executed in parallel. The subject of our current work is: should it?

So far, the SAC compiler has generated two alternative codes for every WITH-loop: a sequential and a multithreaded implementation. The choice which route to take is made at runtime based on two criteria:

- If program execution is already in parallel mode, we evaluate nested WITH-loops sequentially.
- If the size of an index set is below a configurable threshold, regardless of the computational complexity per element, we evaluate the WITH-loop sequentially.

Multithreaded program execution follows an offload model (or fork/join-model), as illustrated in Fig. 3. Program execution always starts in single-threaded mode and only if the execution reaches a WITH-loop for which both above criteria are met, worker threads are created that join the master thread in the execution of the data-parallel WITH-loop. A WITH-loop-scheduler assigns array elements for computing to among the worker threads according to one of several policies that range from static scheduling to dynamic self-scheduling with and without affinity control. In fact, the SAC compiler implements various scheduling technique to be selected by the programmer, but all this is irrelevant for our current work. When no more work is available, the worker threads terminate and, having waited for the last worker thread, the master thread resumes single-threaded execution.

The total number of threads, eight in the illustration of Fig. 3, is once determined at program startup and remains the same throughout program execution. This number is typically motivated by the hardware resources of the deployment system. Due to the malleability property of the data-parallel applications concerned, application characteristics are mostly irrelevant. While it would be technically simple to determine the number of available cores at application start, SAC for the time being expects this number to be provided by the user, either through a command line parameter or through an environment variable.

As illustrated on the right hand side of Fig. 3 does not literally implement the fork/join-model, but rather starts the worker threads right at the beginning and before the first WITH-loop is encountered during program execution. All worker threads are preserved until program termination. The conceptual fork/join model

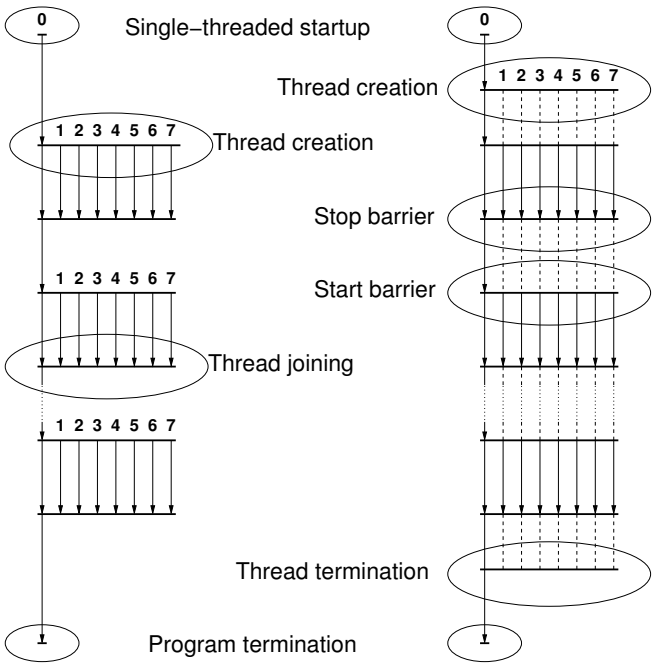


Fig. 3. Multithreaded execution models: conceptual fork-join model (left) and start/stop barrier implementation (right)

is implemented through two dedicated barriers: the *start barrier* and the *stop barrier*. At the start barrier worker threads wait for activation by the master thread. At the stop barrier the master thread waits for all worker threads to complete the parallel section while the worker threads immediately pass on to the following start barrier. We use highly efficient tailor-made implementations that exploit properties of the cache coherence protocol, but are essentially based on spinning. All details about our barrier implementations in particular and SAC's multicore implementation in general can be found in [12, 13].

4 Smart decision training mode

The proposed *smart decision tool* consists of two modes: we first describe the *training mode* in this section and then focus on the *production mode* in the following section. When compiling for smart decision training mode, the SAC compiler instruments the generated multithreaded code in such a way that

– for each WITH-loop we systematically explore the entire design space regarding the number of threads;
– we repeat each experiment sufficiently many times to ensure a meaningful timing granularity while avoiding excessive training times;
– profiling data is stored in a custom binary database.

At the same time we aim at keeping the smart decision training code as orthogonal to the existing implementation of multithreading as possible, mainly for general software engineering concerns. Fig. 4 shows pseudo code that illustrates the structure of the generated code. To make the pseudo code as concrete as possible, we pick up the example WITH-loop introduced in Section 2.

```
size = 5 * 5;

A = allocate_memory( size * sizeof(int));

spmd_frame.A = A;
num_threads = 1;
repetitions = 1;
init = 1;

do {
  start = get_real_time();

  for (int i=0; i<repetitions; i++) {
    StartThreads( num_threads,
                  spmd_fun, spmd_frame);
    spmd_fun( 0, num_threads, spmd_frame);
  }

  stop = get_real_time();

  repetitions, num_threads, init
    = TrainingOracle (init, unique_id, size,
                      repetitions, start, stop);
}
while (repetitions > 0);
```

Fig. 4. Compiled pseudo code of the example WITH-loop from Section 2 in smart decision training mode

The core addition to our standard code generation scheme is a `do-while`-loop plus a timing facility wrapped around the original code generated from our WITH-loop. Let us briefly explain the latter first. The pseudo function `StartThreads` is meant to lift the start barrier for `num_threads`-1 worker threads. They subsequently execute the generated function `spmd_fun` that contains most of the code generated from the WITH-loop, among others the resulting nesting of C `for`-

loops, the WITH-loop-scheduler and the stop barrier. The record `spmd_frame` serves as a parameter passing mechanism for `spmd_fun`. In our concrete example, it merely contains the memory address of the result array, but in general also all values referred to in the body of the WITH-loop would be made available to all worker threads via `spmd_frame`. After lifting the start barrier, the master thread temporarily turns itself into a worker thread by calling `spmd_fun` directly via a conventional function call. Note that the first argument given to `spmd_fun` denotes the thread ID. All worker threads require the number of active threads (`num_threads`) as input for the WITH-loop-scheduler.

Coming back to the specific code for smart decision training mode, we immediately identify the timing facility, which obviously is meant to profile the code, but why do we wrap the whole code within another loop? Firstly, the functional semantics of SaC and thus the guaranteed absence of side-effects in the WITH-loop allow us to actually execute the compiled code multiple times without affecting semantics. In a non-functional context this would immediately raise a plethora of concerns whether running some piece of code repeatedly may have an impact on application logic.

However, the reason for actually running a single WITH-loop multiple times is motivated by creating more reliable timing data. Take into account that we have very little a-priori insight into how long the with-loop is going to run. Shorter runtimes often result in greater relative variety of measurements. To counter such effects, we first run the WITH-loop once to obtain a rough estimate of its execution time. Following this initial execution a *training oracle* decides about the number of repetitions to follow in order to obtain meaningful timings while keeping overall execution time at acceptable levels.

In addition to controlling the number of repetitions our training oracle systematically varies the effective number of threads employed. More precisely, the training oracle implements a three step process:

Step 1: dynamically adjust the time spent on a single measurement iteration to match a certain pre-configured time range. During this step the WITH-loop is executed once by a single thread, and the execution time is measured. Based on this time the training oracle determines how often the WITH-loop could be executed without exceeding a configurable time limit, currently 500ms.

Step 2: measure the execution time of the WITH-loop while systematically varying the number of threads used. This step consists of many cycles, each running the WITH-loop as many times as determined in step 1. After each cycle the execution time of the previous cycle is stored, and the number of threads used during the next cycle is increased by one. Step 2 ends as soon as the number of threads reaches the preset maximum.

Step 3: collect measurement data to create a performance profile that is stored on disk. During this step all time measurements collected in step 2 are packaged together with three characteristic numbers of the profile: a unique identifier of the WITH-loop, the size of the index set and the number of repetitions in step 1. The packaged data is stored in the application-specific binary smart decision database file on disk.

Let us have a closer look into the last of the three steps. The SaC compiler determines the unique identifier at compile time, if in training mode, by simply counting all WITH-loops in the SaC module compiled. The resulting identifier is compiled into the generated code as one argument of the training oracle. Here, it is important to understand that we do not count the WITH-loops in the original source code written by the user, but those in intermediate code after substantial program transformations by the compiler.

The index set size may be known at compile time, as in our simple example, or may only be computed at runtime. In case of a `genarray` or `modarray` WITH-loop the size of the index set coincides with that of the array defined, and is already required for the purpose of memory allocation independent of our current work. However, in the case of a `fold`-WITH-loop we need to generate an expression that symbolically describes the index set size based on the generators' lower and upper bound specifications (and possibly their strides).

The organisation of the binary database file in rows of data is illustrated in Fig. 5. Each row starts with the three integer numbers that characterise the measurement: WITH-loop id, index set size and number of repetitions, each in 64-bit representation. What follows in the row are the time measurements with different numbers of threads. Thus, the length of the row is determined by the preset maximum number of threads. For instance, the first row in Fig. 5 contains a total of seven numbers: the three characteristic numbers followed by profile data for one, two, three and four threads, respectively. The second row in Fig. 5a accordingly stems from a training of the same application with the maximum number of threads set to two.

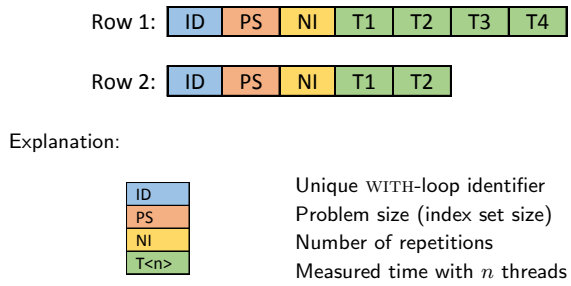


Fig. 5. Illustration of training database rows

The smart decision tool recognises a database file by its name. We use the following naming convention:

`stat.`*`name`*`.`*`architecture`*`.`*`#threads`*`.db`

and store database files in a specific subdirectory of the `.sac2c` subdirectory in the user's home directory. Both *`name`* and *`architecture`* are set by the user through corresponding compiler options when compiling for training mode. Oth-

erwise, we use suitable default values. The name field `#threads` is the preset maximum number of threads. The *`name`* option is mainly meant for experimenting with different compiler options and/or code variants and thus keep different smart decision databases at the same time. The *`architecture`* reflects the fact that the system on which we run our experiments and later the production code crucially affects our measurements. Profiling data obtained on different systems are usually incomparable. With this option we can distinguish data obtained on different execution systems, even if they end up on the same file system, e.g. a file server with user directories mounted on various different machines. In the long run, we would rather want to set this parameter automatically, but rather is an engineering than a research concern and so we postpone it for now.

Users may choose to repeat the training of a SAC program. If name, architecture and number of threads match those of an existing database, new measurements are merged into that file. For every new measurement the smart decision tool first checks if there are already similar measurements in the database file based on WITH-loop identifier and problem size. In case of a match, existing and new measurements are pairwise added, and the resulting values are written back into the corresponding database row.

5 Smart decision production mode

Continuous training leads to a collection of database files. In an online approach running applications would consult these database files in deciding about the number of threads to use for each and any instance of a WITH-loop encountered during program execution. However, locating the right data base in the file system, reading and interpreting its contents and then making a non-trivial decision would incur considerable runtime overhead that we would need to compensate first before realising any advantage through smarter decisions regarding the effective number of threads.

Therefore, we decided to take a different route and actually consult the database files created by training mode binaries when compiling production binaries. This way we can move almost all overhead to production mode compile time while keeping the actual production runtime overhead minimal. In production mode the SAC compiler does three things with respect to the smart decision tool:

1. it reads the relevant database files identified by *`name`* and *`architecture`*;
2. it applies a merge operation to combine information from several database files;
3. it creates a recommendation table for each WITH-loop.

These recommendation tables are compiled into the SAC code and used at runtime component to decide for each instance of a WITH-loop how many threads to actually.

The combination of *`name`* and *`architecture`* must match with at least one database file, but it is well possible that a specific combination matches with

several files, for example if the training is first done with a maximum of two threads and later repeated with a maximum of four threads. In such cases we read all matching database files for any maximum number of threads and merge them. The merge process is executed for each WITH-loop individually. As in training mode we identify each WITH-loop by a unique identifier. Since training and production mode compilation does not lead to different intermediate code representations otherwise, we are guaranteed to obtain the same unique identifier for each WITH-loop in either mode. These unique identifiers are matched with the identifiers in the database files to create subselections of database rows.

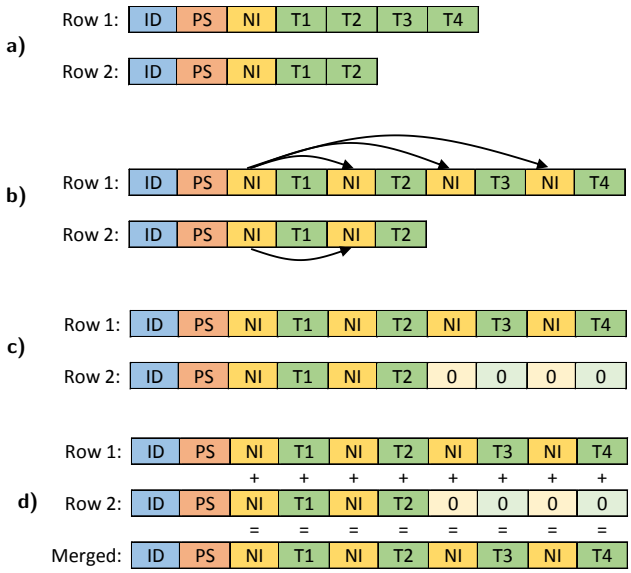


Fig. 6. Illustration of database row merging

Database rows are merged pairwise, as illustrated in Fig. 6. First, a mini-database is created in memory to store the merged rows. Second, the rows from the sub selection are read one by one and prepared for merging: The index set sizes of the row are copied in front of each time measurement (Fig. 6b). Rows are padded with empty entries where needed to make all rows as long as the one resulting from running the largest number of threads (Fig. 6c). Third, the position of each row in the mini-database is determined using rank sort. The problem size of each row is used as index for the rank sort algorithm. Rows with the same index become new rows in the mini-database. If two or more rows have the same index (e.g. they have the same problem size), they are merged by

simply adding the problem sizes and time measurements of the corresponding rows (Fig. 6d). Finally, all time measurements are divided by the corresponding problem sizes to compute the average execution time of the WITH-loop, which are likewise stored in the mini-database.

Following the merge process, the compiler creates a recommendation table, based on the in-memory mini-database. This recommendation table consists of two columns. The first column contains the different problem sizes encountered during training. The second column holds the corresponding recommended number of threads. Recommendations are computed based on the average execution times in relation to the problem sizes. Average execution times are turned into a performance graph by taking the inverse of each measurement. The performance graph is then normalized to the range zero to one. Then, we determine the gradient between any two adjacent numbers of threads in the performance graph and compared it with a configurable threshold gradient (default: 10 degrees). The recommended number of threads is the highest number of threads for which the gradient towards using one more thread is below the gradient threshold. In other words, the gradient threshold is the crucial knob by which users control whether to tune for performance or for performance/energy trade-offs. At last, the entire recommendation table is compiled into the production SAC code, just in front of the corresponding WITH-loop.

The runtime component of the smart decision production code is kept as lean and efficient as possible. When reaching some WITH-loop during execution, we compare the actual problem size encountered with the problem sizes in the recommendation table. If we find a direct match, the recommended number of threads is taken from the recommendation table. If the problem size is in between two problem sizes in the recommendation table, we use linear interpolation to estimate the optimal number of threads. If the actual problem size is smaller than any one in the recommendation table, the recommended number of threads for the smallest available problem size used. In case the actual problem size exceeds the largest problem size in the recommendation table, the recommended number of threads for the largest problem size in the table is used. So, we do interpolation, but refrain from extrapolation beyond both the smallest and the largest problem size in the recommendation table.

6 Smart decision runtime

In this section we describe the extensions necessary to actually implement the decisions made by the smart decision tool at runtime. As outlined in Section 3, a SAC program compiled for multithreaded execution alternates between sequential single-threaded and data-parallel multithreaded execution modes. Switching from one mode to the other is the main source of runtime overhead, namely for synchronisation of threads and communication of data among them. As illustrated in Fig. 7, start and stop barriers are responsible for the necessary synchronisation and communication, but likewise for the corresponding overhead. Hence, their efficient implementation is crucial.

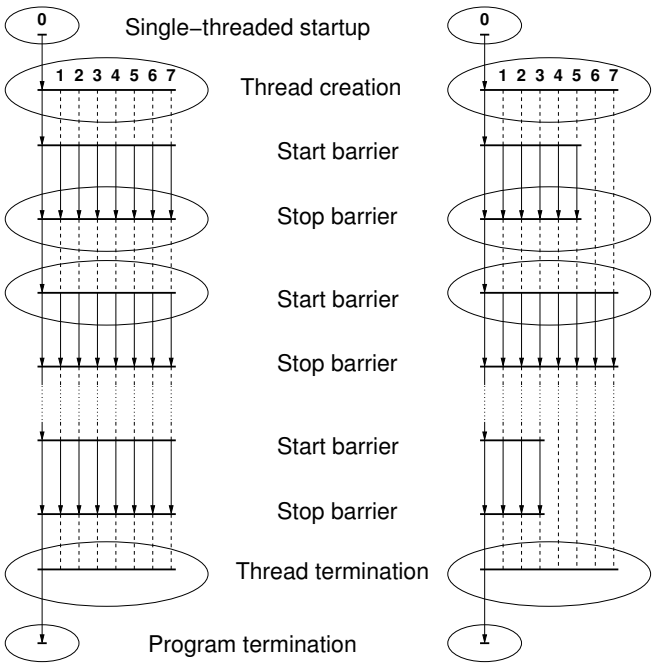


Fig. 7. Multithreaded execution: start/stop barrier model with fixed number of threads (left) and proposed model with fixed thread pool but tailor-made activations on a per WITH-loop basis (right)

SAC's standard implementations of start and stop barriers are based on active waiting, or *spinning*. This choice is motivated by the low latency of spinning barriers in conjunction with the expectation that SAC applications typically spend most execution time within the multithreaded execution mode. Thus, threads are expected to never wait long at a start barrier for their next activation while load balancing scheduling techniques make sure that waiting times at stop barriers are kept low.

For the high performance application scenario of our work, we ignore all energy-saving opportunities and stick to spinning barriers as outlined in Fig. 8. Our implementation makes use of two counters: the `local` counter is stack-allocated and private to each thread, whereas the volatile `global` counter is shared by all threads. As long as their values coincide, execution is captured in the `while`-loop. The master thread releases the worker threads by incrementing the global counter. For a more detailed discussion of this synchronisation mechanism we refer the interested reader to [13].

```
void BarrierWait( volatile unsigned int *global,
                            unsigned int *local)
{
  while (*global == *local) {
    // spin
  }
  (*local)++;
}

void BarrierRelease( volatile unsigned int *global)
{
  (*global)++;
}
```

Fig. 8. Start barrier implementation with thread spinning

For the purpose of our current work we introduce an additional high watermark for threads, as illustrated in Fig. 7. This high watermark achieves that only the recommended number of threads is actually activated in the start barrier and waited for in the stop barrier. We always use those threads with thread ID below the high watermark. With minimal change of existing implementation this can be achieved by using the changing high watermark instead of the fixed total number of threads in all scheduling decisions, regardless of the actual scheduling policy chosen. This way WITH-loop-schedulers do not assign any work to threads with ID beyond the high watermark, and these threads then immediately hit the stop barrier and subsequently the start barrier.

Spinning barriers are of course of little use for the performance/energy trade-off scenario of our work. Therefore, we introduce three further non-spinning barrier types to be selected by the user via a corresponding command line option of the SAC compiler. These barriers suspend threads at the start barrier and re-activate them as needed. The second barrier implementation is the built-in PThread barrier, the third is based on condition variables and shown in Fig. 9 and the fourth barrier implementation is a variant solely based on mutex locks.

The thread suspending start barrier implementation shown in Fig. 9 takes in principle the same approach as the spinning barrier of in Fig. 8, but additionally uses a mutex lock and a condition variable. Instead of spinning on the equality condition of the two counters, we now call `pthread_cond_wait` to suspend. Analogously, we call `pthread_cond_broadcast` to wake up the suspended worker threads for action.

7 Experimental evaluation

We evaluate our approach with a series of experiments using two different machines of the DAS-4 research cluster. The smaller of our test systems is equipped with two Intel Xeon quad-core E5620 processors with hyperthreading enabled. These eight hyperthreaded cores run at 2.4 GHz and the entire system has 24GB

```
void BarrierWait( volatile unsigned int *global,
                            unsigned int *local)
{
  pthread_mutex_lock( &barrier_mutex);
  if (*global == *local) {
    pthread_cond_wait( &barrier_condvar,
                       &barrier_mutex);
  }
  pthread_mutex_unlock( &barrier_mutex);
  (*local)++;
}

void BarrierRelease( volatile unsigned int *global)
{
  pthread_mutex_lock( &barrier_mutex);
  (*global)++;
  pthread_cond_broadcast( &barrier_condvar);
  pthread_mutex_unlock( &barrier_mutex);
}
```

Fig. 9. Start barrier implementation with thread suspension

of memory. The larger of our test system features four AMD 6100 12-core processors and has 128GB of memory. Both systems are operated in batch mode giving us exclusive access for the duration of our experiments. In the sequel we will refer to these systems as the Intel or as the AMD system, respectively.

Before exploring the actual smart decision tool, we investigate the runtime behaviour of the four barrier implementations sketched out in the previous section. In Fig. 10 we show results obtained with a synthetic micro benchmark that puts maximum stress on the barrier implementations. We systematically vary the number of cores and show actual wall clock execution times of the micro benchmark.

Two insights can be gained from this initial experiment. Firstly, from our three non-spinning barrier implementations the one based on condition variables clearly performs best across all levels of concurrency. Therefore, we restrict all further experiments to this implementation as the representative of thread-suspending barriers and relate its performance to that of the spinning barrier implementation. Secondly, we observe a substantial performance difference between the spinning barrier on the one hand side and all three non-spinning barriers on the other hand side. Positively, this experiment nicely demonstrates how well tuned the SAC synchronisation primitives are. Nevertheless, the experiment also shows that the performance/energy trade-off scenario we sketched out earlier is not easy to address.

Before exploring the effect of the smart decision tool on any complex application programs, we need to better understand the basic properties of our approach. Therefore we use a very simple, almost synthetic benchmark through-

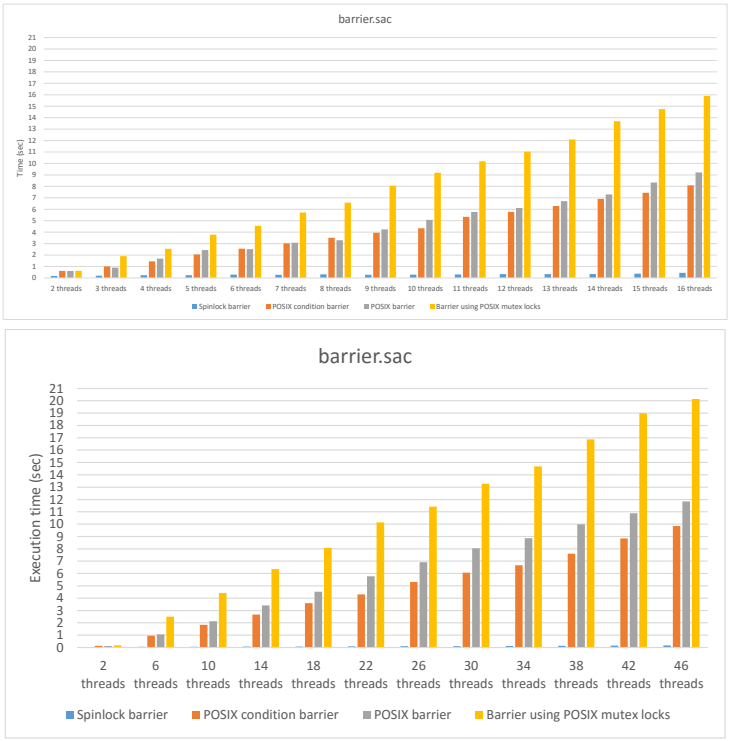


Fig. 10. Scalability of our four barrier implementations on the 8-core hyperthreaded Intel system and on the 48-core AMD system

out the remainder of this section: repeated element-wise addition of two matrices. We explore two different problem sizes, 50×50 and 400×400, that have proven to yield representative results for spinning and non-spinning barrier implementations, respectively. We first present experimental results obtained on the AMD system with spinning barriers in Fig. 11 and with suspending barriers in Fig. 12.

Frankly speaking, we cannot be happy with the results reported. For the larger problem size of 400x400 the human eye easily identifies that no fundamental speedup limit is reached up to the 48 cores available. Nonetheless, an intermediate plateau around 26 cores makes the smart decision tool (or not so smart decision tool) choose to limit parallel activities at this level.

For the smaller problem size of 50x50 we indeed observe the expected performance plateau, and the smart decision decides to limit parallelisation to 24 core. Subjectively, this appears to to be on the high side as 12 core already achieve a

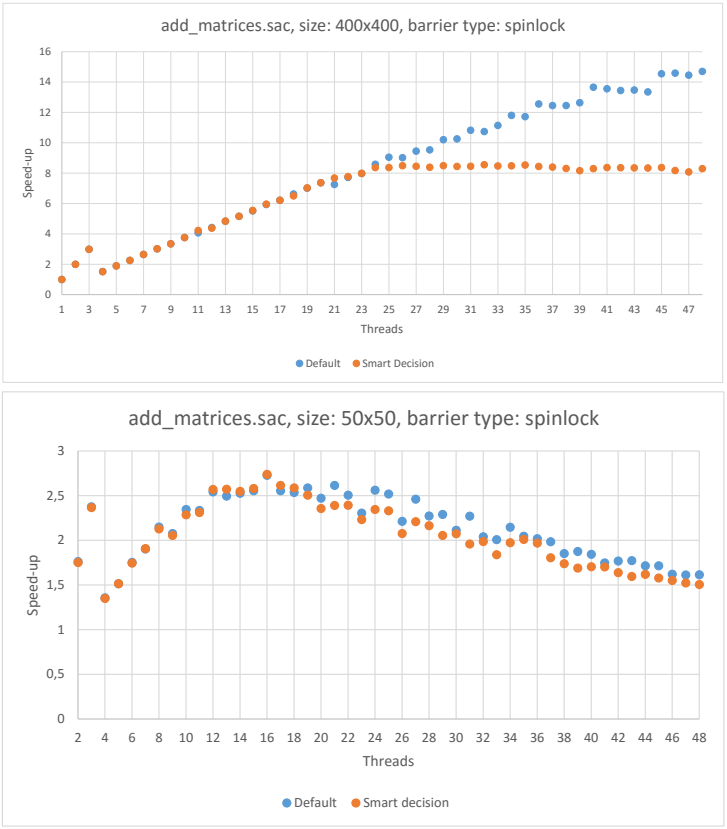


Fig. 11. Performance on AMD 48-core system with and without the proposed smart decision tool for two different problem sizes and spinning barrier implementation; smart decision tool recommendations: 26 and 24

speedup of 2.5, which is very close to the maximum. Trouble is we cannot realise the expected performance for higher thread numbers. Our expectation would be to keep a speedup of about 2.5 even if the maximum number of threads is chosen at program start to be higher.

We attribute this to the fact that our implementations of the start barrier always activate all threads, regardless of what the smart decision tool suggests. Its recommendation merely affects the WITH-loop-scheduler, which divides the available work evenly among a smaller number of active threads. We presume

that this implementation choice inflicts two much overhead in relation to the fairly small problem size, and synchronisation cost dominate our observations.

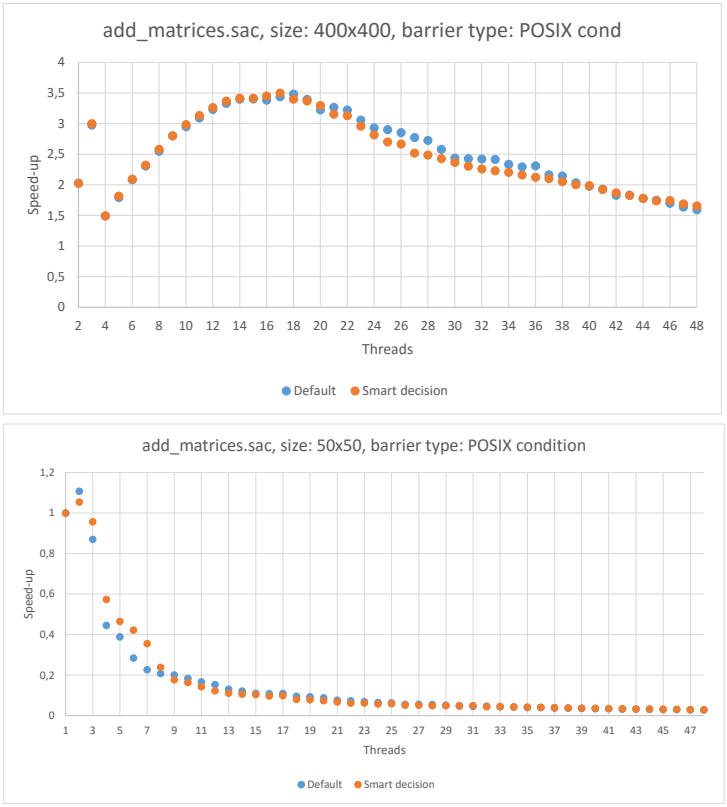


Fig. 12. Performance on AMD 48-core system with and without the proposed smart decision tool for two different problem sizes and suspending barrier implementation; smart decision tool recommendations: 24 and 1

When using suspending barriers instead of spinning barriers, as shown in Fig. 12, we see a similar picture. Only the considerably higher overhead of suspending barriers, as already observed in Fig. 10, makes the 400x400 graph for suspending barriers resemble the 50x50 graph for spinning barriers. Accordingly, running the 50x50 experiment with suspending barriers results in a major slowdown due to parallel execution. Still, we can observe that our original, motivating

assumption indeed holds: the best possible performance is neither achieved with one core nor with 48 cores, but in this particular case with two cores.

A general observation across all experiments so far is that the runtime behaviour with up to four threads is rather unexpected. This can best be seen in the graphs for the 400x400 matrices (top). With up to three threads we see expected speedups, but with four threads performance considerably goes down. From the low basis of four thread performance we can then observe an continuous incline again. This behaviour appears to be related to the system configuration of four processors with 12 cores each, but we do not have a plausible explanation of the concrete behaviour observed.

What is clear, however, is that the reproducible local performance maximum at low thread counts irritates our heuristics in constructing recommendation tables. To cope with such training data we decided not to take the angle as mentioned in Section 5 literal, but rather as a rough indication.

We now present experimental results obtained on the Intel system with spinning barriers in Fig. 13 and with suspending barriers in Fig. 14. Overall these figures confirm the results obtained on the 48-core AMD system. In the 400x400 experiments we can clearly identify the hyperthreaded nature of the architecture. For example in the 400x400 experiment with spinning barriers speedups continuously grow up to eight threads, dramatically diminish for nine threads and then again continuously grow up to 16 threads.

8 Related Work

It is a well known fact that not every parallelisable loop should indeed be run in parallel for optimal performance. Successful parallelisation not only depends on functional properties of the loop body, which may be known at compile time, but to a large extent on runtime parameters such as data set sizes or even data values themselves. Therefore, many parallel programming approaches provide basic means to switch off parallel execution in generally parallelised loops based on runtime values.

An example of this kind of mechanism is the `if`-clause of OpenMP [7], which allows programmers to postpone the decision whether or not to execute a parallel section of code actually in parallel by multiple worker threads or still sequentially by the single master thread until runtime. The `if`-clause contains a predicate that may be based on runtime variables of the programmer's choice. Analogous to our own motivation, the `if`-clause reflects the fact that parallel execution may not always be beneficial for runtime performance, but whether or not it actually is depends on information only available at application runtime.

Another similar example is the `--dataParMinGranularity` command line option of Chapel [4, 5], which likewise allows the user to exercise control over the mostly implicit data parallel constructs of the Chapel language. Unless instructed otherwise, Chapel automatically uses all available cores for implicit parallelisation. The above command line option is automatically added to every Chapel compiled code and allows the user of an application to a-posteriori

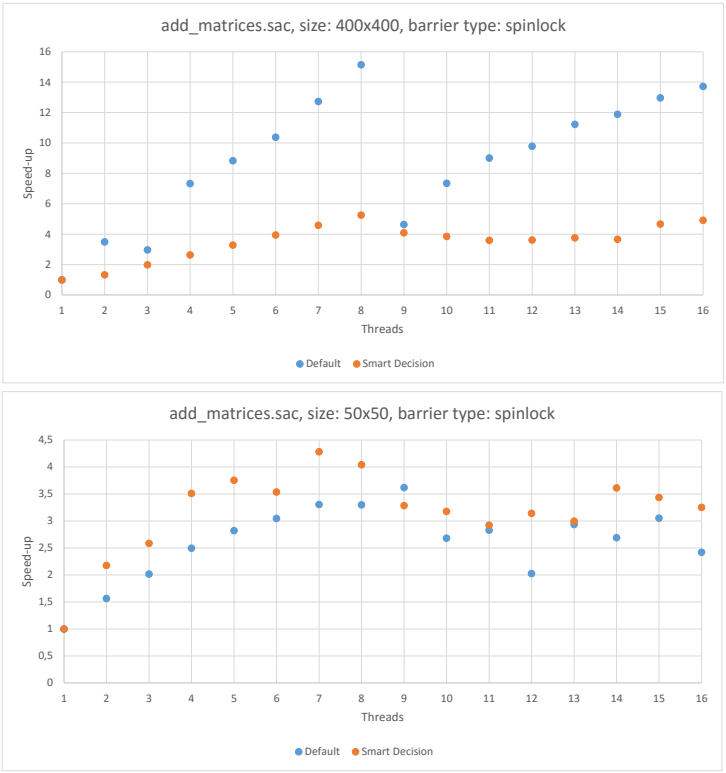


Fig. 13. Performance on Intel 8-core hyperthreaded system with and without the proposed smart decision tool for two different problem sizes and spinning barrier implementation; smart decision tool recommendations: 9 and 9

predicate the decision to parallelise or not to parallelise on the size of the index space (or *domain* in Chapel speak) of each data-parallel operation. In contrast to the `if`-clause of OpenMP only one value can be set across the entire application, whereas the optimal value obviously depends on the compute intensity per element of each individual data-parallel operation.

Prior to our current work, the SaC compiler has adopted a similar strategy as Chapel: at compile time the user may set a minimum index size for parallelisation, and the generated code at runtime decides between sequential and parallel execution based on the given size [12, 13]. The sac compiler makes use of a suitable default minimum index set size if the user does not provide specific information.

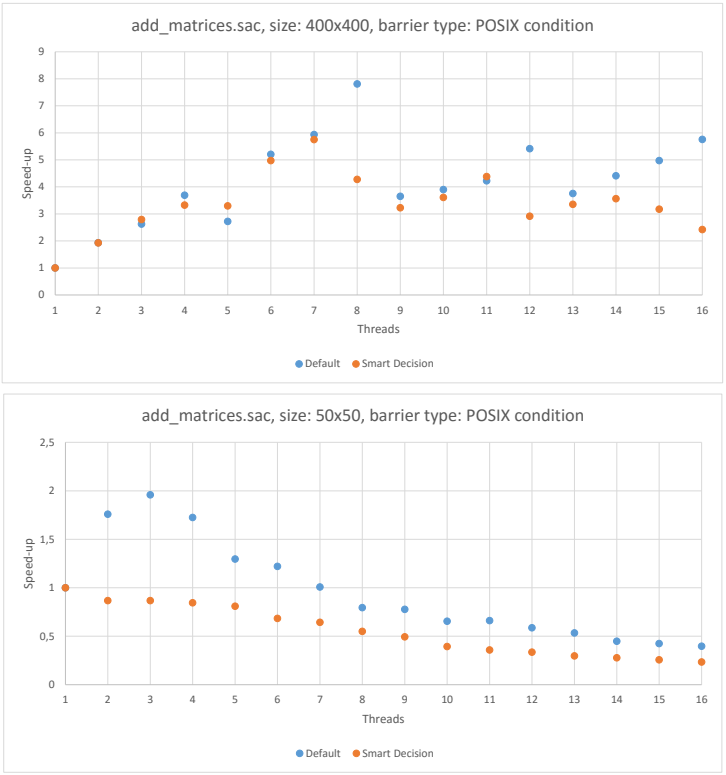


Fig. 14. Performance on Intel 8-core hyperthreaded system with and without the proposed smart decision tool for two different problem sizes and suspending barrier implementation; smart decision tool recommendations: 9 and 1

All approaches mentioned so far share a fundamental shortcoming: any data-parallel operation can only either be executed using all available worker threads or completely sequential by a single thread: a classical all-or-nothing decision.

The latest versions of OpenMP [] go one step beyond and introduce the `num_threads`-clause, which allows programmers to precisely specify the number of threads to be used for each parallelised loop, if they wish so. Like in the `if`-clause, the `num_threads`-clause contains an arbitrary C or Fortran expression that may access all program variables in scope. While the `num_threads`-clause is mainly motivated as a vehicle to express nested parallelism, it could also be used to explicitly set the number of threads in relation to problem set sizes or similar

runtime parameters of an application. However, programmers are completely on their own when using this feature of OpenMP.

This gap is filled by a multitude of performance analysis and tuning tools as for example Intel's VTune [21]. Corresponding guidelines [20, 9] explain the issues involved. These tools and methodologies indeed allow performance engineers to manually adapt effective parallelism in individual data-parallel operations to data set sizes and machine characteristics, but the process is highly labour-intensive if not to say painful. Furthermore, it needs to be repeated for every new architecture and data set size.

In contrast, our proposed approach for compiler-directed parallelisation in SaC works almost entirely automatically with the sole exception that users must explicitly compile their source for training and production mode and run training codes on representative input data. We would also like to remind the reader that our implicit approach works on the data-parallel operations in intermediate code *after* far-reaching code restructuring compiler optimisation. In contrast, any manual tuning approaches operate on the level of source code and either restrict the effectiveness and scope of compiler transformations or suffer from any decoupling between source and binary code.

In this sense, *feedback-driven threading* proposed by Suleman et al goes even one step further as a completely implicit compiler-based solution [29]. In an OpenMP-parallelised loop they peel off up to 1% of the initial iterations. They are executed by a single thread while hardware performance monitoring counters collect information regarding off-chip memory bandwidth and cycles spent in critical section. After this initial training phase the generated code evaluates the hardware counters and predicts the optimal number of threads to be used for the remaining bulk of iterations based on a simple analytical model. Apart from the obvious beauty of being completely transparent to users, this approach has some disadvantages. Considerable overhead is introduced at production runtime for choosing the (presumably) best number of threads. This needs to be offset first by more efficient parallel execution before any total performance gains can be realised. Peeling off and sequential execution of up to 1% of the loop iterations restricts potential gains of parallelisation according to Amdahl's law. This is a (high) price to be paid for every data-parallel operation, including all those that would otherwise perfectly scale.

In contrast to our approach that continuously accumulates insight into the scaling behaviour of individual data-parallel operations with every program run in training mode, Suleman et al do not carry over any information from one program run to the next and, thus, cannot reduce the overhead of feedback-driven threading. Last not least, they rely on the assumption that the initial iterations of a parallelised loop are representative in runtime behaviour for all remaining iterations, whereas we always measure the entire data-parallel operation. Therefore, we only reach the limits of reproducibility and predictability if the runtime behaviour critically depends on varying data being processed instead of data set sizes. But even in this adverse scenario increasing the number of training runs in

conjunction with the accumulation and averaging of profile data, we may come to meaningful conclusions.

Pusukuri et al proposed *ThreadReinforcer* [27]. While their motivation is similar, their proposed solution again differs in many aspects from what we propose. Most importantly, they treat any application as a black box and determine one single number of threads to be used consistently throughout the application's life time. In contrast, we approach the problem on the granularity of individual data-parallel operations and systematically vary the number of threads during an application's execution. Similar to Suleman et al, ThreadReinforcer integrates learning and analysis into application execution. This choice creates overhead, that only pays off for long-running applications. At the same time, having the analysis on the critical path of application performance immediately creates a performance/accuracy trade-off dilemma. In contrast, we deliberately distinguish between training mode and production mode in order to train an application as long as needed and to accumulate statistical information in persistent storage without affecting application production performance.

Another approach in this area is *ThreadTailor* [23]. Here, the emphasis lies on *weaving threads together* in order to reduce synchronisation and communication overhead where available concurrency cannot efficiently be exploited. This scenario differs from our setting in that we explicitly look into malleable data-parallel applications. Therefore, we are able to set the number of active threads to our liking and even differently from one data-parallel operation to another.

Much work on determining optimal thread numbers on multi-core processors, such as [22] or [1], stems from the early days of multi-core computing and are limited to the small core counts of that time. Furthermore, there is a significant body of literature studying power-performance trade-offs, among others [24, 6]. In a sense we also propose such a trade-off, but in our model performance and energy consumption are not competing goals. Instead, we aim at achieving runtime performance within a margin of the best performance observable with the least number of threads.

Coming back to SAC at last, we mention the work by Gordon and Scholz [10]. They aim at adapting the number of active threads in a data-parallel operation to varying levels of competing computational workload in an interactively configured multi-core system. The goal is to avoid thread context switches and thread migration and rather vacate cores that turn out to be oversubscribed by unrelated applications at program runtime. For this purpose they continuously monitor execution times of data-parallel operations, and upon observing significant changes in the performance level they adapt the number of threads of the running SAC application accordingly. Their work differs from our's not only in the underlying motivation, but likewise in the pure online approach. In contrast, we distinguish between training and production mode, focus on our own application's characteristics and assume that we face no significant external competing computational workload on our system, i.e. we rather look at batch-operated machines.

9 Conclusions and Future Work

Malleable data-parallel application programs offer interesting opportunities for compilers and runtime systems alike to adapt the effective number of threads separately for each data-parallel operation in order to achieve best runtime performance. Alternatively, a favourable trade-off between runtime performance and resource investment appears equally relevant these days. We explore this opportunity in the context of the functional data-parallel array language SaC and propose a combination of offline training that builds up a persistent profiling database, production code that incorporates gathered profiling data into recommendation tables and a runtime system extension that actually implements such recommendations during parallel program execution.

We are particularly concerned with data-parallel operations that turn out to be too small to make efficient use of all available cores on the system while their purely sequential execution still foregoes considerable performance gains. There are two variations of this scenario. Firstly, we see entire applications that could successfully exploit a certain number of cores in parallel execution but not the tens and hundreds of cores that computer architecture roadmaps promise for the foreseeable future. Secondly, data-parallel operations that only benefit from a limited number of cores could make part of larger applications that as a whole scale much better. Here, executing certain data-parallel operation with a single thread just as with too many threads reduces parallelisation gains on the whole application level. Our approach to control the number of threads on a per-operation basis is exactly geared at such cases.

Following our experimental evaluation in Section 7, we must admit that we are not there yet. While we are still convinced by our general approach, not to mention the relevance of the underlying problem, additional research is need to make our approach succeed. We have identified the following directions to take immediate action. We must make our approach more robust to training data that does not expose the shape shown in Fig. 1 as characteristically as we would wish. In particular we must develop robust methods to detect outliers.

Furthermore, we must refine our barrier implementations to activate worker threads more selectively. And we plan to explore how we can speed up suspension-based barriers in comparison to spinning barriers. A likely option to this effect would be to use hybrid barriers that spin for some configurable time interval before they suspend.

A particular problem that we underestimated at the beginning of our work is the by nature short execution time of the data-parallel operations for which our work is particularly relevant. Consequently, overall performance is disproportionately affected by synchronisation and communication overhead, thus pushing our second research direction sketched out above. In addition, short parallel execution times likewise incur a large relative variation. Although our offline training approach does take this into account, there are still practical limits to execution times in training mode and thus to averaging over many measurements. Moreover, a large variation also means that the average or median often is not a good representative of actual behaviour.

References

1. K. Agrawal, Y. He, W. Hsu, and C. Leiserson. Adaptive task scheduling with parallelism feedback. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing (PPoPP'06)*. ACM, 2006.
2. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, T. Schreiber, R. Simon, V. Venkatakrishnam, and S. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.
3. B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B. Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro*, 30(2):41–55, 2010.
4. B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
5. Cray Inc. *Chapel Language Specification Version 0.98.* Cray Inc., Seattle, USA, 2015.
6. M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *International Conference on Supercomputing (ICS'06)*, 2006.
7. L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Transactions on Computational Science and Engineering*, 5(1), 1998.
8. M. Diogo and C. Grelck. Towards heterogeneous computing without heterogeneous programming. In K. Hammond and H. Loidl, editors, *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK*, volume 7829 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2013.
9. M. Gillespie and C. Breshears. *Achieving Threading Success.* Intel Corp, 2005.
10. S. Gordon and S. Scholz. Dynamic control of runtime systems through a common interface. In R. Lämmel, editor, *Draft Proceedings of the 27th International Symposium on Implementation and Application of Functional Languages (IFL'15), Koblenz, Germany.* University of Koblenz-Landau, 2015.
11. C. Grelck. Implementing the NAS Benchmark MG in SAC. In V. K. Prasanna and G. Westrom, editors, *16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, USA*. IEEE Computer Society Press, 2002.
12. C. Grelck. A Multithreaded Compiler Backend for High-Level Array Programming. In M. H. Hamza, editor, *2nd International Conference on Parallel and Distributed Computing and Networks (PDCN'03), Innsbruck, Austria*, pages 478–484. ACTA Press, 2003.
13. C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.
14. C. Grelck. Single Assignment C (SAC): high productivity meets high performance. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer, 2012.
15. C. Grelck, S. Herhut, C. Jesshope, C. Joslin, M. Lankamp, S.-B. Scholz, and A. Shafarenko. Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors. In *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zürich, Switzerland*, 2009.

16. C. Grelck and S.-B. Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, 32(7+8):507–522, 2006.
17. C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.
18. C. Grelck and S.-B. Scholz. SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In *2nd Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France*, pages 25–33. ACM Press, 2007.
19. J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*, pages 15–24. ACM Press, 2011.
20. Intel. *Developing Multithreaded Applications: A Platform Consistent Approach*. Intel Corp, 2003.
21. Intel. *Threading Methodology: Principles and Practices*. Intel Corp, 2003.
22. C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing (PPoPP'05)*. ACM, 2005.
23. J. Lee, H. Wu, M. Ravichandram, and N. Clark. Thread Tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *International Symposium on Computer Architecture (ISCA'10)*, 2010.
24. J. Li and J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Symposium on High Performance Computer Architecture (HPCA'06)*, 2006.
25. T. Macht and C. Grelck. SAC goes cluster: From functional array programming to distributed memory array processing. In J. Knoop, editor, *Tagungsband des 18. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS 2015), Pörtschach am Wörthersee, Austria*. Technical University of Vienna, 2015.
26. J. Nieplosha et al. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *Computing Frontiers*, 2007.
27. K. Pusukuri, R. Gupta, and L. Bhuyan. Thread Reinforcer: Dynamically determining number of threads via OS level monitoring. In *IEEE International Symposium on Workload Characterization (IISWC'11), Austin, TX, USA*, pages 116–125. IEEE Computer Society, 2011.
28. S. Saini et al. A scalability study of Columbia using the NAS parallel benchmarks. *Journal of Computational Methods in Science and Engineering*, 2006.
29. M. Suleman, M. Qureshi, and Y. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII), Seattle, WA, USA*, pages 277–286. ACM, 2008.

Anything goes unless forbidden

Notes on synchronization and the operational semantics of a relaxed memory model

Daniel Schnetzer Fava,[1] Martin Steffen[1] and Volker Stolz[1,2]

[1] Dept. of Informatics, University of Oslo
[2] Western Norway University of Applied Sciences

Abstract. A *memory model* dictates which values may be observed when reading from memory, thereby regulating how concurrent processes communicate through shared memory.
In this note, we discuss a weak memory model for a calculus inspired by the Go programming language, focusing on buffered channel communication as the sole synchronization primitive. In contrast to an axiomatic semantics, we present an operational interpretation and discuss its rationale and its design.

1 Introduction

A *memory model* dictates which values may be observed when reading from memory, thereby affecting how concurrent processes communicate through shared memory. In this paper we discuss ideas behind an operational formalization of a memory model that mixes message passing synchronization with shared-variable communication. The formal model is inspired by the informal specification [14] of the Go programming language's [13, 9] memory model. In the following, we sketch the principles underlying our design choices. Up to recently, when it comes to memory models, axiomatic semantics have arguably received more focus than operational ones. We chose the *operational* approach and believe it leads to more intuitive interpretation of the model's guarantees. The semantics takes a perhaps unusual view on synchronization, stressing the following dichotomy between synchronization and communication:

> communication makes information observable, synchronization makes it *unobservable.*

Synchronization as constraint on observations Memory models regulate the interaction of multiple threads using shared memory. Often, this is formulated in an *observational* manner. It is not the realization of the memory that matters, it is what/which values can be encountered by threads interacting with memory. Interactions, in their most basic form, are elementary read and writes or loads and stores, while "observing" a value by a thread means reading it from memory. One general reason which complicates the situation is the asynchronous nature of interaction. For example, threads performing a read operation may not instantly observe values, as in the case of speculative execution. Similarly, writes done by a process do not immediately and not necessarily

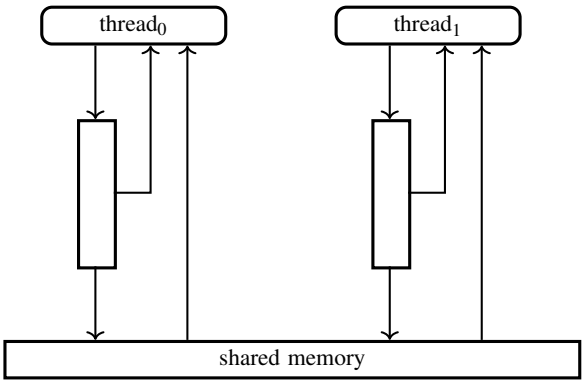


Fig. 1: Write buffers between processors and memory.

atomically change the status of a shared variable. When it comes to writes, this delay can be understood by the presence of write buffers, shown in Figure 1.

The picture is meant to only give a schematic impression. In hardware memory models, the non-instantaneous effect of writes is caused by complex interaction in the memory hierarchy. One complicating factor is the fact that memory, unlike in Figure 1, might not consist of a uniform global "data store." Instead, multiple copies of a shared variable may exist. Thus, it may not be appropriate to interpret a read-access to a variable as observing *the* variable's value, as there might be more than one copy. Different threads may read from different copies and observe different values.

Synchronization is what allows us to program on such slippery grounds. In very broad terms, synchronization between processes can be understood as *restricting* possible interleavings. Processes running independently in parallel, without synchronization, perform their respective steps unconstrained and undisturbed by each other. Synchronization means disallowing some of those interleavings, for example, blocking the progress of one process up until the other process has arrived at a certain point or produced some result. This interpretation casts synchronization as constraining the control flow of concurrently executing processes.

Communication, the exchange of data between processes, also needs consideration. For weak memory models and shared variable communication, synchronization is often "explained" as guaranteeing that data is available, thereby assisting communication. Synchronization statement like fences may understood as "flushing the write buffer." For example, with the execution of a write operation in Figure 1, a value is placed in the write buffers. Synchronization is then needed to make sure that the value becomes observable by other threads. See, for instance, a statement from an ARM programmer's guide talking about that a data memory barrier (DMB).[3]

> "... forces all earlier-in-program-order memory accesses to *become globally visible* before any subsequent accesses."

[3] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CJAIAJFI.html`

This view of synchronization as a mechanism for making writes *observable* is perfectly legitimate, of course, and may reflect the way hardware realizations of memory systems actually operate. Our semantics takes a *different view,* though. In line with seeing synchronization as restricting the control flow, our semantics formalizes synchronization as follows

a) for control: it prevents a process from proceeding (temporarily);
b) for data: it makes written values *unobservable*.

The formalization highlights a dichotomy between *communication* and *synchronization* and makes a sharp distinction of responsibilities: communication makes data available, while synchronization makes data *unavailable.* In these terms, reading and writing to memory is about communication only, not synchronization. Note that such interpretation is also implicit in the informal description of the happens-before memory model of Go [14], which takes a very liberal standpoint: any written value is observable unless and until it is made unobservable by synchronization. Such standpoint is what leads also to a very relaxed memory model.

Basically, the *only* mechanism with synchronizing power is *channel communication.* There are no fences, synchronized or volatile variables, locks,[4] semaphores, etc. Channels are thereby a mechanism used both for communication of data as well as synchronization. On the one hand, a channel transmits "positive" information, namely the data being relayed from sender to receiver. On the other, channels transmit "*negative*" information related to synchronization. Receiving is blocking, as usual; it prevents the receiver to proceed until a value is available to be received. Blocking covers the control-related part of synchronization. The other part, which has to do with shared variables and their observable values, is also affected by synchronization via channel communication.

Sequential consistency as baseline One of the simplest memory models, called *sequentially consistent*, stipulates that operations must appear to execute one at a time and in program order [17]. SC was one of the first formalizations and, to this day, constitutes a baseline for well-behaved memory. For efficiency reasons, however, modern hardware does not guarantee sequential consistency. SC is also considered much too strong to serve as the underlying memory semantics of programming languages; the reason being that sequential consistency prevents many established compiler optimizations and robs from the compiler writer the chance to exploit the underlying hardware for efficient parallel execution. The research community has struggled to agree on what, exactly, a proper memory model should offer. Consequently, a bewildering array of *weak* or *relaxed memory models* have been proposed, investigated, and implemented. Different taxonomies and catalogs of so-called *litmus tests*, which highlight specific aspects of memory models, have also been researched [1].

In light of these difficulties and despite many attempts, there does not exist a well-accepted comprehensive specification of the C++11 [5, 6] or Java memory models [4, 18, 20]. Despite the existence of broadly disparate approaches towards memory and

[4] Except that locks are available via libraries.

its formalization, the following principle of relaxed memory has gained overwhelming acceptance: regardless of how much relaxation is permitted by the memory model, if a program is *data-race free* or *properly synchronized*, then the memory model must ensure that the program behaves sequentially consistently [2, 18]. This principle is known as the *SC-DRF* guarantee.

Operational semantics We sketch an *operational* semantics for a weak memory. Our calculus is inspired by the Go programming language: similar to Go, our model focuses on channel communication as the main synchronization primitive. Go's memory model, however, is described, albeit succinctly and precisely, in prose [14]. We provide a formal semantics instead. In this paper, we informally discuss ideas underlying the semantics and possible extensions. We refer the reader to [11] and the technical report [12] for more details.

2 Background

Go's memory model Concerning synchronization primitives, the model covers goroutine creation and destruction, channel communication, locks, and the `once`-statement. Our semantics will concentrate on thread creation and channel communication because lock-handling and the `once` statement are *not* language primitives but part of the `sync`-library. Thread destruction, i.e. termination, comes with *no* guarantees concerning visibility: it involves no synchronization and thus the semantics does not treat thread termination in any special way. In that sense, our semantics treats all of the *primitives* covered by Go's memory model specification. As will become clear in the next sections, our semantics does not, however, relax read events. Therefore, our memory model is stronger than Go's. On the plus side, this prevents a class of undesirable behavior called *out-of-thin-air* [7]. On the negative, the absence of relaxed reads comes at the expense of some forms of compiler optimizations.

Happens-before relation and observability Like Java's [18, 20], C++11's [5, 6], and many other memory models, ours centers around the definition of a *happens-before* relation. The concept dates back to [16] and was introduced in a pure *message-passing* setting, i.e., without shared variables.[5] The relation is a technical vehicle for defining the semantics of memory models. It is important to note that just because an instruction or event is in a *happens-before* relation with a second one, it does not necessarily mean that the first instruction *actually* "happens" before the second in the operational semantics. Consider the sequence of assignments $x := 1; y := 2$ as an example. The first assignment "happens-before" the second as they are in program order, but it does not mean the first instruction is actually "done" before the second,[6] and especially, it does not mean that the effect of the two writes become observable in the given order. For example, a compiler might choose to change the order of the two instructions. Alternatively, a processor may rearrange memory instructions so that their effect may not

[5] The relation was called happened-before in the original paper.

[6] Assuming that x and y are not aliases in the sense that they refer to the same or "overlapping" memory locations.

be visible in program order. Conversely, the fact that two events happen to occur one after the other in a particular schedule does not imply that they are in happens-before relationship, as the observed order may have been coincidental. To avoid confusion between the technical happens-before relation and our understanding of what happens when the programs runs, we speak of event e_1 "happens-before" e_2 in reference to the technical definition (also abbreviated as $e_1 \rightarrow_{\mathsf{hb}} e_2$ as opposed to its natural language interpretation. Also, when speaking about steps and events in the operational semantics, we avoid talking about something happening before something else, and rather say that a step or transition "occurs" in a particular order.

The happens-before relation regulates observability, and it does so very liberally. It allows a read r from a shared variable to *possibly observe* a particular write w to said variable *unless* one of the following two conditions hold:

$$r \rightarrow_{\mathsf{hb}} w \quad \text{or} \tag{1}$$

$$w \rightarrow_{\mathsf{hb}} w' \rightarrow_{\mathsf{hb}} r \quad \text{for some other write } w' \text{ to the same variable.} \tag{2}$$

For the sake of discussion, let us concentrate on the following two constituents for the happens-before relation: 1) *program order* and 2) the order stemming from channel communication.[7] According to the Go memory model [14], we have the following constraints related to a channel c with capacity k:

$$\text{A send on } c \text{ happens-before the corresponding receive from } c \text{ completes.} \tag{3}$$

$$\text{The } i^{th} \text{ receive from } c \text{ happens-before the } (i+k)^{th} \text{ send on } c. \tag{4}$$

Condition (4) accounts for the boundedness of channels by transmitting happens-before information in the backward direction for some receiver to some sender. Note that for *synchronous* channels, which have capacity zero, conditions (3) and (4) degenerate: the send and receiving threads participate in the rendezvous and symmetrically exchange their happens-before information.

In summary, the operational semantics captures the following principles:

Immediate positive information: a *write* is globally observable instantaneously.

Delayed negative information: in contrast, negative information overwriting previously observable *writes* is *not* immediately effective. Instead, the information is spread via message passing in the following way:

- **Causality:** information regarding condition (3) travels with data through channels.
- **Channel capacity:** *backward channels* are used to account for condition (4).

Local view: Each thread maintains a local view on the happens-before relationship of past write events, i.e. which events are unobservable. Thus, the semantics does not offer multi-copy atomicity [8].

3 Axiomatic semantics and litmus tests

To position the work in a slightly broader context, we revisit notions from axiomatic semantics of memory models. In particular, we use well-known litmus tests to highlight similarities and differences between our semantics and alternative ones. In the

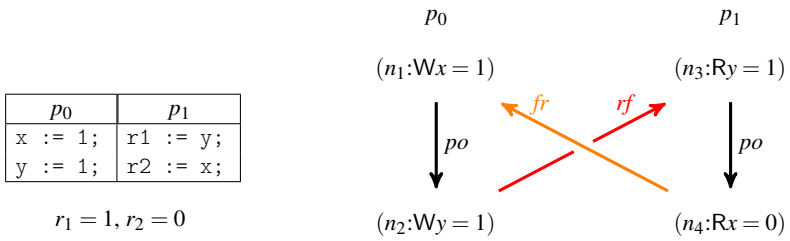
[7] There are additional conditions in connection with channel creation and thread creation, the latter basically a generalization of program order; we ignore it in the discussion here.

discussion, we sometimes refer to [3], which not only contains a general and elaborate axiomatic semantics, but also collects numerous litmus tests for illustrating their axiomatic framework. Other relevant related work is captured in the tutorial [19].

Axiomatic semantics operates on *event graphs*, i.e., graphs with events as nodes and various relations between the events as edges. Events correspond to occurrences of executed instructions, with loads and stores as the most basic form of memory interaction. Relations capture various forms of dependencies between instructions; for example, data dependencies and intra-process control-flow dependencies such as program order. The graph, in particular the relationship between the events, depends on the program code itself and also on the memory model. One program typically gives rise to more than one such graph, each of which represents one possible "run" of a program. Still, such graphs only represent *candidate* executions, as unrealizable graphs are filtered out by axioms. The axioms spell out conditions on the various relations, typically requiring *acyclicity* of particular combinations of the involved edges. Aspects of a memory model are often captured or illustrated by so-called litmus tests, which are tailor-made code snippets used to highlight expected or disallowed behavior in a given setting. Thus, a litmus test can also be seen as a partial pre- and post-specification for a small piece of concurrent code. The precondition, often left implicit, assumes that all shared variables are in some definite, initial state.

As illustration, Figure 2 contains the well-known litmus test for "message passing" on the left and a corresponding candidate execution on the right. The crucial question for the given code is whether the observation $r_1 = 1$ and $r_2 = 0$ is possible. The intention of the code is the following: process p_0 wants to send data via x to p_1 and uses a write to y to signal that the value is "ready" to be read. In a memory model where the litmus test of Figure 2 is expected to work without additional synchronizing instructions, the observation $r_1 = 1$ and $r_2 = 0$ must be *forbidden*. The assumption, in this case, is that the order of reads by p_1 reflects the order in which the writes are effected (when done in program order) by p_0. Under this assumption, one can generalize the pattern in that p_1 repeatedly reads the flag variable y in a busy wait loop until it is assured that the value communicated via x is ready to be read.

In moderately weak memory models, ones with per-location write buffering in particular, message passing is not realizable without additional synchronizing instructions. The candidate execution of Figure 2b shows exactly that; it gives a justification for the observation $r_1 = 1$ and $r_2 = 0$, which violates the MP pattern. The edge $n_2 \rightarrow_{\mathsf{rf}} n_3$ of the "read-from" relation $\rightarrow_{\mathsf{rf}}$ simply expresses the fact that n_3 reads the value written by n_2. More complex is the "from-read" relation: the edge $n_4 \rightarrow_{\mathsf{fr}} n_1$ stipulates that n_4 "reads-from" some write event left unmentioned for which n_1 comes "after." More precisely, it abbreviates $n_0 \rightarrow_{\mathsf{rf}} n_4$ for some write event n_0 with $n_0 \rightarrow_{\mathsf{co}} n_1$ and where $\rightarrow_{\mathsf{co}}$ represents the so-called *coherence order*, which is a total order of writes over the same memory location. In the example, n_0 is the write event setting x to its initial value 0 (which, by convention, is often left out in representations of candidate executions). Using the mentioned coherence order, the from-read relation captures the intuition that a read observes a value which is *not* yet overwritten by a given write. In contrast, in our setting as well as in others, the information on which writes are observable by a read is *local* per observing thread. Conceptually, there is no notion of a total order of writes



p_0	p_1
`x := 1;`	`r1 := y;`
`y := 1;`	`r2 := x;`

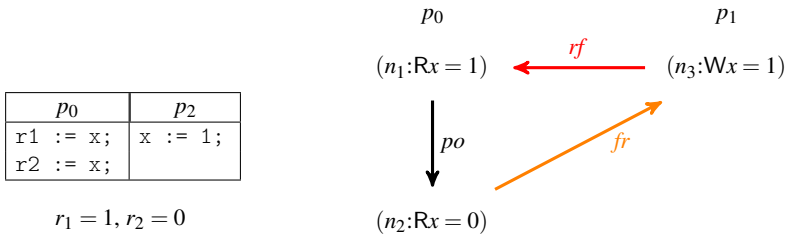
$r_1 = 1, r_2 = 0$

(a) Litmus test (b) Candidate execution

Fig. 2: mp

on a location. Past writes events are seen as unordered even if issued by one thread or when seen from the perspective of one observing thread.[8]

One, perhaps unfamiliar, aspect of our semantics is that writes, once performed, never invalidate earlier writes (at least those done by a thread different from the observer). In the absence of synchronization, writes remain observable indefinitely. A litmus test typifying that kind of behavior is known as coRR,[9] shown in Figure 3.



p_0	p_2
`r1 := x;`	`x := 1;`
`r2 := x;`	

$r_1 = 1, r_2 = 0$

(a) Litmus test (b) Candidate execution

Fig. 3: coRR

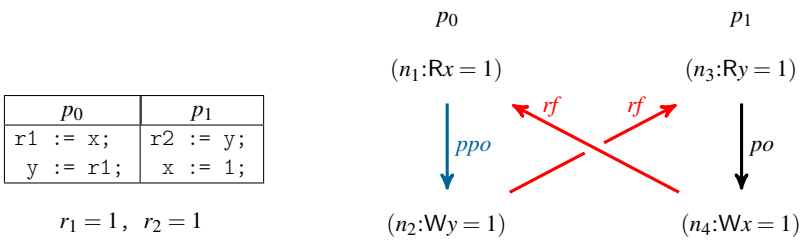
The fact that repeated reads by the same thread give different seemingly incoherent values may seem odd at first. It can, however, be interpreted as a form of oscillation: when reads and writes happen "at the same time," i.e., in a racy way without proper synchronization, the memory can be understood as oscillating between the racy memory updates. In the example, the value can, in theory, be perceived as oscillating between 0 and 1 indefinitely. This behavior is allowed by our proposed semantics. As a matter

[8] Except in the special single threads case, where the reads and writes are done by the same thread.

[9] In general, coherence tests coXY involve an access of kind X and an access of kind Y with X and Y standing for either R (read) or W (write).

of fact, it is also officially allowed by Sparc RMO [15] and pre-Power4 machines [21]. Many other models, however, including the axiomatization by [3], disallow it.

Note that our treatment of write-accesses is rather liberal and therefore, some of the litmus tests for write buffering also characterize our semantics. For example, both our model and PSO-style memory models with per-location write buffers allow the observation $r_1 = 1$ and $r_2 = 0$ in the mp litmus test of Figure 2. On the other hand, from our perspective, the treatment of the writes is best not seen as "buffering;" after all, the value of a write in the operational semantics becomes *immediately* observable. It is the *negative* information of being *unobservable* that is not immediately effective for all observers. This negative information requires synchronization via channel communication in order for it to percolate though the concurrent system.



(a) Litmus test (b) Candidate execution

Fig. 4: Load buffers (lb)

Our basic weak semantics treats writes in a rather relaxed manner. Reads, in contrast, are treated in a conventional or "strong" way. *Load buffering* is a relaxation which complements write buffering. The effect of load buffers is often illustrated by the litmus test of Figure 4. The candidate execution graph on the right shows a run which justifies $r_1 = 1$ and $r_2 = 1$. This execution can be interpreted as follows: the load or read of event n_1 is buffered, thereby taking effect after the write event n_4. This buffering causes the instructions n_3 and n_4 to seem executed *out-of-order*, thus, the program order $n_3 \rightarrow_{\mathsf{po}} n_4$ is perceived as "ignored." For p_0, however, the read cannot be postponed until after the write instruction, as the value of the write *depends* (via r_1) on the value being read: the read and write events in p_0 are in program order, but unlike the situation in p_1, the load cannot be buffered; program order has to be preserved due to a data dependence. The preserved program order is marked in the graph by a $\rightarrow_{\mathsf{ppo}}$-edge. The circumstances in which program order is preserved or not depends on the programming language semantics and/or the given hardware memory model. For example, various forms of special fence instructions (e.g. light-weight fences, full fences, control fences) may be available on a given platform.

Load buffering is conceptually more challenging than write buffering. Thinking operationally, dispatching a write instruction in an asynchronous manner is like "fire-and-forget," but executing an "asynchronous" read means the corresponding process

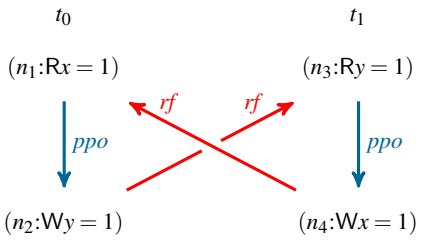
continues regardless of whether the value it wishes to read has been obtained. This non-blocking nature in particular problematic if it is assumed (as in our happens-before model) that reading a value, buffered or not, is done *without* any synchronization.[10] Subsequent code may *depend* on the value being read; the dependency may not only be a data-dependency (as the write to y in Figure 4a), but also a control flow dependency, where choosing a branch to be taken depends on a value that is not yet available.

One important aspect in connection with load buffering is illustrated in Figure 5. It closely resembles the previous case from Figure 4. In particular, the four memory events in the candidate execution graph in Figure 5b are *identical* to those in Figure 4b. The crucial difference is an additional data dependency in p_1: the write statement has a data dependency on the preceding read event. This dependency is reflected in the graph by a *ppo*-edge, as opposed to a *po*-edge as in Figure 4b.

p_0	p_1
`r1 := x;`	`r2 := y;`
`y := r1;`	`x := r2;`

$r_1 = 1 = r_2$ (out-of-thin-air)

(a) Litmus test



(b) Candidate execution

Fig. 5: lb+ppos

The outcome $r_1 = 1 = r_2$ of the litmus test could be justified by the following argument, as illustrated by the candidate execution graph: n_1 reads the value 1 written by n_4, subsequently used in the write n_2, which in turn is read by n_3, and used in the write event n_4. This justification involves a circular argument, and it is able to produce a value, the number 1, that does not even appear in the program text. Such behavior is termed "out-of-thin-air" and is generally, though not universally, considered illegal. In other words, the *candidate* graph of Figure 5b, which justifies the post-condition of the litmus test, is a candidate that is ruled out by many memory models. Note, however, that in the informal happens-before Go memory model, out-of-thin-air behavior of this kind is allowed, as there are no statements or mechanisms specified which forbid the behavior.

Note that the candidate execution from Figure 5b is, for *that* model, not significant. In particular the shown $\rightarrow_{\mathsf{ppo}}$-edges presuppose that the order of the reads and the subsequent write cannot be changed, i.e., due to the data dependency of the litmus test, there is no conceptual load-buffering in that particular situation. The Go memory model, on the other hand, does not mention data dependencies nor "preserved program order."

[10] Remember, that basically the *only* way of achieving synchronization is channel communication.

Instead, it operates with the plain notion of program order $\rightarrow_{\mathsf{po}}$, stipulating that $\rightarrow_{\mathsf{po}} \subseteq \rightarrow_{\mathsf{hb}}$. Therefore, in the situation of Figure 5, the out-of-thin-air observation is perfectly acceptable.[11] Certainly, the happens-before relation is assumed to be acyclic,[12] but still, the situation of Figure 5b does not constitute a happens-before cycle as $\rightarrow_{\mathsf{hb}}$ is not defined making use of the read-from relation. As a final remark, it should be noted that in [3] and similar sources, the happens-before relation is a *derived* relation. The relation is defined as an extension of preserved program order $\rightarrow_{\mathsf{ppo}}$, taking into account read-from-edges. In our discussion, however, the happens-before relation consists of $\rightarrow_{\mathsf{po}}$ and the dependencies implied by channel synchronization. The condition (3) can be interpreted as analogue to a $\rightarrow_{\mathsf{rf}}$-edge, though based on channel communication as opposed to communication via an unsynchronized shared variable.

4 Memory as collection of past write events

In this section we sketch the operational formalization of our memory model by highlighting a few of its central rules. We slightly simplify the rules and favor communicating intuition over notational rigor.

The model is formulated as structural operational semantic rules over program configurations. Our configurations contain "sets" of processes or threads, channels, and a "memory component." Memory consists of the collection of past write events. The write events remembered in the configuration are written as $n(\!|z{:=}v|\!)$ with n as unique identifier. To regulate observability in accordance to the happens-before relation, each thread keeps track of whether write events are, from the perspective of the given thread, *unobservable* (we use "unobservable" and "shadowed" interchangeably). Additionally, threads keep track of write events that are locally known to be in happens-before relation to their current point of execution. Happens-before and shadowed sets form a tuple (E_{hb}, E_s) which is referred to as "local state" and denoted as σ in the rules of the operational semantics.

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (n,z), E_s + E_{hb}(z)) \qquad \mathit{fresh}(n)}{p\langle \sigma, z := v; t\rangle \;\rightarrow\; p\langle \sigma', t\rangle \parallel n(\!|z{:=}v|\!)} \text{ R-WRITE}$$

$$\frac{\sigma = (\_, E_s) \qquad n \notin E_s}{p\langle \sigma, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel n(\!|z{:=}v|\!) \;\rightarrow\; p\langle \sigma, \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle \parallel n(\!|z{:=}v|\!)} \text{ R-READ}$$

Table 1: Operational steps: reads and writes

Table 1 shows the rules covering reads and writes. Executing a write instruction spawns a new corresponding write event and updates the local state σ of the execut-

[11] That is not to say that language implementations will exhibit that behavior, just that out-of-thin-air is consistent with the specification.

[12] Note that it is not stated explicitly in [14] but it is safe to assume that is what is intended.

ing thread. The update records information about the write event as part of the issuing thread's happens-before memory. Additionally, when it comes to the variable in question, all events that are known to have happened-before are then marked as shadowed. In a single-threaded setting, this book-keeping realizes visibility according to program order, which means, the only value observable for each variable is the one written last. This behavior is what one would expect from a sequential program.

Channels communication is responsible not only for message passing communication but also for synchronization between threads. The synchronization conditions are spelled out in equations (3) and (4). Equation (3) corresponds to the fact that receiving over a channel is a potentially blocking operation: the receiver has to wait until a value is available. This blocking is a manifestation of the control-flow aspect of synchronization, as it disables, under certain conditions, steps from being taken. More concretely, equation (3) connects the steps of the sender before the send with the steps in the receiver after the reception; this connection is made by the $\rightarrow_{\mathsf{hb}}$-relation. In the operational interpretation of our semantics a channel send and receive communicates not just the communicated value, but also the local state σ from the sender to the receiver (see table 2). Upon reception, the receiver updates its local state with the new information. This update is reflected by setting $\sigma' = \sigma + \sigma''$ as shown in rule R-REC. With the update, the receiver increases its knowledge about which events have happened-before, thereby increasing the events which become *un*observable from the receiver's perspective.

Equation (4) works in reverse direction and represents synchronization due to the *boundedness* of channels. Boundedness connects the happens-before knowledge of a sender with that of a receiver on the same channel. Note that this connection is not necessarily from the receiver of a value back to the sender of said value: passing a message from a process p_1 to a process p_2 via a channel c makes the send of p_1 to happen-before an earlier receive on c. This earlier receive was not necessarily done by p_2; it could have been done by a "third party" p_3. In the operational semantics, this "receiver-to-sender" synchronization is captured by the transmission of local state σ in the reverse direction. Each program level channel is thus represented by two channels, a "forward" channel, transmitting in a conventional manner the data being transmitted (plus the happens-before and shadow information) and a "backward" channel (transmitting only happens-before and shadow information). Thus, receiving a value in rule R-REC not just dequeues one entry from the forward queue, but enqueues also the receiver's current σ into the backward channel. Correspondingly, a send in rule R-SEND is enabled only, if the backward queue is not empty.

Communication via synchronous channels, which can be seen as channels with capacity 0, is treated separately in rule R-SEND-REC. This rule leads to an immediate exchange of the local states between sender and receiver. Synchronous communication, therefore, corresponds to a full bidirectional fence between the two partners engaged on a rendezvous. There are additional rules, left out here, that deal with creating and closing channels.

$$\frac{\neg closed(c_f[q_2]) \qquad \sigma' = \sigma + \sigma''}{c_b[q_1 :: \sigma''] \parallel p\langle \sigma, c \leftarrow v; t\rangle \parallel c_f[q_2] \;\rightarrow\; c_b[q_1] \parallel p\langle \sigma', t\rangle \parallel c_f[(v,\sigma) :: q_2]}\ \text{R-SEND}$$

$$\frac{v \neq \bot \qquad \sigma' = \sigma + \sigma''}{\begin{array}{l} c_b[q_1] \parallel p\langle \sigma, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle \parallel c_f[q_2 :: (v,\sigma'')] \;\rightarrow \\ c_b[\sigma :: q_1] \parallel \; p\langle \sigma', \texttt{let}\ r = v\ \texttt{in}\ t\rangle \;\parallel c_f[q_2] \end{array}}\ \text{R-REC}$$

$$\frac{\sigma' = \sigma_1 + \sigma_2}{\begin{array}{l} c_b[] \parallel p_1\langle \sigma_1, c \leftarrow v; t\rangle \parallel p_2\langle \sigma_2, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t_2\rangle \parallel c_f[] \;\rightarrow \\ c_b[] \parallel \quad p_1\langle \sigma', t\rangle \quad \parallel p_2\langle \sigma', \texttt{let}\ r = v\ \texttt{in}\ t_2\rangle \quad \parallel c_f[] \end{array}}\ \text{R-SEND-REC}$$

Table 2: Operational steps: basic channel communication

5 Load buffering

In order to further relax the memory model described in the previous section, we need to introduce load buffering, also referred to as *delayed reads*. The presence of load buffering, however, gives rise to a number of complications. For one, reads have to be executed *asynchronously:* a thread reading a value has to proceed even if the value is not yet available. Such "non-blocking" reads are problematic in that a thread's continuation after an asynchronous read typically depends on the value read. Process p_0 in the code of Figure 4a and both processes in Figure 5a contain examples of this type of dependency. In these examples, the dependency is a pure *data dependency*: a "subsequent" instruction writes the value read back to memory.

Control dependencies, which have not yet been discussed in connection with the examples from Section 3, are more complicated than data dependencies. A control dependency happens, for example, when the condition of an if-then-else involves the value read by a prior load-instruction. If the load is executed "out-of-order" and after the conditional expression, then both branches are in principle possible. This superposition leads to a form of *speculative* execution.

The asynchronous execution of a load instruction is similar to a simple *future*-like mechanism; one involving *implicit* futures. While futures allow the asynchronous execution of arbitrary sequential code that produces an eventual value, asynchronous reads execute only the `load`-instruction and nothing else. We say that the load is done, or the "future" is resolved, when a concrete value can be passed back to issuer of the asynchronous read.

Asynchronous loads and futures differ in terms of synchronization. Conventionally, futures involve a form of synchronization that is sometimes called "wait-by-necessity," meaning that the caller blocks when accessing results that are not yet available. With its strict separation between synchronization (achieved by sending-to and receive-from channels) and communication, reads and writes are *completely devoid* of synchronization. Informally, the lack of synchronization can be seen as a "don't-wait-not-even-

when-necessary" semantics. In other words, a read and a subsequent write —subsequent in terms of program order— can be swapped and executed out-of-order. Thus, as an example, the "preserved-program-order" edge in Figure 4b would be represented in our model as $\rightarrow_{\mathsf{po}}$-edge instead.

To delay the read and thereby account for out-of-order execution, configurations are extended with read events (in addition to the write events already discussed in the previous section). The semantics is also extended to deal with references to future values. For example, before adding delayed reads, store instructions took the form $z := v$ where v denotes a user-level data value such as an integer. Once delays are added, a new form of the store instruction is needed: $z := n$ with n a "future reference" to an asynchronous read. Note that, as shown in rule R-READ, future references are written $n[\![\sigma, \texttt{load}\ z]\!]$.

When it comes to ensuring that "outdated" or shadowed writes are not observable by a read instruction (see equation (2)), the semantics with asynchronous reads differs from the semantics without them. In the presence of asynchronous reads, the future reference $n[\![\sigma, \texttt{load}\ z]\!]$ is responsible for making sure that shadowed writes are not observed. In other words, the future reference must ensure that the value resulting from an asynchronous read does not come from a write that is shadowed to the reader. In the semantics without asynchronous reads, a thread's shadow set was used to exclude visibility of shadowed writes. This exclusion is captured by the corresponding premise of the R-READ rule of Table 1.

When it comes to causality (see equation (1)), the semantics lacking asynchronous reads trivially guarantees that a read instruction cannot observe writes that happen-after. Once asynchronous reads are introduced, however, additional measures must be taken so causality is not violated. Note that the σ-information of the reader is of no help here: in the operational execution, the write event is generated *after* the asynchronous issuing of $n[\![\sigma, \texttt{load}\ z]\!]$. Consequently, the subsequent write is unknown to $n[\![\sigma, \texttt{load}\ z]\!]$ and, therefore, the write is not mentioned in σ. In general, the fact that a read event does *not "know"* about a write event is useless to determine whether the read can observe said write event (as both is consistent with being not mentioned in the local σ of the read event). Instead, it is from the perspective of the write-event that one can determine whether a read can observe the write: if the write event "knows" that a read event has "happened-before," then the write should not be made observable to the read. Thus, write events now also carry happens-before and shadow information and are of the form $n(\!|\sigma, z := v|\!)$. Note also that the conditions corresponding to equations (1) and (2) are captured by the premises of rule R-OBS from Table 3.

The implicit future references not only occur on the right-hand side of the let-construct (see rule R-READ), but can also occur as stand-in for the value to be loaded from memory. In particular, assignments may take now the form $z := n$ where n is a reference to a value that has not yet been resolved. Consequently, write events may take the form $n_1(\!|\sigma, z := n_2|\!)$ and delayed read constructs can take the form of a reference to a reference: $n_1[\![n_2]\!]$. As references are resolved into concrete values, the semantics must have rules to shorten chains of indirections. For instance, $n_1(\!|\sigma, z := n_2|\!) \parallel n_2[\![v]\!] \rightarrow n_1(\!|\sigma, z := v|\!) \parallel n_2[\![v]\!]$. Additionally, the semantics needs to deal with compound expressions containing future references, which likewise cannot be immediately evaluated. The corresponding dereferencing rules are left out in this discussion. Also left out of

$$\frac{\mathit{fresh}(n)}{p\langle\sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \ \rightarrow\ p\langle\sigma, \texttt{let}\ r = n\ \texttt{in}\ t\rangle \parallel n[\![\sigma, \texttt{load}\ z]\!]}\ \text{R-READ}$$

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (n,z), E_s + E_{hb}(z)) \qquad \mathit{fresh}(n)}{p\langle\sigma, z := v; t\rangle \ \rightarrow\ (p\langle\sigma', t\rangle \parallel n(\!|\sigma, z := v|\!))}\ \text{R-WRITE}$$

$$\frac{\sigma_1 = (\_, E_s^1) \qquad \sigma_2 = (E_{hb}^2, \_) \qquad n_2 \notin E_s^1 \qquad n_1 \notin E_{hb}^2}{n_1[\![\sigma_1, \texttt{load}\ z]\!] \parallel n_2(\!|\sigma_2, z := v|\!) \ \rightarrow\ n_1[\![v]\!] \parallel n_2(\!|\sigma_2, z := v|\!)}\ \text{R-OBS}$$

Table 3: Operational semantics: shared memory

this note is the exact treatment of *control* dependencies in the presence of delayed reads. When the decision on which branch to take depends on the value of a read that is not yet resolved, the semantics allows execution to continue in a speculative manner, collecting a symbolic representation of the branch condition in the form of a path condition.

Example 1 (Out-of-thin-air). For illustration, let us revisit the litmus test from Figure 5a. Assume that both threads start with a local state of σ_0. We don't show the write events for initialization of the two shared variables. Now, after executing four instructions, the resulting configuration may continue as follows, dereferencing the load reference:

$$\begin{aligned} &p_1\langle\sigma_1, n_1\rangle \parallel n_1[\![\sigma_0, \texttt{load}\ x]\!] \parallel n_1'(\!|\sigma_0, y := n_1|\!) \parallel n_2[\![\sigma_0, \texttt{load}\ y]\!] \parallel n_2'(\!|\sigma_0, x := n_2|\!) \parallel p_2\langle\sigma_2, n_2\rangle \ \rightarrow \\ &p_1\langle\sigma_1, n_1\rangle \parallel n_1[\![n_2]\!] \parallel n_1'(\!|\sigma_0, y := n_1|\!) \parallel n_2[\![\sigma_0, \texttt{load}\ y]\!] \parallel n_2'(\!|\sigma_0, x := n_2|\!) \parallel p_2\langle\sigma_2, n_2\rangle \ \rightarrow \\ &p_1\langle\sigma_1, n_1\rangle \parallel n_1[\![n_2]\!] \parallel n_1'(\!|\sigma_0, y := n_1|\!) \parallel n_2[\![n_1]\!] \parallel n_2'(\!|\sigma_0, x := n_2|\!) \parallel p_2\langle\sigma_2, n_2\rangle \end{aligned}$$

After these two dereferencing steps, the configuration contains $n_1[\![n_2]\!]$ and $n_2[\![n_1]\!]$, each attempting to solve each other's indirect reference in some form of *deadlock*. Adding a rule that "resolves" such cyclic dependencies by picking a random value (in a form of self-justification) allows the modeling of out-of-thin-air behavior. □

6 Conclusion

We present the ideas behind an *operational* specification for a weak memory model. The semantics is accompanied by an implementation in the $\mathbb{K}$ framework and by several examples and test cases [10]. We plan to use the implementation towards the verification of program properties such as data-race freedom.

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Research Report 95/7, Digital WRL, Sept. 1995.

[2] S. V. Adve and M. D. Hill. Weak ordering — a new definition. *SIGARCH Computer Architecture News*, 18(3a):2–14, 1990.

[3] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM TOPLAS*, 36(2), 2014.

[4] D. Aspinall and J. Ševčík. Java memory model examples: Good, bad and ugly. *Proc. of VAMP*, 7, 2007.

[5] Programming languages — C++. ISO/IEC 14882:2001, 2011.

[6] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2008.

[7] H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 7:1–7:6, New York, NY, USA, 2014. ACM.

[8] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992.

[9] A. A. A. Donovan and B. W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2015.

[10] D. Fava. Operational semantics of a weak memory model with channel synchronization. `https://github.com/dfava/mmgo`, Oct. 2017.

[11] D. Fava, M. Steffen, and V. Stolz. Operational semantics of a weak memory model with channel synchronization. In K. H. et.al., editor, *FM*, volume 10951 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag, July 2018.

[12] D. Fava, M. Steffen, and V. Stolz. Operational semantics of a weak memory model with channel synchronization: Proof of sequential consistency for race-free programs. Technical Report 477, University of Oslo, Faculty of Mathematics and Natural Sciences, Dept. of Informatics, Jan. 2018. Available at `https://www.duo.uio.no/handle/10852/61977`.

[13] The Go programming language specification. `https://golang.org/ref/spec`, Nov. 2016.

[14] The Go memory model. `https://golang.org/ref/mem`, 2014. Version of May 31, 2014, covering Go version 1.9.1.

[15] S. I. Inc and D. L. Weaver. *The SPARC architecture manual*. Prentice-Hall, 1994.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[18] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05*. ACM, Jan. 2005.

[19] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models (version 120), Oct. 2012.

[20] W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM Java Grande Conference*, June 1999.

[21] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Q. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

Beiträge aus den Jahren 2016 und 2017

Translets — Parsing Diagnosis in Small DSLs, with Permutation Combinator and Epsilon Productions

(Bad Honnef 2016)

Markus Lepper[1] and Baltasar Trancón y Widemann[1,2]
post@markuslepper.eu
Baltasar.Trancon@tu-ilmenau.de

[1] <semantics/> GmbH, Berlin, DE
[2] Ilmenau University of Technology, Ilmenau, DE

Abstract. Translets is a light-weight embedded domain-specific language for easy implementation of small text parsers, which serve as the front-ends of light-weight non-embedded domain-specific languages. In many situations these are required for the processing of readable and convenient denotations of values from complex, domain-specific data types. Translets is a parser combinator library to enable also domain experts, who are neither computing scientists nor language experts, to design small, ad-hoc languages for their realm. This is supported by these design criteria:
(1) Both sides, the syntax of the input and the data type of the generated parser's output, are defined together and in strict correspondence, thus explaining each other;
(2) parsing handles non-determinism and hides most technical details from the user;
(3) the resulting data is immediately accessible after parsing, without further transformation steps or analysis.
(4) extensive error messages are generated in case of parsing failure, and detailed diagnostic information can be retrieved also in case of success.
The synthesis of sensible error messages is a challenge esp. in combination with permutation parsers and epsilon productions and is the main contribution in this article. Translets have been successfully employed in widely varying contexts, from graphic animation scripts to conventional music notation.

KEYWORDS: parser combinators, parsing error messages, embedded domain-specific language, Java generics, type inference

1 Introduction

1.1 Domain-Specific Languages and Domain Experts

In the recent decades Domain-Specific Languages, DSLs, have become more and more popular. They intend to give domain experts the opportunity to describe a model, a situation, a problem etc., of their specific realm in a form readable by both, computers and humans, for further automated processing in very different directions, from simple storing and retrieving up to simulating, solving and reasoning.

This development is enabled by increasing processing power, and driven by the experience that the graphical description methods, using dragging and dropping, drawing and painting, etc., which had been very popular in the beginning of computer application, are sufficient in many cases, but not suited to express more formal or complicated relationships, which are much more naturally and easily expressed by *text*. Also the insight that properties like parameterizable genericity, strict typing, lexical scoping, modularity etc. are indispensable, at least in larger project contexts, makes text based approaches look superior.

The DSL approach is an emancipatory one, because it aims at enabling also those scientists who are no experts in computing science to operate with the computer on eye level.

In this context the domain expert can act in two different roles: First as the author of the model, using the DSL to write the DSL source text, observing the DSL's syntax rules and semantic definitions. This is the normal case which covers nearly hundred percent of daily practice. In a second step, the domain expert selects evaluating functions from elaborated and well-proven libraries and combines them for processing the model, see e.g. the practice in statistics, using the language "R" [5]. The employed general purpose programming language acts primarily for glueing between input data and library calls. In this way, physicists, sociologists, biologists, etc. can safely act as programmers.

But domains expert can take also a very different role, namely when designing the DSL itself. In such a process they are involved normally as consultants, and the language definition and implementation are done in co-operation of computer scientists, language experts and domain experts.

1.2 Translets for Small, Possibly ad hoc Languages

In this context, our project Translets aims at driving further the emancipatory tendency of DSLs, by enabling domain experts even to define and implement (parts of) a necessary DSL on their own. In its current state this project does not reveal new theoretical results, but is intended as a technological achievement and political manifest.

Of course, larger systems which deal with complex language design problems (modularity, scoping, type systems, etc.) will always require the expertise of computing scientists, as well as elaborated compiler construction infrastructure. But in many situations in practice there is the need for micro languages, which

have a very limited syntax, simple semantics, are used frequently for denotations of data instances of some complex type in a readable fashion. E.g. think of labels encoding the different configurations in a series of experiments for a lab journal; or encoding different settings in a series of interviews in sociology; or the combination of light, aperture, focal length and filter, when encoding films or fotographs, etc. Such a DSL is called *model language* in the following; the need for it often arises spontaneously and it is sensible to have a tool which allows to define and implement such a model language also by the domain experts themselves. So the usage of the tool must be comparable easily teachable, and its implementation must present the generated output data for immediate processing, in a way manageable by naïve programmers. This is what Translets aims at.

The original context of Translets is that of the `tscore` program [3]. This is a framework for defining arbitrary functions indexed by time, where the time model itself, and the format of all parameter values, are generic and hardly limited. The framework controls the top-level syntax and data organisation, triggers the parsing process, constructs the relations between time points, events and parameters, calculates implicit sub-divisions between explicitly notated time instances, etc. It finally delegates the parsing of time points and parameter values to sub-parsers, which must be plugged into the system explicitly. This achieves the intended openness for arbitrary time domains and parameter ranges. In this context Translets have been developed and successfully employed.

Other comparable contexts for their application could be text input fields in a screen mask or structured data embedded in semi-structured flow text, etc. In all cases only small input data must be parsed, and the parsing results must immediately be accessible for further calculations.

According to our experience with students from computer composition and musicology, the meanings and intentions of the standard basic parser combinators (like "star", "question mark" and "alternative") are easily understood by most of them, and we assume a similar potential in all groups of scientists who are no computer experts. But to be useful for them in a practical programming context, severe *restrictions* seem necessary, or at least sensible:

- Only the level of regular expressions is supported, enhanced by simple recursion. More complicated levels of languages (like context-free) are out of scope.
- The chosen parsing process is possibly not optimal w.r.t. resources. You can easily explain the function of basic syntax combinators to domain experts, but you cannot burden them with "first and follow sets", "left-recursion" or "LL(k) property".
- Thus the parsing algorithm must deal with non-determinism in a user-friendly way.
- But resources turn out not to be problematic, since only comparable short character sequences are fed to the parsing process, one by one.
- A high degree of static type safety is desirable,
- and extensive error diagnosis for the unavoidable run-time errors.

– All mere technical aspects of parsing should be kept behind the scenes.
– Last not least: The definition of the types of the generated data and their technical realisation for further processing must not require explicit programming, but automatically be performed when plugging together the parser combinators. The structure of the output data must closely follow the structure of the input syntax, both illustrating each other, and must be well-typed and organized according to the user's need. Esp. the typical detour through an intermediate Abstract Syntax Tree etc. cannot be imposed on the domain expert.

1.3 Translets Technical Realization

These requirements led to the decision to implement Translets as an embedded DSL. We call it the *parser (construction) language* in the following. In contrast to the model language it is a fully embedded DSL, i.e. it is realized as a collection of Java class definitions. Parsers are constructed by expressions (i.e. nested method calls) in the host language Java, and the full scale of its features can be used for their definition and control. An important role in parser construction and type checking plays the type inference algorithm for generic type parameters, which had been introduced with Java 5.0, because it contributes substantially to non-redundant user-friendly notation, see the example in Section 4 below.

The parsing process itself is also initiated and configured by genuine Java means. It transforms some source in the model language, which is an input *text*, i.e. a non-embedded DSL, into a collection of data, i.e. Java object instances, which again can be called an embedded DSL. The user has immediate access to the parsing results, which are realized by instances of precisely parameterized `Map` and `Multimap` types, see Section 2.3 below and the example in Section 4. The structure of this data is intuitively clear, because it closely corresponds to the syntax of the model language. Then, in a second step, as with the "R" example above, processing blocks from domain-specific libraries can be selected, combined and applied for transformation, evaluation and output.

Additionally, the interface structure of the classes involved and the protocol of the parsing process are open and documented, so that more qualified programmers can seamlessly plug in own implementations of parsers and new combinators.

2 Translets Concepts and Operation

2.1 Basic Data Types

Table 1 shows the basic data types and the typing of Translets parser combinators. Let S be the set of all possible character strings, C_0 a finite collection of pre-defined classes of the host language, possibly with some type parameters, and $\mathcal{T}_{\text{prim}}$ its pre-defined simple data types. Let $\texttt{string} \in \mathcal{T}_{\text{prim}}$ and $\texttt{boolean} \in \mathcal{T}_{\text{prim}}$ be the well-known host language built-in data types. Then T gives the possible types of all parsers' results.

$$\mathsf{disjoint}(C_0, \mathcal{T}_{\text{prim}}, P, S, f_{\_,\_}, K, M_{\_,\_}, \overline{M}_{\_,\_})$$

$$T ::= C_0\langle T, \ldots\rangle \mid \mathcal{T}_{\text{prim}} \mid \texttt{List}\langle T\rangle \mid \texttt{CoTuple}\langle T, \ldots, T\rangle$$

$$\begin{aligned} P ::= {} & \texttt{CONST}(S) \mid \texttt{CONST}(S, v) \mid \texttt{CAT}(S, \ldots, S) \mid \texttt{REGEX}(S) \\ \mid {} & \texttt{SEQU}(P, \ldots, P) \mid \texttt{SEQU}(f, P, \ldots, P) \mid \texttt{SEQU}(c, P, \ldots, P) \\ \mid {} & \texttt{PERM}(P, \ldots, P) \mid \texttt{PERM}(f, P, \ldots, P) \mid \texttt{PERM}(c, P, \ldots, P) \\ \mid {} & \texttt{ORn}(P, \ldots, P) \mid \texttt{OR1}(P, \ldots, P) \mid \texttt{PRIORn}(P, \ldots, P) \mid \texttt{PRIOR1}(P, \ldots, P) \\ \mid {} & \texttt{HEAD}(P, P, \ldots) \mid \texttt{FRAME}(P, P, P) \mid \texttt{OPT}(b, v, P) \mid \texttt{STAR}(b, P, P) \mid \texttt{PLUS}(b, P, P) \\ \mid {} & \texttt{STORE}(M_{\_,\_}, P) \mid \texttt{STORE}(\overline{M}_{\_,\_}, P) \end{aligned}$$

$$[\![\,\_\,]\!]^T : P \to T$$

$$[\![\,\texttt{CONST}(s : S)\,]\!]^T = [\![\,\texttt{REGEX}(s : S)\,]\!]^T = [\![\,\texttt{CAT}(s : S, \ldots)\,]\!]^T = \texttt{string}$$

$$[\![\,\texttt{CONST}(s : S, v : t)\,]\!]^T = t$$

$$\frac{\begin{array}{c}[\![\,p_1\,]\!]^T = t_1 \quad \wedge \quad \ldots \quad \wedge [\![\,p_n\,]\!]^T = t_n \\ f : (t_1, \ldots, t_n \to t_R) \qquad c : C_0\langle T, \ldots, T\rangle \qquad b : \texttt{boolean}\end{array}}{\begin{array}{c}[\![\,\texttt{ORn}(p_1, \ldots, p_n)\,]\!]^T = [\![\,\texttt{PRIORn}(p_1, \ldots, p_n)\,]\!]^T = \texttt{CoTuple}(t_1, \ldots, t_n) \\ [\![\,\texttt{OR1}(p_1, \ldots, p_n)\,]\!]^T = [\![\,\texttt{PRIOR1}(p_1, \ldots, p_n)\,]\!]^T = \mathsf{upperLimit}(t_1, \ldots, t_n) \\ [\![\,\texttt{SEQU}(p_1, \ldots, p_n)\,]\!]^T = [\![\,\texttt{PERM}(p_1, \ldots, p_n)\,]\!]^T = \texttt{Object} \\ [\![\,\texttt{SEQU}(f, p_1, \ldots, p_n)\,]\!]^T = [\![\,\texttt{PERM}(f, p_1, \ldots, p_n)\,]\!]^T = t_R \\ [\![\,\texttt{SEQU}(c, p_1, \ldots, p_n)\,]\!]^T = [\![\,\texttt{PERM}(c, p_1, \ldots, p_n)\,]\!]^T = c \\ [\![\,\texttt{HEAD}(p_1, p_2, \ldots)\,]\!]^T = t_1 \quad [\![\,\texttt{FRAME}(p_1, p_2, p_3)\,]\!]^T = t_2 \quad [\![\,\texttt{OPT}(b, v : t_1, p_1)\,]\!]^T = t_1 \\ [\![\,\texttt{STAR}(b, p_1, p_2)\,]\!]^T = [\![\,\texttt{PLUS}(b, p_1, p_2)\,]\!]^T = \texttt{List}(t_1)\end{array}}$$

$$\frac{[\![\,p\,]\!]^T \leq t \quad \wedge \quad m : M_{K,t} \quad \wedge \quad \overline{m} : \overline{M}_{K,t}}{[\![\,\texttt{STORE}(m, p)\,]\!]^T = [\![\,\texttt{STORE}(\overline{m}, p)\,]\!]^T = t}$$

Table 1. Parser Combinators and Types of the Results

The `List` type constructor from the standard Java `collections` framework is used to represent the parsing results of `STAR` and `PLUS`. The only required new data type is `CoTuple`, which corresponds to a "tagged union" in the wording of algebraic data types, and which reifies the outcome of a parser disjunction; for the tags which discriminate the cases we simply use zero-based numbers.[3]

Let further $(t_1, \ldots, t_n \to t_R)$ with $t_R, t_1, \ldots, t_n : T$ be a set of known functions in the host language, with arguments of type $t_1, \ldots, t_n$ and a result of type t_R.

[3] A dedicated data type for tupling is not necessary: it can be achieved either by parallel `STORE` statements with the same key value, or by function/construcor application from a `SEQU` constructor, as described below.

2.2 Parser Combinators and Resulting Types

The Translets parser construction language offers the combinators listed in Table 1. Each derives a new parser $\in P$ from constant values or from other parsers. The expectable type of the result when later applying a parser p to some input is defined by $[\![p]\!]^T$.

The design of the Translets API is not guided by theoretical principles like minimality or compositionality, but by the prospective user's practical needs and abilities. Most of the combinators behave quite conventionally, except for some characteristic variants:

`CONST` and `REGEX` accept the string as described by their argument, and return this string value as a result, unprocessed. The first expects its argument verbatim, the second employs the Java regular expression library. For convenience only, there is a variant of `CONST` which takes a second argument v, a value from an arbitrary type t from the host language, which defines the parsing result. (This host-level typing is indicated informally by "$v : t$"). `CAT` (from "catalog") accepts one of many string constants, in a longest-prefix-match discipline.

The central and heavily overloaded operator for sequential combination is `SEQU`. Its leading position can be taken by a function, followed by (one or more) sub-parsers. This function must have a type signature matching the sequence of the results of the sub-parsers, and the result of the `SEQU`-parser is the result of applying this function.

Instead of a function also a Java `Class` object is allowed, which represents a particular host level class. In this case all its known constructors are considered as functions and their arguments' type signature are compared to the results of the sub-parsers; the first matching constructor is applied. Using a function is strictly statically typed. The constructor variant is very elegant to write, but statically completely unsafe, due to the insufficient Java type system.

If neither of these appears at the head of the arguments then the results are simply discarded, which happens frequently when `STORE` operations are nested inside, see below.

The operator `HEAD` also takes a sequence of sub-parsers and discards all but the first parsing result. This allows to filter input which is syntactically relevant for parsing but shall not be represented in the data output. The operator `FRAME` discards all but the middle parser's result, convenient for the very frequent use of parentheses in the syntax, not represented in the result.

The `PERM` operator behaves like the `SEQU` operator w.r.t. the data output, but allows arbitrary permutation of the sub-parsers on the input side.

The disjunctive combinators come in two times two flavours: `ORn` and `OR1` behave non-deterministically, opposed to `PRIORn` and `PRIOR1`, which cut further alternatives as soon as the first (in parser argument order) matches, as known from the "/" operator in PEGs [2].

`ORn` and `PRIORn` return a co-tuple, combining the types of all alternatives, while `OR1` and `PRIOR1` return always one and the same type, which is the lowest upper limit of these types, as computed by the Java built-in type unification algorithm. The co-tuple variant can also be used in cases where the types of the

alternatives are even identical, but the mere fact *which* of them has been chosen shall be reflected in the data.

The parser combinator for options `OPT` has a default value v, which defines the result in case of non-acceptance, and a boolean flag whether to behave greedy or non-deterministically.

The parser combinators for repetitions, `PLUS` and `STAR`, also have the greedy attribute. They have two sub-parsers: The results of the first one are collected into the resulting sequence, those of the second one are discarded; it is used only on the input side, for delimiters, and may be set to `CONST("")`.

Additionally there is a method `RECURSIVE`(), the result of which can be used as a place holder which can later be overwritten with a reference to the Java object tree in which itself is contained.

2.3 Language Embedding and Execution

The constructors for P as listed in Table 1 are realized as a fully embedded DSL, i.e. as signatures of `static` Java factory methods. These employ heavily the unification algorithm for generic type parameters, which combines ease of writing with a high degree of static type safety. For convenience, most methods are accompanied by several variants which supply default values for omitted parameters.

Figure 1 shows a mid-scale complicated example for a model language definition. Please note that only the first lines (1-10) contain explicit typing information, namely for the containers of the parsing results. Nevertheless, all following construction statements are also completely, strictly and statically typed.[4]

The execution of the parsing process, when applying a $p \in P$ to some character sequence as its input, is realized as brute force breadth first search. Of course this is feasible only due to the limited size of the input texts. Any parse tree is considered a valid solution iff it consumes the input completely. The user must explicitly choose the **reaction on ambiguity**: If there is more than one possible solution, the system may (1) either stop execution with an error message or (2) arbitrarily select one of the results and issue a warning to the user, or (3) do so silently.

After a single result has been selected, the `STORE` operations contained in that parse tree are executed. Each single call of the parsing process must be parameterized with a *key*, an element $k \in K$, which is used to store the results in map or multimap objects. A map $m \in M_{t_1,t_2}$ maps one single element from t_1 to at most one element from t_2, while a multimap $\overline{m} \in \overline{M}_{t_1,t_2}$ realizes a mathematical m-to-n-relation between both types.

In the original application context, the music language `tscore`, the keys $k \in K$ stand for so-called "musical events", objects related to time points, voices and other events, and the different parsing results realize their "musical parameters".

[4] This fact will allow an easy implementation of some text based front-end for Translets, i.e. as a *non*-embedded DSL, e.g. for documents which contain their own syntax description: S imply read lines 11 *pp.* from Figure 1 as a non-Java text.

But of course totally different settings are possible, as long as K is a set of unique keys for retrieving the parsing results.

3 Diagnosis and Didactics

The fundamental considerations w.r.t. the parsing process must take into account that the user of the model language may be a domain expert, not a language expert. The same holds for the designer of the model language, and, furthermore, both may even be the same person. Therefore an "execution error", which is the impossibility to parse a particular input text completely, may have its reason on the definition side, or on the application side, or on both sides.

Furthermore, diagnosis information may also play an important role in cases without any "error" in the narrow sense, for to check if a successful parsing really reflects the intentions of the author(s). Therefore diagnosis has to be implemented as first class resident. On the other hand, explosion of error diagnosis information must be avoided.

Our current implementation can be parametrized to different levels of verbosity: With maximal verbosity, not only errors but also all successes are analysed in detail. In case of an error, all recognized prefices (= partially successful parsing attempts) are presented, anyhow, together with the indication of the legal continuations, which could have followed but are not present in the input. Contrarily, interpretations of the input which cover the whole of a grammar expression but do not consume the input totally are also considered unsuccessful parsing attempts.

The basic idea in all these cases is to give helpful suggestions to the user, at which positions of the input, each corresponding to a particular position in the grammar, some component of the intended input simply has been forgotten to enter, or has been swapped with its neighbour, or is superfluous due to a typo, etc. In all cases the print-out is done by "ASCII art", based on an equidistant output font, and is structured as follows: First appear, possibly, an error message, a location indication of the source text, a description of the context in which the parsing takes place, etc. Then the structure of the parser expression is pretty-printed with indentations and alignments. This is done by the `Format` library of the authors' `metatools` [7].

Every parsing attempt, complete or only a prefix, is now printed by putting the consumed input characters horizontally aligned under the terminal parsers in the print-out of the parser expression. So the user can see at one glance, how the algorithm interpreted the input and assigned it to the terminal parsers like `CONST` or `CAT`.[5] Figure 4 shows a typical srceen shot.[6]

After these successfully parsed parts there may follow either the indication of the non-parsable suffix in the input, and/or the grammar which is expected

[5] The pretty-printing algorithm in the `Format` library has been modified that no two representations of such parsers start in the same text column.

[6] On this basis, further graphical high-lighting is easy to implement, e.g. based on the emacs table mode, etc.

$$Q ::= \texttt{CONST}(S) \mid \texttt{OR}(\mathbb{P}Q) \mid \texttt{SEQU}(Q^+) \mid \texttt{PERM}(Q^+)$$

$$R = S \times Q^? \times R^? \quad A = R \times Q^+$$

$$\begin{aligned}
\mathsf{parse}_T &: Q \times S &&\to \mathbb{P}R \times \mathbb{P}A \\
\mathsf{parse}_m &: Q \times \mathbb{P}R \times \mathbb{P}A &&\to \mathbb{P}R \times \mathbb{P}A \\
\mathsf{parse}_1 &: Q \times R &&\to \mathbb{P}R \times \mathbb{P}A \\
\mathsf{parse}_{P1} &: Q^+ \times R \times \mathbb{P}\mathbb{N} &&\to \mathbb{P}R \times \mathbb{P}A \\
\mathsf{parse}_{P2} &: Q^+ \times \mathbb{P}R \times \mathbb{P}\mathbb{N} &&\to \mathbb{P}R \times \mathbb{P}A
\end{aligned}$$

$$\mathsf{parse}_T(q,\gamma) = \mathsf{parse}_m(q, \{(\gamma, \bot, \bot)\}, \{\})$$

$$\frac{\mathsf{parse}_m(q,\rho,a) = (\rho', a') \quad \mathsf{parse}_1(q,r) = (\rho_1, a_1)}{\mathsf{parse}_m(q, \{r\} \cup \rho, a) = \begin{cases} (\rho', a' \cup \{(r,q)\}) & \textbf{if} \quad \rho_1 = \{\} \wedge a_1 = \{\} \\ (\rho' \cup \rho_1, a' \cup a_1) & \textbf{otherwise} \end{cases}}$$

$$\frac{a' = \{(r,x) \in a \bullet (r, x \blacktriangleleft q)\}}{\mathsf{parse}_m(q, \{\}, a) = (\{\}, a')}$$

$$\frac{q = \texttt{CONST}(c) \quad r = (\gamma, \_, \_)}{\mathsf{parse}_1(q,r) = \begin{cases} (\{(\gamma', q, r)\}, \{\}) & \textbf{if} \quad \gamma = c \frown \gamma' \\ (\{\}, \{\}) & \textbf{otherwise} \end{cases}}$$

$$\frac{\begin{array}{c} q = \texttt{OR}(q_1, \ldots, q_n) \quad 1 \le k \le n \quad (r_k, a_k) = \mathsf{parse}_1(q_k, r) \\ \mathsf{pos} = \{q_k | r_k \neq \{\} \ \vee \ a_k \neq \{\}\} \quad \mathsf{neg} = \{q_k | q_k \notin \mathsf{pos}\} \\ \bar{r} = \bigcup r_k \quad \bar{a} = \bigcup a_k \end{array}}{\mathsf{parse}_1(q,r) = \begin{cases} (\{\}, \{\}) & \textbf{if} \quad \mathsf{pos} = \{\} \\ (\bar{r}, \bar{a} \cup \{(r, \texttt{OR}(\mathsf{neg}))\}) & \textbf{else if} \quad \mathsf{neg} \neq \{\} \\ (\bar{r}, \bar{a}) & \textbf{otherwise} \end{cases}}$$

$$\frac{\begin{array}{c} q = \texttt{SEQU}(q_1, \ldots, q_n) \quad (\rho_1, a_1) = \mathsf{parse}_1(q_1, r) \\ q' = \texttt{SEQU}(q_2, \ldots, q_n) \quad r = (s, \_, \_) \end{array}}{\mathsf{parse}_1(q,r) = \begin{cases} (\{\}, \{\}) & \textbf{if} \quad \rho_1 = \{\} \ \wedge \ \forall a \in a_1 \bullet a = ((s, \_, \_), \_) \\ (\rho_1, a_1) & \textbf{else if} \quad n = 1 \\ \mathsf{parse}_m(q', \rho_1, a_1) & \textbf{otherwise} \end{cases}}$$

$$\mathsf{parse}_1(\texttt{PERM}(q), r) = \mathsf{parse}_{P1}(q, r, \{\})$$

$$\frac{\begin{array}{c} \overline{q} = \langle q_1 \ldots, q_n \rangle \quad q_k \in \overline{q} \\ (\rho_k, a_k) = \mathsf{parse}_1(q_k, r) \quad \mathsf{parse}_1(\bot, \_, \_) = (\{\}, \{\}) \quad r = (s, \_, \_) \\ \varepsilon_k = \{(s', \_, \_) \in \rho_k | s' = s\} \quad \overline{\rho_k} = \rho_k \setminus \varepsilon_k \quad \mathsf{epsDone} = \{y \bullet \varepsilon_y \neq \{\}\} \\ (\rho'_k, a'_k) = \mathsf{parse}_{P2}(\overline{q} \oplus (k \mapsto \bot), \overline{\rho_k}, e \cup \mathsf{epsDone}) \\ (\rho''_k, a''_k) = \begin{cases} \mathsf{parse}_{P1}(\overline{q} \oplus (k \mapsto \bot), e_k, e \cup (\mathsf{epsDone} \cap \{1 \ldots k\})) \\ \quad \textbf{if} \quad k \notin e \ \wedge \ \exists e_k \in \varepsilon_k \\ (\{\}, \{\}) \quad \textbf{otherwise} \end{cases} \end{array}}{\mathsf{parse}_{P1}(\overline{q}, r, e) = \left(\bigcup_{q_d \in \overline{q}} \rho'_d \cup \rho''_d, \bigcup_{q_d \in \overline{q}} a'_d \cup a''_d\right)}$$

$$\frac{\mathsf{parse}_{P1}(\overline{q}, r, e) = (\rho_r, a_r)}{\mathsf{parse}_{P2}(\overline{q}, \rho, e) = \left(\bigcup_{r \in \rho} \rho_r, \bigcup_{r \in \rho} a_r\right)}$$

Table 2. Skeleton of Parsing Process: Find Results and Alternatives

by the parser at this very end of the parsed prefix. While the former is directly accessible, the latter must be synthesized by the parsing algorithm explicitly, to become informative and helpful for the human reader. This implies avoiding combinatorical explosions, esp. in connection with permutations and epsilon productions. But the ambiguities caused by this combination is even a problem w.r.t. "hard semantics": They are "unavoidable artifacts" and thus in a fundamentally different category than those the user wants to be warned about, as described above. Filtering them out is a major achievement of the Translets software and integral part of the parsing process. Table 2 shows it, reduced to the few basic constructors which contribute to diagnosis information; all other parser constructors behave as usual and are treated transparently.

Basic data type are the tuples from R which represent parsing contexts or stack frames of the parsing process. Every frame (s, q, r) shows the situation of the input s after the parser q has successully been applied; r is the preceding situation. The data type $X^?$ means an optional type, like "`Maybe`" in Haskell; the "`Nothing`" case is needed only for the initial state.

An alternative is represented by a tuple from A which indicates the last successful parsing step and the grammar expression which is expected to follow, but could not be found in the subsequent input. The collected grammar expressions are modelled by the type Q^+, which stands for non-empty lists of Q. (When finally delivered and longer than one, this list can be wrapped into one `SEQU` constructor to get a valid parser language expression.).

The parsing method parse_T is the top-level parsing interface function; it returns zero to many of the context tuples from R, which represent mappings of the complete parser expression to the input, including those which leave an unconsumed suffix, and zero to many alternatives from A, collected on the way.

parse_1 is the auxiliary function which parses a grammar expression starting with one(1) context; only this function must be defined for the different parser types. parse_m simply maps this function over more than one start contexts.

The execution parse_1 of a `CONST` parser is the point of decision: if it accepts, simply the corresponding prefix of the input is consumed; if not, an empty result set is returned and an alternative object constructed by the calling level. The "syntactic sugar" parsers `CAT` and `REGEX` are left out in Table 2 and act accordingly.

The function parse_m maps over all incoming parsing states. Afterwards it appends the current parser to the tail of all incoming alternatives; so in parallel to the normal, productive parsing, all alternatives also grow, by appending all sub-parsers which follow their point of failure either on the same nesting level, or on higher nesting levels after unrolling the function stack. When returning, these older alternatives are joined with those collected anew when further descending, for further growth.

3.1 Condensing Diagnosis Information

The only sources of alternatives are those branches of `OR` expressions which cannot be parsed. (All other choice points, like `OPT` or `STAR`, do not contribute

to the ergonomic problem discussed here.) To be easy readable, the collected representations of alternatives shall be condensed as far as sensible: All branches of an `OR` expression which neither contribute to alternatives nor to any result will be wrapped into on single synthesized `OR` expression to start a new alternative. Similar with `SEQU`: If the first alternative in a sequence brings no results, and all alternatives do not consume any input characters, then the whole sequence will be presented as one single, common alternative, discarding all possible epsilon consumptions.

The idea is, that any explicit presentation of epsilon consumption to the user does not help with the suggestions of possible error causes, as listed above, but would contrarily only be confusing.

This is esp. critical with the permutation opertor `PERM`, which is special anyhow. It does not fit into the common theory of regular expressions. It is not associative, but an n-ary version is required: "`a&(b&c)`", "`(a&b)&c`" and "`a&b&c`" behave differently in most designs. Nevertheless it is commonly needed in practice: In case of syntactically distinguishable sub-elements the user must be disburdened to learn an additinal rule for sequential order, which has nothing to do with the intended semantics. For example, in the music sheet language lilypond [4] the additional attributes for a note event like "`,`", "`(`", "`?`", "`-\sfz`", etc. are lexically totally disjoint and semantically unrelated; nevertheless complicated rules for their sequential order must be obeyed. This is ergonomically a severe problem, and thus the permutation operator is undispensable in practice.

Together with epsilon valued sub-expressions, the permutation operator will cause combinatorical explosions of error message generation when treated naïvely: If an expansion to the empty word is possible for a particular sub-expression, this is (in most cases) possible at many different positions and with many different permutations of the sibling branches, without adding any sensible diagnosis information. Therefore all solutions matching the empty string are filtered out and treated specially: only the "earliest" survives, as pars pro toto, when generating solutions as well as alternatives. (Similar means are taken when epsilon expansions appear under a `STAR` operator.)

Due to the greedy attribute, the epsilon extensions cannot be collected statically by transforming the parser expression, but must be detected dynamically: the greedy parser expression "`a?`" does *not* match the empty string when applied to "`a`". Again, ergonomical considerations rule our design: The greedy operators are required by practical needs. The combination of `OPT` and `PERM` is rather frequent in practice, see the above mentioned lilypond note attributes which are all optional. Consider the expression

```
PERM ( A?, B?, C?, D? )
```

If all possible permutations were tested, a simple empty input would yield 24 variants of parsing. Therefore the algorithm operates as follows: The function parse_{P1} makes one step in a breadth-first discipline. It tests every sub-expressions once. The resulting frames are classified into consuming $= \overline{\rho_k}$ and non-consuming, ie. having accepted only epsilon, $= \varepsilon_k$. The remaining expres-

sions $\overline{q} \oplus (k \mapsto \bot)$ are parsed with all non-epsilon states into (ρ'_k, a'_k) and with all epsilon states into (ρ''_k, a''_k) and the results put together.

The third parameter to parse_{P1} is the set of indexes of all sub-expressions which have already appeared in an epsilon variant in some earlier instantiation of the function. The indexes which accepted an epsilon in this particular step are collected in epsDone and added to this set when the subsequent parser steps are performed by calling parse_{P2} recursively. (This function simply maps over a set of parsing state and joins the results.)

Only when a particular branch q_k is not explicitly blocked by this set parameter, one of its epsilon frames (e_k, randomly selected) is included in the further parsing process.[7]

In this case, when the subsequent components are parsed (by calling parse_{P1} recursively), only the lower indexed sibling branches are excluded from accepting epsilon (by $\mathsf{epsDone} \cap \{1 \ldots k\}$): This allows maximally only one case in which the epsilon consumptions of different parsers are combined.

The following scheme illustrates this principle: each vertical line | stand for an instantiation of parse_{P2} followed by the results of the calling of the subparsers parse_1; the recursive calls follow from left to right, with shrinking sets of branches, see $\mathsf{parse}_{P2}(\overline{q} \oplus (k \mapsto \bot), \ldots)$. Upper case characters stand for matches of the subexpressions of the expression from above; all are non-empty, except those annoted with =<>. After having matched (a non-empty) `A`, there are two matches for `C`, one of these is empty, and one empty match for `D`. Then in the third and fourth incarnation of parse_{P2} neither of these may be mapped to epsilon, indicated symbolically by ! =<>. The only exception is the position $***$, the only place where the combination of an empty `C` and an empty `D` would be accepted. The epsilon expansions in parallel calls to parse_m have no effect, see `A` =<>:

```
|A    |B    |C!=<>   D!=<>
|     |     |D!=<>   C!=<>
|     |
|     |C    |B       D!=<>
|     |     |D!=<>   B
|     |
|     |C=<> |B       D ***
|     |
|     |D=<> |B       C!=<>
|     |     |C!=<>   B
|
|B    |A=<> |C       D
|     |     |D       C
|     |
|     |C    |A!=<>   D
|     |     |D       A!=<>
|     |
|     |D    |A!=<>   C
|     |     |C       A!=<>
|
|C    |A    |B!=<>   D
|     |     |D       B!=<>
|     |B=<>
| (etc.)
```

This simple algorithm miminzes the set of solutions also for pathological cases, which happen to appear in languages designed by laymen. Consider the

term from above with `A=a?, B=b?, C = (a|b)?`, i.e. `PERM (a?, b?, (a|b)?, D)`, and input is "`abD`".

There are six permutations of three expressions. With non-greedy `OPT` combinators we get nine different mappings. From these, our algorithm eliminates six due to repeated epsilon assignment:

a	**b**	**(a|b)**	
a	b	-	$\Longrightarrow$ pass
a	-	b	$\Longrightarrow$ pass
a	**(a|b)**	**b**	
a	b	-	
a	-	b	
-	a	b	$\Longrightarrow$ pass
b	**a**	**(a|b)**	
-	a	b	
b	**(a|b)**	**a**	
(a|b)	**a**	**b**	
a	-	b	
-	a	b	
(a|b)	**b**	**a**	
a	b	-	

With all greedy `OPT` operators we get three mappings instead of nine:

```
|a          |b      |(a|b)=<> | D
|           |(a|b)  |b!=<> -->no match
|
|b=<>       |a      |(a|b)     | D
|           |(a|b)  |a   -->no match
|
|(a|b)      |a=<>   |b!=<>     | D
|           |b      |a!=<>   -->no match
```

In case of non-greedy = non-deterministic `OPT` operators, all epsilon variants can appear in the first position and thus are blocked in all subsequent calls to parse_{P1} by the set parameter e. So one of them must appear in the first call:

```
|a=<>       |(a|b)!=<> |b!=<>       | D
|
|b=<>       |a!=<>      |(a|b)!=<>   | D
|
|(a|b)=<>   |a!=<>      |b!=<>       | D
```

[7] This state which serves as a start state is selected randomly. If it were required that it contains the complete information of epsilon extensions, all grammar expressions of all states from ε_k must be collected into a new, synthesized "`OR`" expression. In the current implementation this not necessary only due to the special way in which parsing information is finally presented, see the description above and Figure 4, namely guided by the original top-level parser expressions and all non-empty consumed terminal symbols.

```
Map<Event,ThForm> thForm = new HashMap<>(),
Map<Event,String>
  incompl = new HashMap<>(),
  inverse = new HashMap<>(),
  retrogr = new HashMap<>();
Map<Event,Integer> cpNumber = new HashMap<>();
Map<Event,Rational> factor = new HashMap<>();

public static class ThForm extends Tuple<String,Integer>{
  public ThForm (final String s, final Integer i){ super(s,i);}}

Parser p_theme
  = STORE (thForm(OR1(SEQU(ThForm.class,
                           SEQU(roman2integer, CAT("I", "II", "III", "IV")),
                           OPT(true, "", CAT("D", "C"))),
                      SEQU(thForm.class, CONST("", 1), CAT("D", "C"))
                ));

Parser p_kp = STORE(cpNumber, SEQU(string_to_integer, REGEX("[1-9]")));

Parser p_mul = SEQU(CONST("*"),
                    STORE(factor, SEQU(string_to_rational, REGEX("[2-9]"))));
Parser p_div = SEQU(CONST("/"),
                    STORE(factor, SEQU(rational_invert,
                                       SEQU (string_to_rational,
                                             REGEX("[2-9]")))));
Parser p_divMul = OR1(p_mul, p_div) ;

Parser p_incompl = STORE (incompl, CONST(">"));
Parser p_inverse = STORE (inverse, CONST("u"));
Parser p_retrogr = STORE (retrogr, CONST("r"));

Parser p_freeMaterial = CONST("X");

Parser all = OR1 ( SEQU (p_theme, PERM(OPT(p_incompl), OPT(p_inverse),
                                       OPT(p_retrogr), OPT(p_divMul))),
                   SEQU (p_kp, PERM(OPT(p_incompl), OPT(p_inverse) )),
                   p_freeMaterial,
                   CONST("%") );
```

Fig. 1. The Fugue-Form-Plan Language

```
PARS KdF_VII
T          1      2     3    4    5     6     7    8    9    10    11   12   13
VOX sop            Du                      X         3 X
VOX alt                  Cu/2      X           C/2       X
VOX ten    D/2           1       X       D/2>+ X    3 X  D/2u       X
VOX bas                            C*2u                                    %
                                 // C*2uX yields error, see text
```

Fig. 2. The Beginning of *Contrapunctus Vii* From *Die Kunst Der Fuge* By J.S.Bach.

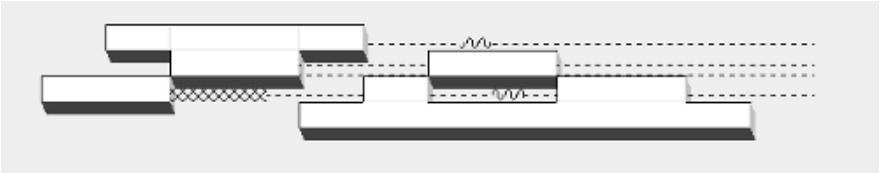


Fig. 3. A Simple Visualization Derived From the Source in Figure 2

```
kdf_vii.tscore:6:34 No parsing result for "C*2uX"
The grammar rule for this input is
 Parser all = OR1(SEQU(OR1(SEQU(CAT("I", "II", "III", "IV")), CAT("D", "C")?),
                           SEQU( "", CAT("D", "C"))),
                       PERM(((">")?, ( "u")?, ("r")?,
                            OR1(SEQU(  "/",REGEX("[2-9]")),
                            SEQU( "*",  REGEX("[2-9]"))),
                            )
                      ),
                  SEQU(REGEX( "[1-9]"), PERM((  ">")?, ("u")?, ("r")?)),
                  "X",
                   "%"
                 )
After having parsed ...
                                                                  "C"
                                      "*"   "2"
                                           "u"
... there could follow input according to ...
 PERM( ">"?, "r"?)
There are unparsable rest characters
 "X"  at position 5
```

Fig. 4. Messaging Erroneous Input With Alternatives

4 Practical Example

As mentioned above, `tscore` is a framework for writing down, representing and processing sequences of arbitrary values, in a *diachronic* notation context.

The following example of a Translets parser is already at the upper end of the supported and required complexity. It is a typical application from *musicology*, namely from *music theory*.

A format shall be defined which allows to describe the *formal disposition of a fugue*, as composed by Bach and others. This form plan data can then be subject of further processing, e.g. statistical analysis, or visualization for educational purposes, or even as a basis for new synthesis.

Without going into the details of the domain, the parser constructed in Fig. 1 can roughly be described as follows:

- In the parser `p_theme` the letters `D` and `C` stand for an appearance of the first, main theme of the fugue in its *dux* form (i.e. the first appearance) or in its *comes* form (second appearance).
  `II`, `III`, `IV` stand for further themes, and the form indication letter `D`/`C` can be appended or omitted. (In practice it is not always sensible to apply this distinction to the themes $> I$). The resulting parser is a typical example for non-determinism, which the author should not be bothered to deal with explicitly.
- Contrarily, the parser `p_kp` stands for some particular contrapuntal material, which (on one side) re-appears frequently and can be perceived as identical, but (on the other side) is not stable enough to be called a theme on its own. (This is called “fixed” or “sticky counterpoint”).

- `p_divMul` and its sub-parsers stand for augmentation and diminution (i.e. slowing down and accelerating of the duration values) with the indicated factor.
- Additionally any appearance of any theme and counterpoint may be *incomplete* i.e. prematurely ended, *inverse*, i.e. played "upside down", and *retrograde*, i.e. played backwards in time. This is realized by the parsers `p_incompl`, `p_inverse` and `p_retrogr`.
- The parser `p_all` combines
  - the parser for theme identification with all modifiers, in arbitrary order;
  - the parser for fixed counterpoint material with most, but not all modifiers (if you *do* need them all, you must promote the fixed counterpoint to a theme);
  - the parsers indicating (the start of) free material and pauses.

This combined parser is applied to the different voices in a `tscore` document. The "events" stand for the appearances of whole themes and counterpoints, the time axis is defined to measure numbers, and the voices of the `tscore` source are the same as the voices in the composition to be described. Please note that the text in Fig. 1 must be completed only by the corresponding **`import`** statements and by the calls which initiate the parsing of the time line and the voice lines. Nothing else must be explicitly programmed to get the event data into the correctly typed `Map` objects, ready for further processing.

Figure 2 shows the beginning of *Contrapunctus VII* from *Die Kunst der Fuge* by J.S.Bach.

The few lines from Fig. 1 are all the user has to define for this rather advanced purpose. All preparation and scheduling is carried out by the `tscore` framework. After this, the maps defined in the first lines of Fig. 1 are filled accordingly, where each `Event` object represents one of the thematic appearances. The `tscore` framework offers control structures to step through the collection of all musical events in different orders. Using these, and the events as keys into the result maps, all data can be passed to graphic libraries, yielding a visualization as shown in Fig. 3.

5 Related Work

The combination of epsilon productions and permutation operator in parsing has been discussed before in [1], in the context of Hakell parser combinator libraries. Basically the authors have found the same strategy as ours, namely to factor out the epsilon case and treat it separately, after all possible permutations of the non-epsilon base parsers have been tested. The differences to our approach are, that they can figure out and transform the regular expression statically, in advance, while we support arbitrary black box parsers, plugged in as Java classes, and greedy variants of the `?` and `*` combinators. Thus in our approach the transformation must be executed at parse time. Diagnosis is not in their focus (the ASCII art the presentation scheme from our Figure 4 seems original)

and non-determinism is not discussed, neither the difference between wanted and unwanted non-determinism. Last not least, they "utilise lazy evaluation to make the resulting implementation efficient", which implies programming tactics in the small different to ours in the eager Java language.

References

1. Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Parsing permutation phrases. *J. Funct. Program.*, 14(6):635–646, November 2004.
2. Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages*, pages 111–122. ACM Press, 2004.
3. Markus Lepper and Baltasar Trancón y Widemann. tscore: Makes computers and humans talk about time. In Joaquim Filipe and Jan Dietz, editors, *Proc. KEOD 2013, 5th Intl. Conf. on Knowledge Engineering and Ontology Development*, pages 176–183, Portugal, 2013. instincc, scitePress.
4. *Lilypond Music Notation*, 2011. `http://lilypond.org`.
5. R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.
6. J. M. Spivey. *The Z Notation: a reference manual*. International Series in Computer Science. Prentice Hall, 1988.
7. Baltasar Trancón y Widemann and Markus Lepper. *The BandM Meta-Tools User Documentation*. `http://bandm.eu/metatools/docs/usage/index.html`, 2010.

A Mathematical Notation

The employed mathematical notation is fairly standard, inspired by the Z notation [6]. But for leaner notation, we add some overloading. The following table lists some details:

$\mathbb{P}\ A$	Power set, the type of all subsets of the set A.
$A \times B$	The product type of two sets A and B, i.e. all pairs $\{c = (a, b) \vert a \in A \wedge b \in B\}$.
$A \to B$	The type of the *total* functions from A to B.
A^+	All possible finite, non-empty sequences from elements of A.
$r \oplus s$	Overriding of function or relation r by s.
$\langle\rangle$	The empty sequence.
$\alpha \blacktriangleleft b$	A list(/stack) with element b preceded by a prefix α.
$\alpha \frown \beta$	Concatenation of two lists.

Type unification for structural types in Java

Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract

In the past we considered type inference for Java with generics and lambda-expressions. The base of our algorithm was a finitary type unification. The algorithm determines nominal types in subjection to a given environment. This is a hard restriction as separate compilation of Java classes without relying on type informations of other classes is impossible. In this paper we present an extended type unification algorithm as the base of a type inference algorithm for a Java-like language, that infers structural types without given environments.

1 Introduction

We considered in different contributions in the past type inference in Java-like languages e.g. [Plü15]. The resulting types are nominal types, that are dependent from a given environment. Let us consider the following example:

```
import java.util.Vector;
class A { m (v) { return v.elementAt(0); } }
```

For the method `m` the type `Vector<A>` $\rightarrow$ `A` is inferred, as `Vector` is the only class in the environment. This type is not principal. The principal type of `m` would be a structural type `ST<A>`, that have a method `elementAt`: `SA<A>` $\rightarrow$ `A`. We gave a sketch of a type inference algorithm for structural types in [Plü16]. Our algorithm is a generalization and an improvement of an idea, that has given in [ADDZ05]. We replace a complex linking algorithm by type unification. As in [ADDZ05] the algorithm is given for a Featherweight Java.
Basically, we reduce the Java type inference for nominal types to type unification, as in [DM82] the type inference for ML was reduced to ordinary unification [Rob65]. While we gave in [Plü04] type unification for G–JAVA (Java with generics but without wildcards), in [Plü09] we presented type unification for Java with wildcards. Both unifications problems are in general not unitary but finitary. This results from the property that the soltutions of a constraint $T \lessdot ty'$, where T is type variable, ty' is a Java type and $\lessdot$ means *should be a subtype*, are all substitutions $\{ T \mapsto ty \mid ty$ is a subtypes of $ty' \}$. The corresponding type unification algorithms are advancements of Martelli and Montanari's [MM82].
In a similar way we will reduce Java type inference for structural types to a changed type unification, where $T \lessdot ty'$ (resp. $ty \lessdot T$) itself is a result, that is not resolved.
In the following we give some basic definitions. Then we present the type unification algorithm and give an example. Finally, we close with a summary.

2 Basic definitions

The parametrized Java classes and interfaces form a rank alphabet $\Theta = (\Theta^{(n)})_{n \in \mathbb{N}}$ of its names, where n represents the number of class-parameters. In the follwoing TV is a set of type variables.

Definition 1 (Set of type terms)**.** *Let* $\Theta = (\Theta^{(n)})_{n\in N}$ *be the rank alphabet of classes and interfaces. The* set of type terms $T_\Theta(TV)$ *is given as the set of terms over the rank alphabet* Θ *and the set of type variables* TV*.*

The extends/implements relation induces the type term ordering.

Definition 2 (Type term ordering)**.** *Let* $T_\Theta(TV)$ *be a set of type terms and* $<$ *the* extends/implements *relation on* $T_\Theta(TV)$*. The* Type term ordering $\leq^*$ *is given as the smallest ordering with the conditions:*

- *if* $(\theta, \theta') \in T_\Theta(TV) \times T_\Theta(TV)$ *is an element of the reflexive and transitive closure of* $<$ *then* $\theta \leq^* \theta'$*.*
- *if* $\theta_1 \leq^* \theta_2$ *then* $\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)$ *for all substitutions* σ_1*,* σ_2*, which satisfy the following condition:* $\sigma_1(a) = \sigma_2(a)$ *for all* $a \in \mathsf{TVar}(\theta_2)$*.*

The type unification problem is given in the following definition.

Definition 3 (Type unification problem, type unifier)**.** *The* type unification problem *is given as: For a set of constraints* $\{\theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n\}$*, where* $\theta_i, \theta'_j \in T_\Theta(TV)$*, a substitution* σ *is demanded, such that for all* $1 \leqslant i \leqslant n : \sigma(\theta_i) \leq^* \sigma(\theta'_i)$*. The substitution* σ *is called* type unifier*.*

During the unification algorithm $\lessdot$ is replaced by $\doteq$, where $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.

Definition 4 (Most general type unifier)**.** *A given type unifier* σ *of* C *is called* most general type unifier *if for any type unifier* σ' *of* C *there is a substitution* σ'' *such that* $\sigma' = \sigma'' \circ \sigma$*.*

If the type unifications algorithm does not fail the result is given in solved form. The following definition of *solved form* is an extension of the definition for the ordinary unification.

Definition 5 (Solved form)**.** *A set of contraints* C *is in* solved form*, if all elements of* C *has either the form* $T \doteq \theta$*,* $T \lessdot \theta$*, or* $\theta \lessdot T$*, where* T *is a type variable,* $\theta \in T_\Theta(TV)$*, and* $T \notin \mathsf{TVar}(\theta)$ *holds.*

3 The type unification algorithm

The algorithm **TUnify**(C) is given by the rules (Figure 1) application the most often as possible. If C is finally in solved form then C is the result, otherwise the algorithm fails.

Lemma 6 (Termination)**.** The algorithm **TUnify** terminates.

Lemma 7 (Soundness of TUnify)**.** If a substitution σ is a solution of a constraint set C then σ is also a solution of **TUnify**(C).

Lemma 8 (Completeness, most general unifier of TUnify)**.** Let for a set of constraints C the substitution $\sigma = \{\mathtt{T} \mapsto ty \mid \mathtt{T} \doteq ty \in \mathbf{TUnify}(C)\}$ be given. For any solution σ' of C there are substitutions σ'' and σ_{rest}, such that $\sigma' = \sigma'' \circ ((\sigma_{rest} \circ \sigma) \cup \sigma_{rest})$. The substitution $(\sigma_{rest} \circ \sigma) \cup \sigma_{rest}$ is a most general unifier in dependency on σ_{rest}.

Remark The substitution σ_{rest} is a solution of the remaining pairs $\mathtt{T} \lessdot ty$ and $ty \lessdot \mathtt{T}$

(reduce) $$\frac{C \cup \{\, C\langle\theta_1,\ldots,\theta_n\rangle \lessdot D\langle\theta'_1,\ldots,\theta'_n\rangle \,\}}{C \cup \{\, \theta_1 \doteq \theta'_1,\ldots,\theta_n \doteq \theta'_n \,\}}$$
where $C\langle T_1,\ldots,T_n\rangle \leq^* D\langle T_1,\ldots,T_n\rangle$ with T_i are type variables

(adapt1) $$\frac{C \cup \{\, D\langle\theta_1,\ldots,\theta_n\rangle \lessdot D'\langle\theta'_1,\ldots,\theta'_m\rangle \,\}}{C \cup \{\, D'\langle\tilde{\theta}'_1,\ldots,\tilde{\theta}'_m\rangle[T_i \mapsto \theta_i \mid 1\leqslant i\leqslant n] \doteq D'\langle\theta'_1,\ldots,\theta'_m\rangle \,\}}$$
where $(D\langle T_1,\ldots,T_n\rangle \leq^* D'\langle\tilde{\theta}'_1,\ldots,\tilde{\theta}'_m\rangle)$ with T_i are type variables

(adapt2) $$\frac{C \cup \{\, D\langle\theta_1,\ldots,\theta_n\rangle \lessdot S_1, S_1 \lessdot S_2,\ldots,S_{k-1} \lessdot S_k, S_k \lessdot D'\langle\theta'_1,\ldots,\theta'_m\rangle \,\}}{C \cup \{\, \sigma(\, D\langle\theta_1,\ldots,\theta_n\rangle\,) \lessdot S_1, S_1 \lessdot S_2,\ldots,S_{k-1} \lessdot S_k, S_k \lessdot \sigma(\, D'\langle\theta'_1,\ldots,\theta'_m\rangle\,) \,\} \cup \sigma}$$
where
- $k \geq 1$
- $S_i \in TV$ and
- $(D\langle\theta_1,\ldots,\theta_n\rangle \not\leq^* D'\langle\theta'_1,\ldots,\theta'_m\rangle)$ but $(D\langle T_1,\ldots,T_n\rangle \leq^* D'\langle\tilde{\theta}'_1,\ldots,\tilde{\theta}'_m\rangle)$ with $T_i \in TV$
- $\sigma = \mathbf{Unify}[1](\, \{\, D'\langle\tilde{\theta}'_1,\ldots,\tilde{\theta}'_m\rangle[T_i \mapsto \theta_i \mid 1\leqslant i\leqslant n] \doteq D'\langle\theta'_1,\ldots,\theta'_m\rangle \,\}\,)$

(erase1) $$\frac{C \cup \{\, \theta \lessdot \theta' \,\}}{C} \quad \theta \leq^* \theta'$$

(erase2) $$\frac{C \cup \{\, \theta \doteq \theta' \,\}}{C} \quad \theta = \theta'$$

(swap) $$\frac{C \cup \{\, \theta \doteq T \,\}}{C \cup \{\, T \doteq \theta \,\}} \quad \theta \notin TV,\ T \in TV$$

(subst) $$\frac{C \cup \{\, T \doteq \theta \,\}}{C[T \mapsto \theta] \cup \{\, T \doteq \theta \,\}} \quad T \in TV \text{ and } T \text{ occurs in } C \text{ but not in } \theta$$

(refl) $$\frac{C \cup \{\, \theta \lessdot T_1, T_1 \lessdot T_2,\ldots,T_{n-1} \lessdot T_n, T_n \lessdot \theta \,\}}{C \cup \{\, T_i = \theta \mid 1\leqslant i\leqslant n \,\}} \quad \theta \notin TV,\ T_i \in TV$$

Figure 1: Type unification

3.1 Type inference algorithm

The *type inference algorithm* is given by the three follwoing functions **TYPE**, **construct**, and **solve**:

- **TYPE** collects the type constraints.
- **construct** builds the interfaces, that represent the structural types.
- **solve** unifies the constraints by the type unification algorithm.

[1] **Unify** is ordinary unification given in [Rob65] or more efficient in [MM82]

4 Example

Let the following typeless Java class be given

```
class A { mt(x, y, z) { return x.sub(y).add(z); } }
```

The result of the type inference algorithm is

```
interface Sub<R, T> { R sub(T x); }
interface Add<R, T> { R add(T x); }
class A <ν1,ν3,ν4,ν6>
        [ν3 extends ν5, ν4 extends ν7, ν1 extends Sub<ν2,ν5>, ν2 extends Add<ν6,ν7>]{
    ν6 mt(ν1 x, ν3 y, ν4 z) { return x.sub(y).add(z); } }
```
[2]

In this case the type unification function **solve** changes nothing as the result constraints set of **construct** $\{\, \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \texttt{Sub<}\nu_2,\nu_5\texttt{>}, \nu_2 \lessdot \texttt{Add<}\nu_6,\nu_7\texttt{>} \,\}$ is already in solved form. These constraints are given as bounds in the output syntax of `A`. The four class-parameters are the argument- and the return-types of the method `mt`. The interfaces `Add` and `Sub` represents the structural types.

Now we give a class `myInteger`, that implements `Sub` as well as `Add`.

```
class myInteger extends Sub<myInteger, myInteger>, Add<myInteger, myInteger> {
    Integer i;
    myInteger sub(myInteger x) { return new myInteger(i - x.i); }
    myInteger add(myInteger x) { return new myInteger(i + x.i); } }
```

Finally, in the class `Main` an instance of `A` is used and the method `mt` is called.

```
class Main {
    main() { return new A<>()
                    .mt(new myInteger(2), new myInteger(1), new myInteger(3)); } }
```

The result of **TYPE** and **construct** is the constraint set, that has to be unified:

$$C_{\texttt{main}} = \{\; \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \texttt{Sub<}\nu_2,\nu_5\texttt{>}, \nu_2 \lessdot \texttt{Add<}\nu_6,\nu_7\texttt{>}, \quad . \\ \texttt{myInteger} \lessdot \nu_1, \texttt{myInteger} \lessdot \nu_3, \texttt{myInteger} \lessdot \nu_4 \,\}$$

The class declarations implies the subtyping ordering $\texttt{myInteger} \leq^* \texttt{Sub<myInteger,myInteger>}$ and $\texttt{myInteger} \leq^* \texttt{Add<myInteger,myInteger>}$.

The type unification algorithm applied to $C_{\texttt{main}}$ is given as follows: With the *adapt2*-rule follows from $\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<}\nu_2,\nu_5\texttt{>}$: $\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<myInteger,myInteger>}$, $\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}$. From this follows with the *subst*-rule $\texttt{myInteger} \lessdot \texttt{Add<}\nu_6,\nu_7\texttt{>}$ and with the *adapt1*-rule: $\texttt{Add<myInteger,myInteger>} \doteq \texttt{Add<}\nu_6,\nu_7\texttt{>}$. With the *reduce*- and *swap*-rule we get: $\nu_6 \doteq \texttt{myInteger}, \nu_7 \doteq \texttt{myInteger}$. With the *subst*-rule follows $\texttt{myInteger} \lessdot \nu_3$, $\nu_3 \lessdot \texttt{myInteger}$ and $\texttt{myInteger} \lessdot \nu_4, \nu_4 \lessdot \texttt{myInteger}$ and from this with the *refl*-rule: $\nu_3 \doteq \texttt{myInteger}$ and $\nu_4 \doteq \texttt{myInteger}$.

[2] The output syntax of the type inference algorithm differs from standard Java. On the one hand the bounds of the class-parameters are given in a additional declaration in `[ ]`-bracket. On the other hand type variables, that are only necessary to describe the bounds and not used in the class fields or methods need not to be declared explicitly as class-parameters.

The result of **solve** is given as:

$$\{\, \texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \texttt{Sub<myInteger,myInteger>} \\ \nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}, \nu_6 \doteq \texttt{myInteger}, \\ \nu_7 \doteq \texttt{myInteger}, \nu_3 \doteq \texttt{myInteger}, \nu_4 \doteq \texttt{myInteger} \,\}$$

The resulting Java class is given as:

```
class Main [ myInteger extends ν1, ν1 extends Sub<myInteger, myInteger> ] {
    myInteger main() {
        return new A<>().mt(new myInteger(2), new myInteger(1), new myInteger(3)); } }
```

There is one remaining type variable ν_1, that is not used in a argument- or return-type of a method. Therefore ν_1 is no class-parameter of `Main`. The two remaining bounds of ν_1 are consistent. This means `main` is executable and the result of execution is 4.

5 Summary

We have presented a type unification algorithm as the base of a type inference algorithm for a Java-like language, that infers structural types. The presented type unification is an extension of the type unification for nominal types. While in the nominal case all solutions of a constraint $a \lessdot \theta$ are determined, in the structural case the constraint $a \lessdot \theta$ itself is a result, that is resolved not until an instance of the corresponding class is created. This approach has two main advantages: It reduces the number of unification solutions enormously and allows separate compilation of Java classes without relying on type information of other classes.

References

[ADDZ05] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 26–37, New York, NY, USA, 2005. ACM.

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

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

[Plü04] Martin Plümicke. Type Unification in Generic–Java. In Michael Kohlhase, editor, *Proceedings of 18th International Workshop on Unification (UNIF'04)*, Cork, July 2004.

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

[Plü15] Martin Plümicke. More type inference in Java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

[Plü16] Martin Plümicke. Structural type inference in java-like languages. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016.*, pages 109–113, 2016.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, January 1965.

Erweiterung und Neuimplementierung der Java Typunifikation

Florian Steurer und Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
florian.steurer95@gmail.com, pl@dhbw.de

Zusammenfassung. Java is a programming language where it is necessary to declare types explicitly. With type inference, type declarations can be omitted, making programs shorter and easier to write. We gave a type inference algorithm for Java. This algorithm uses a constraint-based type unification, which, for a given set of type constraints, computes all possible typings of a program. The type unification algorithm has been implemented as part of a compiler, which supports implicitly typed Java programs.
In this paper the Java type unification is extended by real function types. Additionally, some cases concerning wildcard types are added, that were not considered before. This leads to a simplification of the algorithm.
The old implementation is too inefficient to handle inputs of relevant size. A complete reimplementation aims to improve efficiency by using immutable data structures and parallelism.

1 Einführung

In Java müssen bisher alle Typen explizit deklariert werden. Mit Typinferenz ist es möglich Typdeklarationen auszulassen, so dass die Typen automatisch inferiert werden. Durch inferierte Typen werden Programme kürzer und einfacher zu schreiben. Ein Typinferenzalgorithmus für Java wurde in [Plü15b] vorgestellt. Der Typinferenzalgorithmus benutzt einen constraint-basierten Typunifikationalgorithmus [Plü07], welcher, für eine Menge an Constraints alle möglichen Typisierungen eines Java-Programms berechnet.

In [LP07] wurde dieser Algorithmus implementiert. Die Implementierung wird in einem Compiler [Sta15] genutzt, welcher implizit typisierte Java-Programme

unterstützt, indem er Typen inferiert. Die Implementierung des Typunifikationsalgorithmus ist allerdings zu ineffizient um Eingaben von relevanter Größe unifizieren zu können. Das hat zur Folge, dass nur kleine Java-Programme in einer angemessenen Zeit kompiliert werden können.

Durch eine Neuimplementierung des Typunifikationsalgorithmus soll die Effizienz des Algorithmus verbessert werden, so dass Typen auch in größeren Java-Programmen inferiert werden können. Der Algorithmus soll um die in [Plü15a] beschriebenen echten Funktionstypen für Java erweitert werden. Dabei soll auch auf eine softwaretechnisch gute Implementierung geachtet werden, die erweiterbar, anpassbar und verständlich ist. Im Rahmen dieser Arbeit wird der Algorithmus außerdem um bisher nicht betrachtete Fälle erweitert, aufgrund derer bestimmte Lösungen bisher nicht gefunden wurden.

2 Typunifikationsalgorithmus

In diesem Kapitel wird die Typunifikation auf einer formalen Ebene beschrieben. Zu Beginn folgt eine allgemeine Beschreibung des Algorithmus und der Typunifikation.

Im Anschluss werden in Abschnitt 2.2 ergänzend zu [Plü07] die Funktionen **smaller**, **greater**, **grArg** und **smArg** definiert und an Beispielen erläutert.

Im darauffolgenden Abschnitt 2.3 werden Informationen zu echten Funktionstypen in Java [Plü15a] zusammengefasst.

In Abschnitt 2.4 wird der Typunifikationsalgorithmus um zusätzliche Inferenzregeln ergänzt.

Abschnitt 2.5 beschreibt, wie die Berechnung des kartesischen Produktes im vierten Schritt des Algorithmus angepasst werden kann, um den Lösungsraum der Unifikation um bisher nicht enthaltene Fälle zu erweitern.

Anpassungen im vierten Schritt führen dazu, dass die Abbruchbedingung der Rekursion umformuliert werden muss. Diese wird in Abschnitt 2.6 behandelt.

Zuletzt folgt eine Zusammenfassung der durchgeführten Erweiterungen am Typunifikationsalgorithmus in Abschnitt 2.7.

2.1 Beschreibung

Die klassische Unifikation [Rob65,MM82] versucht Unifikatoren zur finden, die zwei Terme identisch machen. Die Typunifikation versucht Unifikatoren zu finden, die Constraints (Paare) der Form $(\theta \oplus \theta')$ erfüllen. Im Fall der Java-Typunifikation

ist der Operator $\oplus \in \{ \lessdot , \lessdot_? , \doteq \}$. Bei der Erfüllung des Constraints ist die Semantik der Operatoren von Bedeutung:

- $(\theta \lessdot \theta^{'})$ bedeutet, um den Constraint zu erfüllen, muss a ein Subtyp von b sein, d.h. $\theta \leq^* \theta^{'}$.
- $(\theta \lessdot_? \theta^{'})$ bedeutet, um den Constraint zu erfüllen, muss $\theta \in \mathbf{smArg}(\theta^{'})$ bzw. $\theta^{'} \in \mathbf{grArg}(\theta)$ sein.
- $(\theta \doteq \theta^{'})$ bedeutet, um den Constraint zu erfüllen, müssen θ und $\theta^{'}$ gleich sein. Dies ist der Fall der klassischen Unifikation.

Der Typunifikationsalgorithmus nach [Plü07] findet alle Unifikatoren, die eine Menge von Constraints Eq erfüllen. Er geht dazu in sieben Schritten vor.

1. Im ersten Schritt werden Inferenzregeln solange auf die Menge Eq angewandt, bis diese sich nicht mehr ändert.
2. Im zweiten Schritt wird eine Menge $Eq_1^{'}$ erstellt, in der alle Paare enthalten sind, deren Elemente beide Typvariablen sind.
3. Im dritten Schritt wird eine Menge $Eq_2^{'}$ erstellt, in der alle Paare enthalten sind, deren Elemente nicht beide Typvariablen sind.
4. Im vierten Schritt wird der Lösungsbaum durch Berechnung eines kartesischen Produktes aufgespannt, d.h. hier werden mögliche Lösungswege erzeugt.
5. Im fünften Schritt werden Typvariablen deren Wert gefunden wurde substituiert. Dies ähnelt der subst-Regel im Martelli-Monatanari-Unifikationsalgorithmus [MM82].
6. Im sechsten Schritt folgt der rekursive Aufruf für Mengen in den substituiert werden konnte und die Vereinigung aller Lösungen.
7. Der siebte Schritt gibt die Typunifikatoren aus.

2.2 Funktionen auf Typen

Sei $<$ die Extends–Relation eines java–Programms und $\leq^*$ die daraus resultierende Subtyp–Relation. Die Funktionen **smaller**, **greater**, **smArg**, **grArg** lassen sich anhand der Finite Closure von $(<)$ (FC$(<)$) (Definition siehe [Plü07]) berechnen. Ergänzend zu [Plü07] werden diese Funktionen hier definiert und beschrieben.

Definition 1 (smaller). *Sei $\leq^*$ die transitive und reflexive Subtyp-Relation auf Extended-Typen wie sie in [Plü07] definiert ist. Dann ist*

$$\boldsymbol{smaller}(\theta) = \{x \mid x \leq^* \theta, \nexists \overline{x} \leq^* \theta, \nexists \text{ Subsitution } \sigma \text{ mit } \sigma(\overline{x}) = x\}.$$

Die Funktion $\boldsymbol{smaller}(\theta)$ *berechnet die Menge aller allgemeinsten*[1] *Typen die in der Subtyp-Relation* $\leq^*$ *kleiner als* θ *sind.*

Beispiel 1. Sei ein Java-Programm gegeben mit (List<T> < Collection<T>), (Integer < Number) und (Number < Object). Dann ist z.B.:

smaller(Object) = {Integer, Number, Object}

smaller(Collection<T>) = {Collection<T>, List<T>}

smaller(Collection<? extends Number>) = {Collection<? extends Number>, Collection<? extends Integer>, Collection<Number>, Collection<Integer>, List<? extends Number>, List<? extends Integer>, List<Number>, List<Integer>}

Definition 2 (greater). *Sei* $\leq^*$ *die transitive und reflexive Subtyp-Relation auf Extended-Typen wie sie in [Plü07] definiert ist. Dann ist*

$$\boldsymbol{greater}(\theta) = \{x \mid \theta \leq^* x\}$$

Die Funktion $\boldsymbol{greater}(\theta)$ *berechnet die Menge aller Typen die in der Subtyp-Relation* $\leq^*$ *größer als* θ *sind.*

Beispiel 2. Sei ein Java-Programm gegeben mit (List<T> < Collection<T>), (Integer < Number) und (Number < Object). Dann ist z.B.:

greater(Integer) = {Integer, Number, Object}

greater(List<T>) = {List<T>, Collection<T>}

greater(List<Number>) = {List<Number>, List<? extends Number>, List<? super Number>, List<? extends Object>, List<? super Integer>, Collection<Number>, Collection<? extends Number>, Collection<? super Number>, Collection<? extends Object>, Collection<? super Integer>}

Definition 3 (smArg). *Die Funktion* $\boldsymbol{smArg}(\theta)$ *berechnet die Menge an Typen* θ *welche, im Argument eines umschließenden Typs C dafür sorgen, dass* $C\langle\theta\rangle \leq^* C\langle\theta'\rangle$.

$$\boldsymbol{smArg}(\theta) = \begin{cases} \{{}^{?}\theta' \mid \bar{\theta} \leq^* \theta'\} \cup \{\theta' \mid \bar{\theta} \leq^* \theta'\} & \theta = {}^{?}\bar{\theta} \\ \{{}_{?}\theta' \mid \theta' \leq^* \bar{\theta}\} \cup \{\theta' \mid \theta' \leq^* \bar{\theta}\} & \theta = {}_{?}\bar{\theta} \\ \{\theta\} & sonst \end{cases}$$

[1] vgl. Example 6 und 7 in [Plü07]

Beispiel 3. Sei ein Java-Programm gegeben mit (List<T> < Collection<T>), (Integer < Number) und (Number < Object). Dann ist z.B.:

smArg(Integer) = {Integer}
smArg(? extends Number) =
{Integer, Number, ? extends Number, ? extends Integer}
smArg(List<? extends Number>) = {List<? extends Number>}
smArg(? extends List<? extends Integer>) =
{? extends List<? extends Integer>, ? extends List<Integer>,
List<Integer>, List<? extends Integer>}

Definition 4 (grArg). *Die Funktion* $\boldsymbol{grArg}(\theta)$ *berechnet die Menge an Typen* θ' *welche, im Argument eines umschließenden Typs C dafür sorgen, dass* $C<\theta> \leq^* C<\theta'>$.

$$\boldsymbol{grArg}(\theta) = \begin{cases} \{{}^{?}\theta' \mid \theta' \leq^* \bar{\theta}\} & \theta = {}^{?}\bar{\theta} \\ \{{}_{?}\theta' \mid \bar{\theta} \leq^* \theta'\} & \theta = {}_{?}\bar{\theta} \\ \{\theta\} \cup \{{}_{?}\theta' \mid \theta \leq^* \theta'\} \cup \{{}^{?}\theta' \mid \theta' \leq^* \theta\} & sonst \end{cases}$$

Beispiel 4. Sei ein Java-Programm gegeben mit (List<T> < Collection<T>), (Integer < Number) und (Number < Object). Dann ist z.B.:

grArg(? super Number) = {? super Number, ? super Integer}
grArg(Integer) = {Integer, ? extends Integer, ? super Integer, ? extends Number,
? extends Object}
grArg(? extends List<? extends Integer>) =
{? extends List<? extends Integer>, ? extends List<? extends Number>,
? extends List<? extends Object>, ? extends Collection<? extends Integer>,
? extends Collection<? extends Number>,
? extends Collection<? extends Object>}

Korollar 1. *Sei* $<$ *eine Extends-Relation eines Java-Programms mit der Eigenschaft, dass in der daraus resultierenden Subtyp-Relation* $\leq^*$ *keine Elemente* $\theta \leq^* C<\ldots, {}_{?}\sigma(\theta), \ldots>$ *(F–bounded Elemente)existieren. Dann sind die Mengen* ***smaller***, ***greater***, ***smArg*** *und* ***grArg*** *endlich.*

Korollar 2. *Sei* $P_{gr}(\theta, \theta^{'}) = \theta \in \boldsymbol{greater}(\theta^{'})$ *ein Prädikat das angibt ob ein Typ* θ *größer als ein Typ* $\theta^{'}$ *ist. Aus der Transitivität und Reflexivität der Subtyp-Relation folgt, dass auch* P_{gr} *transitiv und reflexiv ist. Analoges gilt für die Funktionen* ***smaller****,* ***smArg*** *und* ***grArg****.*

Korollar 3. *Es gilt:*

$$\theta \in \boldsymbol{greater}(\theta^{'}) \Leftrightarrow \theta^{'} \in \boldsymbol{smaller}(\theta)$$

$$\theta \in \boldsymbol{grArg}(\theta^{'}) \Leftrightarrow \theta^{'} \in \boldsymbol{smArg}(\theta)$$

2.3 Funktionstypen

In [Plü15a] werden echte Funktionstypen ($FunN*$-Typen) für Java vorgestellt.

$FunN*$-Typen treten in der FC(<) nur als reflexives Paar auf, d.h. sie sind kein Teil des Vererbungsbaumes, erben nicht, und können nicht vererbt werden. In der Subtyp-Relation unterscheiden sich von Referenztypen, denn für $FunN*$-Typen gilt:

$$FunN*<\theta_0^{'},\theta_1,\ldots,\theta_n> \leq^* FunN*<\theta_0,\theta_1^{'},\ldots,\theta_n^{'}> \text{ gdw } \forall i : \theta_i \leq^* {\theta'}_i$$

$FunN*$-Typen sind also kontravariant bezüglich ihres Rückgabetyps und kovariant bezüglich ihrer Argumente. Aufgrund der impliziten Varianz macht es keinen Sinn Wildcard-Typen in den Argumenten von $FunN*$-Typen zu verwenden. Wildcard-Typen sind in den Argumenten daher verboten [Plü15a]. Das Codebeispiel 1.1 zeigt wie sich die Varianz in Java auswirkt.

2.4 Typinferenzregeln

Wildcardtypen Die Inferenzregeln des Typunifikationsalgorithmus werden um weitere vier Regeln aus Abbildung 1 ergänzt um Wildcardtypen zu reduzieren.

Die Regeln reduceWcLowRight und reduceWcUpRight dürfen nicht angewandt werden wenn eine Typvariable auf der linken Seite steht. Eine Anwendung würde den Lösungsraum weiter als nötig einschränken und valide Lösungen könnten vom Unifikationsalgorithmus nicht mehr gefunden werden. Dies wird im nachfolgenden Beispiel für die Regel reduceWcLowRight gezeigt.

Beispiel 5. Sei die $\mathbf{FC}(<) = \{(\text{Integer} < \text{Number})\}$ und sei das Paar (Vector<a> < Vector<? extends Number>) zu unifizieren. Die Typvariable a kann somit die

```
1  Tuple<Number, Number, Number> t1 = new Tuple(1, 2, 3);
2  Tuple<Integer, Object, Number> t2 = new Tuple(1, 2, 3);
3  Tuple<? extends Number, ? super Number, ? super Number>
       t3 = null;
4  t1 = t2; // Not Possible, will not compile.
5  t3 = t2; // Possible because of explicit wildcard
       variance.
6
7  Fun2<Number,Number,Number> multiply = (x) -> (y) -> x*y;
8  Fun2<Integer, Object, Number> foo =
9          (x) -> (y) -> x.toString().length() * y;
10 multiply = foo; // Possible because of implicit variance
       .
```

Codebeispiel 1.1: Varianz von $FunN*$-Typen

Typen {? extends Number, ? extends Integer, Number, Integer} annehmen. Sei nun die Regel reduceWcLowRight aus Abbildung 1 uneingeschränkt anwendbar.

$$\dfrac{\dfrac{(\text{Vector<a>} \lessdot \text{Vector<? extends Number>})}{(\text{a} \lessdot_? \text{? extends Number})}\ \text{(reduce1, siehe [Plü07])}}{(\text{a} \lessdot \text{Number}))}\ \text{(reduceWcLowRight)}$$

Das Paar (a $\lessdot$ Number)) kann nur noch von a $\in$ {Integer, Number} erfüllt werden. Somit wurde der Lösungsraum um die Typen {? extends Integer, ? extends Number} eingeschränkt und das Ergebnis ist nicht mehr vollständig.

Typvariablen Zusätzlich zu den sieben Wildcard-Inferenzregeln wird der Algorithmus um drei Inferenzregeln für Typvariablen aus Abbildung 2 ergänzt.

Paare, die durch diese Regeln reduziert werden, wurden bisher durch das kartesische Produkt im vierten Schritt des Algorithmus behandelt. Es handelt sich dabei um die Paare der Form $(\theta \lessdot a)$, $({}_?\theta \lessdot_? a)$, $({}^?\theta \lessdot_? a)$. Die genaue Anpassung des vierten Schrittes wird in Abschnitt 2.5 beschrieben.

Funktionstypen In Abbildung 2.4 sind die Inferenzregeln für Funktionstypen aufgeführt. Die Unterstützung für Funktionstypen wurde im Rahmen dieser Arbeit erstmals implementiert, daher werden die Regeln hier aus Gründen der Vollständigkeit aufgeführt.

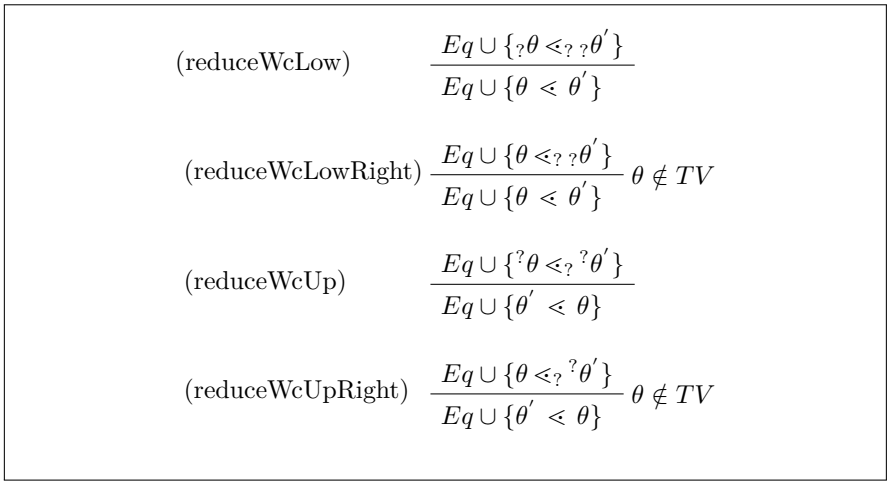


Abb. 1: Typinferenzregeln für Wildcards

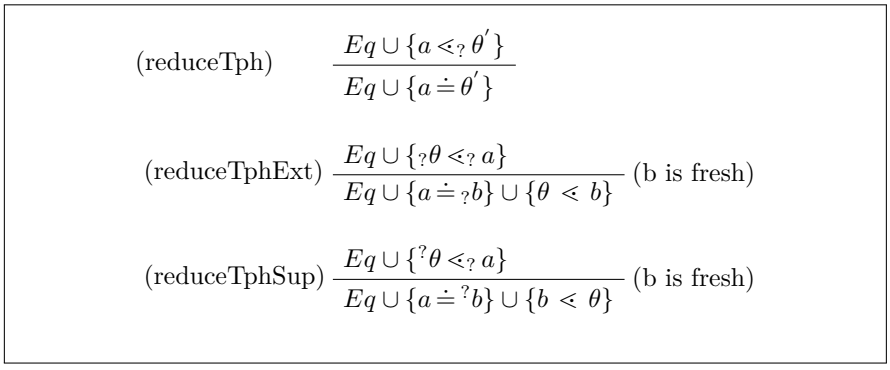


Abb. 2: Typinferenzregeln für Typvariablen

$$(\text{reduce}FunN*)\quad \frac{Eq \cup \{FunN*<\theta,\theta_1^{'},\ldots,\theta_N^{'}> \lessdot FunN*<\theta^{'},\theta_1,\ldots,\theta_N>\}}{Eq \cup \{\theta \lessdot \theta^{'}, \theta_1 \lessdot \theta_1^{'}, \ldots, \theta_N \lessdot \theta_N^{'}\}}$$

$$(\text{greater}FunN*)\quad \frac{Eq \cup \{FunN*<\theta,\theta_1^{'},\ldots,\theta_N^{'}> \lessdot a\}}{Eq \cup \{a \doteq FunN*<b^{'},b_1,\ldots,b_N>, \theta \lessdot b^{'}, b_i \lessdot \theta_i^{'}\}}$$

where $b^{'}$, b_i are fresh

$$(\text{smaller}FunN*)\quad \frac{Eq \cup \{a \lessdot FunN*<\theta^{'},\theta_1,\ldots,\theta_N>\}}{Eq \cup \{a \doteq FunN*<b,b_1^{'},\ldots,b_N^{'}>, b \lessdot \theta^{'}, \theta_i \lessdot b_i^{'}\}}$$

where b, $b_i^{'}$ are fresh

Abb. 3: Typinferenzregeln für Funktionstypen
Quelle: [Plü15a]

2.5 Kartesisches Produkt

In vierten Schritt des Typunifikationsalgorithmus werden mögliche Lösungen durch Berechnung eines kartesischen Produktes gebildet. Haben Paare eine bestimmte Form, kann es vorkommen, dass der Lösungsraum zu weit eingeschränkt wird und bestimmte Lösungen nicht betrachtet werden.

Dies kann geschehen bei Paaren der Form $(a \lessdot_? {}_?b)$, $(a \lessdot_? {}^?b)$, $({}_?b \lessdot a)$, $({}^?b \lessdot_? a)$ sowie bei Paaren der Form $(\theta_b \lessdot_? a)$, $(\theta_b \lessdot a)$, $(a \lessdot \theta_b)$ wobei θ_b bedeutet dass b in θ vorkommt.

Dies wird im Folgenden anhand von zwei Beispielen verdeutlicht. Anschließend wird eine Anpassung am kartesischen Produkt durchgeführt um die zusätzlichen Fälle miteinzubeziehen.

Beispiel 6. Das folgende Beispiel beschäftigt sich mit Paaren der Form $(a \lessdot_? {}_?b)$ stellvertretend für alle Paare der Form $(a \lessdot_? {}_?b)$, $(a \lessdot_? {}^?b)$, $({}_?b \lessdot a)$ und $({}^?b \lessdot_? a)$.

Seien folgende Constraints zu unifizieren:

$$Eq = \{(\text{Vector<a>} \lessdot \text{Vector<? super } b\text{>})), (b \doteq \text{Integer})\} \text{ mit}$$
$$\mathbf{FC}(<) = \{(\text{Integer} < \text{Number})\}$$

Durch Anwendung der reduce1-Regel erhält man:

$$Eq = \{(a \lessdot_? ? \text{ super } b), (b \doteq \text{Integer})\}$$

Dies führt nach [Plü07] zu folgenden kartesischen Produkt:

$$Eq_{set}^{'} = (\bigotimes\{[a \doteq \theta^{'}] \mid \theta^{'} \in \mathbf{smArg}(? \text{ super } b))\}) \times \{[b \doteq \text{Integer}]\}$$

Das Problem tritt nun in der Berechnung von **smArg**(? super b) auf. Nach der Definition aus Abschnitt 2.2 enthält **smArg** alle Werte $\{^{?}\theta^{'} | \theta^{'} \leq^* b\}$. Die Typvariable b befindet sich aber nicht selbst in der Subtyprelation, sondern nur ihre konkreten Ausprägungen. Somit lässt sich **smArg** an dieser Stelle nicht vollständig berechnen.

Um mit dem Algorithmus fortfahren zu können, nehmen wir für **smArg**(? super b) = $\{b, ? \text{ super } b\}$ an, denn dies lässt sich sicher sagen. Es zeigt sich aber, dass durch diese Annahme der Lösungsraum eingeschränkt wurde. Es wird mit dem kartesischen Produkt fortgefahren:

$$\begin{aligned} Eq_{set}^{'} =& (\{[a \doteq b]\} \times \{[a \doteq ? \text{ super } b]\}) \times \{[b \doteq \text{Integer}]\} \\ & \{\{(a \doteq b), (b \doteq \text{Integer})\}, \{(a \doteq ? \text{ super } b), (b \doteq \text{Integer}))\}\} \end{aligned}$$

Durch Anwendung der Substitution ergeben sich zwei Unifikatoren:

$$\begin{aligned} \sigma_1 &= \{(a \mapsto \text{Integer}), (b \mapsto \text{Integer})\} \\ \sigma_2 &= \{(a \mapsto ? \text{ super Integer}), (b \mapsto \text{Integer})\} \end{aligned}$$

Vergleicht man das Ergebnis der Unifikation mit der Ausgangsmenge Eq unter Berücksichtigung der FC($<$), sehen wir, dass folgende Unifikatoren nicht gefunden wurden:

$$\begin{aligned} \sigma_1 &= \{(a \mapsto \text{Number}), (b \mapsto \text{Integer})\} \\ \sigma_2 &= \{(a \mapsto ? \text{ super Number}), (b \mapsto \text{Integer})\} \end{aligned}$$

Beispiel 7. Das folgende Beispiel beschäftigt sich mit Paaren der Form $(\theta_b \lessdot a)$ stellvertretend für alle Paare der Form $(\theta_b \lessdot_? a)$, $(\theta_b \lessdot a)$, $(a \lessdot \theta_b)$, wobei θ_b bedeutet, dass b in θ vorkommt.

Seien folgendene Constraints zu unifizieren:

$$Eq = \{(\text{X<}b\text{>} \lessdot a), (b \doteq \text{? extends Number})\} \text{ mit}$$
$$\mathbf{FC}(<) = \{(\text{Integer} < \text{Number})\}$$

Dies führt nach [Plü07] zu folgenden kartesischen Produkt:

$$Eq'_{set} = (\bigotimes\{[a \doteq \theta'] \mid \theta' \in \mathbf{greater}(\text{X<}b\text{>})\}) \times \{[b \doteq \text{? extends Number}]\}$$

Wie schon im vorherigen Beispiel tritt nun das Problem auf, dass sich die Funktion **greater**(X<b>) nicht vollständig berechnen lässt, da sie vom konkreten Wert der Variablen b abhängt (vergleiche dazu die Definition von **greater** in Abschnitt 2.2 und die Definition der Subtyp-Relation in [Plü15a]). Wie bereits im vorherigen Beispiel, führt dies dazu, dass Lösungen nicht beachtet werden.

Anpassungen

$$\begin{aligned}
&Eq'_{set}\\
&= \{Eq'_1\} \times \Big(\bigotimes_{(a \lessdot \theta') \in Eq'_2} \{([(a \doteq \text{D<}b_1,\ldots,b_m\text{>}), (b_1 \lessdot_? \theta_1), \ldots, (b_m \lessdot_? \theta_m)] \cup \sigma)\\
&\qquad \mid (\bar{\theta} \leq^* \text{C<}\theta_1,\ldots,\theta_n\text{>}) \in \mathbf{FC}(<)\\
&\qquad \bar{\theta}' \in \{\text{C<}\theta'_1,\ldots,\theta'_n\text{>}\\
&\qquad\qquad \mid \theta'_i \in \mathbf{grArg}(\theta_i), 1 \leq i \leq n\}\\
&\qquad \sigma \in Unify(\bar{\theta}', \theta')\\
&\qquad \text{D<}\theta_1,\ldots,\theta_m\text{>} \in \mathbf{smaller}(\sigma(\bar{\theta})),\\
&\qquad b_i \text{ are fresh } \}) \Big)\\
&\quad \times \Big(\bigotimes_{(a \lessdot_? {}_? \theta') \in Eq'_2} \{[(a \lessdot \theta')], [(a \doteq_? b), (b \lessdot \theta')] \mid b \text{ is fresh}\} \Big)\\
&\quad \times \Big(\bigotimes_{(a \lessdot_? {}^? \theta') \in Eq'_2} \{[(\theta' \lessdot a)], [(a \doteq^? b), (\theta' \lessdot b)] \mid b \text{ is fresh}\} \Big)\\
&\quad \times \Big(\bigotimes_{(\theta \lessdot a) \in Eq'_2} \{[(a \doteq \text{C<}b_1,\ldots,b_m\text{>}), (\theta_1 \lessdot_? b_1), \ldots, (\theta_m \lessdot_? b_m)]\\
&\qquad \mid \text{C<}\theta_1,\ldots,\theta_m\text{>} \in \mathbf{greater}(\theta),\\
&\qquad b_i \text{ are fresh}\} \Big)\\
&\quad \times \Big(\bigotimes_{(\theta \lessdot_? a) \in Eq'_2} \{[(a \doteq \theta)], [(a \doteq_? b), (\theta \lessdot b)], [(a \doteq_? b), (b \lessdot \theta)]\\
&\qquad \mid b \text{ is fresh}\} \Big)\\
&\quad \times \{[a \doteq \theta \mid (a \doteq \theta) \in Eq'_2]\}
\end{aligned}$$

Die Beispiele zeigen, dass unter Umständen Lösungen nicht gefunden werden, wenn eine der Funktionen **greater**, **smaller**, **grArg** oder **smArg** auf eine Typvariable b (oder einen Typ θ_b) angewendet werden soll. Dieses Problem lässt sich lösen, indem eine neue Typvariable $b^{'}$ eingeführt wird, welche die alte Typvariable b ersetzt und gleichzeitig in einem zusätzlichen Constraint (d.h. Paar) von b gebunden wird. Dadurch lässt sich die Berechnung der Funktion auf einen späteren Zeitpunkt verschieben, zu dem der Wert von b bekannt ist.

Es fällt auf, dass im Vergleich zu Schritt vier aus [Plü07] die Fälle $(\theta \lessdot a)$, $(_?\theta \lessdot_? a)$ und $(^?\theta \lessdot_? a)$ nicht mehr im kartesischen Produkt behandelt werden. Diese Fälle konnten durch die Einführung neuer Typvariablen vereinfacht und in die Inferenzregeln aus Abbildung 2 ausgelagert werden. Des Weiteren hat sich der Fall $(a \lessdot_? {}_?\theta^{'})$ durch die Einführung der neuen Typvariablen wesentlich vereinfacht.

2.6 Rekursionsfall

In Abschnitt 2.5 wurde eine Anpassung und Vereinfachung des kartesischen Produktes beschrieben. Dazu werden im kartesischen Produkt Paare der Form $(\theta < \theta^{'})$ erzeugt. Dadurch können allerdings Mengen $Eq^{'}$ auftreten, auf welche die subst-Regel nicht angewandt werden kann, die aber dennoch durch einen rekursiven Aufruf unifizierbar sind. Daher muss die Abbruchbedingung der Rekursion aus Schritt 6a abgeändert werden, um in diesen Fällen einen rekursiven Aufruf zuzulassen:

6. (a) Foreach $Eq^{'} \in Eq^{'}_{set}$ where $Eq^{'} \neq Eq$ start again with the first step.

Das bedeutet das ein rekursiver Aufruf erfolgt, solange Änderungen an der Eingabemenge Eq durchgeführt werden konnten.

2.7 Zusammenfassung der Anpassungen

In diesem Kapitel wurde der Typunifikationsalgorithmus auf einer konzeptionellen Ebene behandelt. Dabei wurden Anpassungen durchgeführt, die den Algorithmus um bisher fehlende Fälle ergänzt haben. Hier werden die Änderungen gegenüber dem Algorithmus aus [Plü15a] noch einmal zusammengefasst.

Schritt 1 In Abschnitt 2.4 werden Regeln für Wildcards (Abbildung 1), Regeln für Typvariablen (Abbildung 2) und die in [Plü15a] eingeführten Regeln für Funktionstypen (Abbildung 3) ergänzt.

Schritt 4 In Abschnitt 2.5 wird das kartesische Produkt angepasst um einzelne, bisher nicht enthaltene Fälle berechnen zu können. Dafür werden neue Typvariablen eingeführt, was zur Folge hat, dass sich Teile des kartesischen Produktes wesentlich vereinfachen und in Regeln auslagern lassen.

Schritt 6 Durch die Anpassung des kartesischen Produktes, muss die Abbruchbedingung der Rekursion geändert werden. Dies wird in Abschnitt 2.6 beschrieben.

3 Implementierung

Dieses Kapitel beschreibt zwei Aspekte der Neuimplementierung des Typunifikationsalgorithmus. Dafür werden in Abschnitt 3.1 zunächst unveränderliche Datentypen und der neue Java-Datentyp `Optional` eingeführt.

Im Abschnitt 3.2 wird schließlich auf die Parallelisierung der Implementierung eingegangen.

3.1 Grundlagen

Unveränderlichkeit Unveränderliche Datenstrukturen werden in der funktionalen Programmierung genutzt. Operationen auf diesen Datenstrukturen ändern nicht die Struktur selbst, sondern erzeugen eine, bis auf die erwirkte Änderung, identische Kopie. Unveränderliche Datenstrukturen haben also nur einen einzigen Zustand.

Um Klassen in Java als unveränderlich zu implementieren müssen vier Kriterien erfüllt sein [Ora16].

1. Es darf keine `Setter`-Methoden geben (oder `Setter`-Methoden erzeugen ein neues, geändertes Objekt).
2. Alle Felder müssen `final` und `private` sein, damit sie nicht von erbenden Klassen manipuliert werden können.
3. Unterklassen dürfen nicht in der Lage sein Methoden zu überschreiben. Das kann erreicht werden indem die Klasse als `final` deklariert wird.
4. Enthält die Klasse veränderliche Objekte, dürfen diese nicht verändert werden und keine Referenz darf von außen auf das Objekt gehalten werden können.

Optional Die Klasse `Optional` wurde mit der Java-Stream Bibliothek eingeführt. `Optional` entspricht dem Maybe-Datentyp wie er z.B. in Haskell genutzt wird.

Bei `Optional` handelt es sich um einen generischen Typ. Eine Instanz ist entweder `Optional.empty` oder enthält ein Objekt des entsprechenden Typs.

Für die Implementierung des Typunifikationsalgorithmus werden `Optionals` als Rückgabewerte genutzt. Eine Methode die ein `Optional` zurückgibt, kann durch Rückgabe von `Optional.emtpy` mitteilen, dass sie für die Eingabe nicht anwendbar oder nicht definiert ($\perp$) ist.

`Optionals` dienen zur Vermeidung von `null`-Werten. Durch Rückgabe von `Optionals` wird dem Aufrufenden einer Methode, klarer als durch Rückgabe von `null`, deutlich gemacht, dass diese Methode für bestimmte Eingaben undefiniert ist. `NullPointerExceptions` werden vermieden.

3.2 Parallelisierung

Für die Parallelisierung bieten sich verschiedene Ansätze an. Ein naiver Ansatz ist es, für jeden rekursiven Aufruf im sechsten Schritt einen neuen Thread zu starten. Eine weitere Möglichkeit ist die Parallelisierung mit Javas `parallelStream`. In der Praxis zeigt sich aber, dass beide Ansätze sogar langsamer als eine serielle Ausführung sind.

Die dritte Möglichkeit ist die Nutzung eines Fork-Join-Pools, welcher sich sehr gut zur Parallelisierung rekursiver Algorithmen eignet. Durch die begrenzte Anzahl von Threads im Pool, bleibt der Overhead zur Erstellung neuer Threads gering. Unter Nutzung des Fork-Join-Ansatzes zeigt sich eine wesentliche Verbesserung der Laufzeit des Algorithmus.

4 Zusammenfassung und Ausblick

Der Typunifikationsalgorithmus wurde um bisher nicht beachtete Fälle erweitert, aufgrund derer bestimmte Lösungen nicht gefunden wurden. Dafür wurden neue Inferenzregeln für Wildcards und Typvariablen eingefügt. Außerdem wurde die Berechnung des kartesischen Produktes angepasst. Durch die Änderungen konnte der Algorithmus, insbesondere die Berechnung des kartesischen Produktes, vereinfacht werden.

Die Neuimplementierung unterstützt nun auch echte Java-Funktionstypen. Die Implementierung wurde nach softwaretechnischen Prinzipien entwickelt, so dass Erweiterungen und Anpassungen in der Zukunft leichter durchzuführen sind. Laufzeit und Speicherplatzverbrauch erheblich gesenkt werden, so dass die Typunifikation nun effizienter ist. Dies wurde vor allem durch unveränderliche

Datenstrukturen erreicht. Obwohl die Effizienz erheblich gesteigert wurde, kann die Unifikation bei mittleren und großen Java-Programmen immer noch viel Zeit in Anspruch nehmen.

Ein Ziel ist es, die Laufzeit und Effizienz der Implementierung weiter zu verbessern. Dazu könnten z.B. durch Profiling-Tools besonders rechenintensive Operationen ermittelt und gezielt verbessert werden. Außerdem kann die Laufzeit möglicherweise durch weitere Parallelisierung, z.B. des kartesischen Produktes verbessert werden.

Literatur

LP07. Arne Lüdtke and Martin Plümicke. Implementierung eines Typinferenzalgorithmus für Java 5.0. In Walter Dosch, Clemens Grelck, and Annette Stümpel, editors, *Tagungsband des 14. Kolloquiums Programmiersprachen und Grundlagen der Programmierung*, number A–07–07 in Berichte der Institute für Informatik und Mathematik, pages 141–146. Universtät zu Lübeck, 2007. (in german).

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

Ora16. Oracle. A Strategy for Defining Immutable Objects. https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html, 2016. Abruf am 17.04.2016.

Plü07. Martin Plümicke. Java type unification with wildcards. In *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, pages 223–240, 2007.

Plü15a. Martin Plümicke. Java type system – proposals for java 10 or 11. In Jens Knoop, editor, *Tagungsband des 18. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS'15)*, Pörtschach, 2015. Technische Berichte der TU Wien.

Plü15b. Martin Plümicke. More type inference in Java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

Rob65. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

Sta15. Andreas Stadelmeier. Java type inference as an eclipse plugin. In *45. Jahrestagung der Gesellschaft für Informatik, Informatik 2015, Informatik, Energie und Umwelt, 28. September - 2. Oktober 2015 in Cottbus, Deutschland*, pages 1841–1852, 2015.

Model-Driven-Development im Java-TX Projekt

Jan-Elric Neumann und Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart
Department of Computer Science
jan-elric@web.de, pl@dhbw.de

Abstract. In this study we use the tool, ObjectiF by microTOOL GmbH, to parse the existing Java-TX code to create new and up-to-date UML-models. ObjectiF doesn't only support the creation of simple UML-Models but gives the ability to conduct the full scale of model-driven-development and round-trip-engineering.

Keywords: Model-driven-development, Round-Trip-Engineering, UML-Models

1. Einleitung

Seit ca. 10 Jahren wird an der DHBW der Java-Compiler Java-TX [5] entwickelt. Hauptfunktion von Java-TX ist es, Java um die automatische Berechnung von Typen zu erweitern. Durch bisher 20 Studenten der DHBW wurde dieser Compiler entwickelt und in die Entwicklungsumgebung Eclipse eingebunden. Die Implementierung des Java-TX-Compilers basiert auf veralteten UML-Modellen, die mit einem UML-Tool erzeugt wurden, welches stark veraltet ist. Aufgabe dieser Arbeit ist es, aus dem bestehenden Quellcode des Compilers durch die Verwendung des Tool ObjectiF der microTOOL GmbH [6], neue UML-Modelle zu generieren. Die Erzeugung der Modelle soll möglichst automatisiert stattfinden.
Die Arbeit ist wie folgt aufgebaut. Im zweiten Kapitel wird mit der Vorstellung des Software-Engineering-Tools begonnen, welches Entwickler bei der Anwendung von Model-Driven-Development [1,2,3] und Round-Trip-Engineering unterstützt. Die konkrete Umsetzung des Round-Trip-Engineerings im Java-TX Projekt wird in Kapitel 3 beschrieben. Abschließend wird eine Zusammenfassung und ein Ausblick gegeben.

2. ObjectiF

ObjectiF ist ein Softwaretool der microTOOL GmbH, welches für die modellgetriebene Entwicklung von Softwareprodukten eingesetzt werden kann. Es unterstützt die Modellierungssprachen Business Process Modeling Notation (kurz: BPMN) und UML, zwei Standards verwaltet durch die OMG ([4] S. 1), und die Programmiersprachen Java, C++, C# und .NET ([6] S. 13). Um die Einbettung von ObjectiF in

bestehende Entwicklungsumgebungen und –prozesse zu erleichtern, unterstützt das Tool eine Integration in die IDEs „Eclipse“ und „Microsoft Visual Studio“ ([6] S. 14). ObjectiF unterstützt fast alle Formen der Code-Modell-Bindung wie sie in Abbildung 2.6 Beziehung zwischen Quellmodell und Marking ([3] S. 249) dargestellt sind. Mit dem Tool kann bestehender Code durch das Reverse-Engineering in Modelle umgewandelt werden um darauf basierend das Round-Trip-Engineering anzuwenden. Auch ist es möglich, Formen des Model-Driven-Development (MDD) mit ObjectiF durchzuführen oder ObjectiF als reines UML-Tool zu verwenden, um das Softwaresystem zu modellieren und die Implementation unabhängig von ObjectiF durchzuführen. Modelle, Diagramme und Code sind in ObjectiF stark gekoppelt. Veränderungen an einer Stelle wirken sich auf alle anderen Bereiche aus. Wird beispielsweise eine Java-Klasse im Code durch eine Methode erweitert, wird diese Methode der UML-Repräsentation dieser Klasse hinzugefügt, und die Darstellung der UML-Repräsentation in allen Diagrammen aktualisiert. Wird eine Vererbungshierarchie einer Klasse durch eine Aktion in einem Klassendiagramm hinzugefügt, wird diese im Code automatisch übernommen.
ObjectiF unterstützt eine Auswahl an Diagrammtypen definiert durch die UML: Anwendungsfall-, Aktivitäts-, Sequenz-, Zustands-, Paket- und Klassendiagramme lassen sich mittels ObjectiF anlegen. Zudem sind Geschäftsprozessdiagrammen aus der BPMN unterstützt.

2.1 Bedienung

Das wichtigste Bedienelement in ObjectiF ist das Kontextmenü. Dies wird mit der rechten Maustaste aufgerufen. Die aufgelisteten Optionen sind namensgebend vom den Kotext abhängig, in dem das Menü geöffnet wurde. Der Kontext ist bestimmt durch das Element, welches sich zum Zeitpunkt des Rechtsklicks unter dem Cursor befindet. Abbildung 2.1 zeigt das Kontextmenü eines ObjectiF-Package.

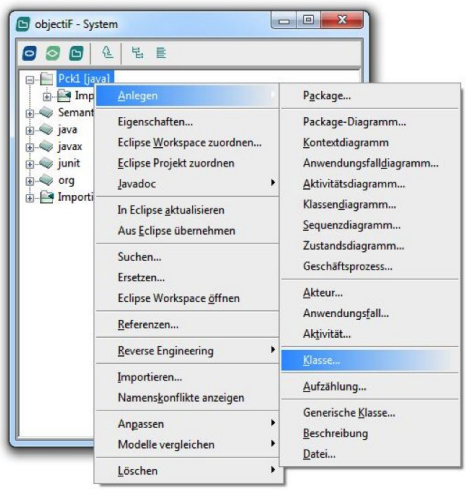


Abbildung 2.1 Kontextmenü eines Package

ObjectiF-System

Jedes Projekt wird in ObjectiF von weiteren Projekten durch das Anlegen eines individuellen ObjectiF-Systems getrennt. Ein ObjectiF-System bündelt alle Informationen, Modelle und Konfigurationen eines Projektes. Beim Start von ObjectiF bietet das Tool eine Auswahl zwischen dem Anlegen eines neuen Systems oder das Öffnen eines bestehenden.
Beim Anlegen eines neuen Systems muss eine Systemvorlage gewählt werden. Diese Vorlage bestimmt, welche Programmiersprache für das neue Projekt verwendet werden soll. Durch die Vorlage werden dem ObjectiF-System automatisch die verfügbaren Elemente der Programmiersprache (z.B. C++ class, template, struct, etc.), ihre Basistypen (z.B. Java bool, int, string, etc.) und die Standardbibliotheken der Sprache hinzugefügt. Ebenso werden, sofern die Vorlage MDD unterstützt, entsprechende Standardtransformatoren hinzugefügt ([6] S. 38). Bei Vorlagen, die keine Programmiersprache zu Beginn festlegen, kann diese auch an einem späteren Zeitpunkt gewählt werden ([6] S. 46).
Ein ObjectiF-System gibt drei Sichten vor, welche unterschiedliche Elemente des Projektes sichtbar machen: Fachliches Modell, Technisches Modell und System. Die Sicht System enthält alle Elemente des ObjectiF-Systems, die ersten zwei Sichten bekommen erst mit der Anwendung von MDD eine Bedeutung. Im weiteren Verlauf wird davon ausgegangen, dass die System-Sicht gewählt ist.

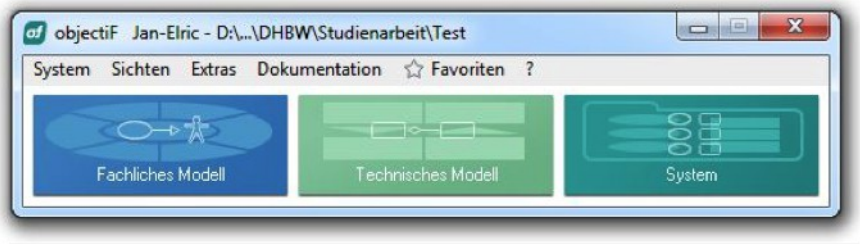


Abbildung 2.2 ObjectiF Sichten

Package

ObjectiF bietet die Möglichkeit, dass modellierte System in Packages aufzuteilen. Dies bietet die Möglichkeit, zusammengehörende Modelle und Diagramme in Modulen zu bündeln. Ein Package kann weitere Packages enthalten. Die Notation der Packages entspricht der UML.
Ein Package definiert in ObjectiF einen eigenen Namensraum. Elemente eines Packages haben Kenntnis von einander, sind diese Elemente zudem öffentlich, können sie auch von Elementen außerhalb des Package referenziert werden. Die öffentlichen Elemente eines Packages definieren dadurch seine Schnittstellen (Ports).

Besitzt die Programmiersprache des ObjectiF-Systems ebenfalls ein Konzept, welches mit einem Package vergleichbar ist, wird der generierte Code ebenfalls in solche unterteilt.
Ein Package kann mit einem Stereotypen versehen werden. Im Kontext des MDD könnten diese Stereotypen „`<<Platform Independent Model>>`“ und „`<<Platform Specific Model>>`“ lauten. Sollte das ObjectiF-System keine Programmiersprache definiert haben, oder möchte man die Programmiersprache innerhalb eines Package ändern, kann einem Package eine Zielsprache zugewiesen werden. Im Kontext des MDD ermöglicht dies, mehrere Platform-Specific-Models (PSMs) mit potentiell unterschiedlichen Zielsprachen innerhalb eines ObjectiF-Systems zu definieren.

Klassen

Klassen bilden die zentrale Einheit aller Modellierungen von (Software-) Systemen. Sie werden für das Systemdesign eingesetzt und bilden die Basis für die Implementation des Systems. Klassen werden dazu eingesetzt, um die fachlichen und technischen Anforderungen an eine Software zu erfüllen ([6] S. 98). ObjectiF unterstützt den vollen Umfang der Klasse im Sinne der UML.
Ähnlich den ObjectiF-Packages bilden Klassen einen eigenen Namesraum. Öffentliche Elemente in den Klassen bilden ihre Schnittstelle zu weiteren Klassen. Abhängig von der Sprache des umschließenden Package können einer Klasse Elemente zugeordnet werden, die der Zielsprache der Klasse entsprechen. Ist die Klasse Teil eines technischen Modells, sind diese Elemente abhängig von der definierten Programmiersprache, ist die Klasse Teil eines fachlichen Modells, entsprechen die Elemente der fachlichen Domäne. Von der Zielsprache hängt auch die Auswahl der Stereotypen ab, die einer Klasse zugewiesen werden können. Erlaubt die Programmiersprache generische Klassen (Java, C++ (template)), können diese ebenfalls angelegt werden.
Beziehungen zwischen einzelnen Klassen werden innerhalb von Klassendiagrammen modelliert und visualisiert. Klassendiagramme können, wie alle Elemente, in eine Hierarchie eingeordnet werden. Innerhalb des Klassendiagramms ist es möglich, bestehende Klassen einzufügen oder neue Klassen anzulegen.
Klassendiagramme unterstützen vier Arten der Beziehung zwischen den einzelnen Klassen. Diese vier Arten entsprechen denen der UML: Generalisierungen, Assoziationen, Kompositionen und Abhängigkeiten. Die Beziehungstypen unterstützen alle, durch die UML definierten, Eigenschaften. Es können Multiplizitäten, Rollennamen, Navigierbarkeiten, Stereotypen, etc. den Beziehungen hinzugefügt werden. Neben Klassen und Klassenbeziehungen können dem Klassendiagramm auch Notizen hinzugefügt werden.
Klassendiagramme werden über das Kontextmenü angelegt. Zunächst ist jedes Klassendiagramm leer.
Abbildung 2.3 zeigt eine Auswahl an Menüpunkten des Klassendiagrammes. Mit den Menüpunkten 1 und 2 ist es möglich, eine neue Klasse anzulegen, oder eine bereits

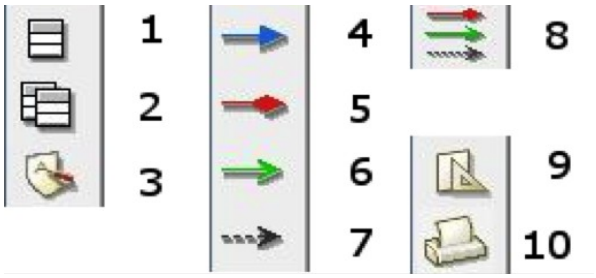


Abbildung 2.3 Klassendiagramm Menü

existierende Klasse dem Diagramm hinzuzufügen. Menüpunkt 3 fügt dem Diagramm eine neue Notiz hinzu.
Beziehungen werden mit den Menüpunkten 4 bis 7 angelegt. Der blaue Pfeil legt Generalisierungen, der rote Pfeil Kompositionen, der grüne Pfeil Assoziationen und der schwarze Pfeil Abhängigkeiten zwischen zwei Klassen an.
Der Menüpunkt 8 wird verwendet, um bestehende, zurzeit nicht visualisierte, Beziehungen zwischen Klassen sichtbar zu machen. Für eine automatische Anordnung der Beziehungspfeile im Diagramm wird der Menüpunkt 9 verwendet, um das Diagramm zu drucken, Menüpunkt 10.
Beziehungen zwischen zwei Klassen können durch das Kontextmenü wieder gelöscht, oder im Diagramm nur verborgen werden.
Es ist möglich, alle Klassendefinitionen zuzuklappen, wodurch nur ihr Name angezeigt wird, oder alle aufzuklappen, um auch Methoden und Attribute aller Klassen anzuzeigen. Dies ist auch für individuelle Klassen möglich. Im Kontextmenü des Klassendiagramms kann zudem gewählt werden, ob der einfache Name der Klasse oder der gesamte Name, also inklusive Package-Zugehörigkeit, angezeigt werden soll.

2.2 Model-Driven-Development mit ObjectiF

ObjectiF unterstützt Model-Driven-Development (MDD). Die Architektur des Systems wird durch ein fachliches Modell definiert, anschließend wird dieses durch Transformatoren in ein oder mehrere technische Modelle umgewandelt. Die Umwandlung von fachlichen zu technischen Modell wird durch die Anpassung der Transformatoren verändert. Codegeneratoren erzeugen anschließend aus dem technischen Modell den modellierten Infrastrukturcode. Markierungen im Quellcode werden eingesetzt, um Bereiche zu markieren, die bei der erneuten Generation von Quellcode nicht überschrieben werden. Innerhalb dieser Bereiche wird die Funktionalität der Anwendung programmiert.
Um die Übersicht des ObjectiF-Systems zu verbessern, bietet ObjectiF neben der System-Sicht noch die Sichten „Fachliches Modell“ und „Technisches Modell“ an. Die „Fachliches Modell“-Sicht blendet alle Teile des Systems aus, die nicht Teil des

Platform Independent Models (PIMs) sind, die „Technisches Modell“-Sicht entsprechend alle Teile, die nicht zu einem Platform-Specific-Model (PSM) gehören.
Fachliche und technische Modelle sind durch Packages voneinander getrennt. Packages der fachlichen Modelle werden durch den Stereotyp „`<<Platform Independent Model>>`“ gekennzeichnet, technische durch den Stereotyp „`<<Platform Specific Model>>`“.
Für das fachliche Modell stellt ObjectiF die BPMN bereit. BPMN umfasst Stereotypen (vgl. MDD: UML-Profile), welche für die fachliche Beschreibung der meisten Systeme ausreichend ist. Diese Stereotypen können durch selbst definierte erweitert werden. Für das technische Modell wird die entsprechende Programmiersprache eingesetzt.
Die Transformatoren von ObjectiF setzen auf den jeweiligen Metamodellen der eingesetzten Sprachen an. Ein Transformator wird durch ein Klassendiagramm beschrieben. Die Klassen in diesen Diagrammen spiegeln die Elemente der Quell- und Zielsprachen wieder. Beispielsweise ist ein „Property“, also ein Attribute einer Klasse in der Sprache C#, durch eine Klasse im Transformator repräsentiert. Die einzelnen Klassen selbst besitzen Stereotypen ihrer Metamodelle. In der Abbildung 2.4 besitzt die Klasse „Class“ den Stereotyp „`<<Element>>`“, da diese ein Element des Metamodells von BPMN repräsentiert.

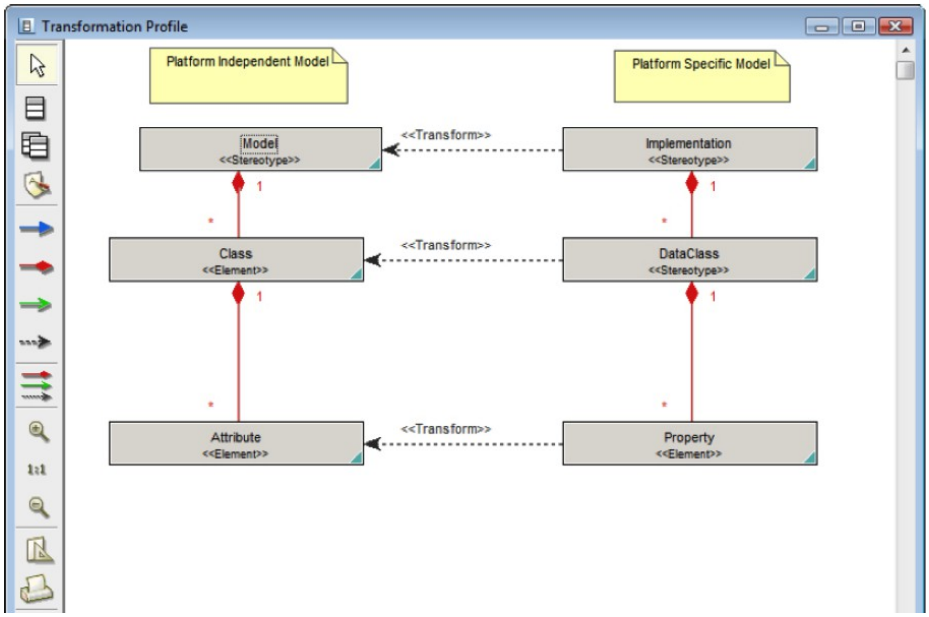


Abbildung 2.4 Transformatoren in ObjectiF ([7] S. 7)

Zwischen den Elementen (im Transformator repräsentiert durch eine Klasse) der Quellsprache und denen der Zielsprache besteht eine Abhängigkeit der Art

„Transform“. Diese Abhängigkeit sorgt dafür, dass die Elemente aus dem Platform Independent Model (PIM) auf diese Elemente des PSM abgebildet werden.

2.3 Round-Trip Engineering mit ObjectiF und Eclipse

ObjectiF bietet die Möglichkeit der Integration mit der Java IDE Eclipse. Dies ermöglicht es, Software Projekte auf Basis von Java mittels Round-Trip-Engineering zu entwickeln.
Für das Round-Trip-Engineering ist es entscheidend, dass Code und Modell konsistent miteinander gehalten werden. Dies verlangt, dass aus bestehendem Quellcode Modelle extrahiert und aus Modellen korrekter Quellcode generiert werden kann. ObjectiF deckt beide Anforderungen ab.
Eclipse setzt für die Organisation von Softwareprojekten Workspaces ein. Ein Workspace kann genutzt werden, um dort zusammenhängende Projekte zu speichern. Der erste Schritt der Integration von Eclipse und einem ObjectiF-System ist es, dem ObjectiF-System einem Eclipse Workspace zuzuordnen. Hierfür gibt es zwei Varianten. Standardmäßig legt ObjectiF für ein ObjectiF-System, welches die Programmiersprache Java nutzt, in dem ObjectiF-System-Verzeichnis einen seperaten Eclipse Workspace an. Ein ObjectiF-System kann entweder mit diesem, oder mit einem anderen bereits bestehendem Workspace verbunden werden. Dazu wird im ObjectiF-System der Kontextmenüpunkt „Eclipse Workspace zuordnen.“

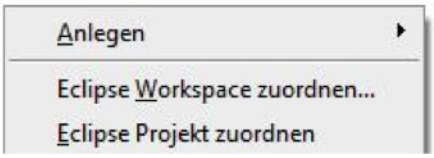


Abbildung 2.5 Eclipse Workspace zuordnen

Anschließend muss dem ObjectiF-System mit dem Kontextmenüpunkt „Eclipse Projekt zuordnen“ ein Eclipse Projekt zugeordnet werden. ObjectiF liest automatisch das Eclipse Projet ein und führt ein anfängliches Reverse-Engineering durch. Hierbei wird aus dem Code in dem Eclipse Projekt Packages und Klassen in ObjectiF angelegt. Dies kann bei großen Eclipse Projekten einige Zeit in Anspruch nehmen. Ein Nachteil hierbei besteht darin, dass ObjectiF keine Fehlermeldungen ausgibt, wenn das automatische Reverse-Engineering fehlschlägt. Daher ist es ratsam, dass Reverse-Engineering manuell durch den Kontextmenüpunkt „aus Eclipse übernehmen“ auszulösen.

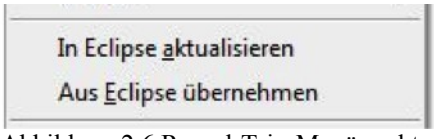


Abbildung 2.6 Round-Trip-Menüpunkte

Mit dem Menüpunkt „Aus Eclipse übernehmen“ wird der Quellcode aus Eclipse zurückgeführt. Dabei werden automatisch Packages und Klassen, die in Eclipse nicht mehr existieren, in ObjectiF ebenfalls gelöscht.
Die Übernahme von Änderungen innerhalb von ObjectiF in Eclipse findet im Gegensatz nicht automatisch statt. Der Vorgang der Codegenerierung wird in ObjectiF durch den Kontextmenüpunkt „In Eclipse übernehmen“ ausgelöst. Auch hierbei werden Packages und Klasse, die in ObjectiF nicht mehr existieren in Eclipse entfernt.
Beide Aktionen können entweder für das gesamte System, oder für eine bestimme Klasse, durchgeführt werden. Soll nur eine einzelne Klasse, oder ein Package, hierbei ist der Vorgang derselbe, muss sich diese unter dem Cursor befinden, bevor das Kontextmenü aufgerufen wird.

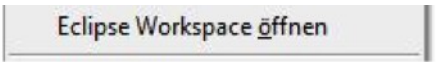


Abbildung 2.7 Eclipse durch ObjectiF starten

Eclipse muss für das Round-Trip-Engineering stets aus ObjectiF heraus gestartet werden (Abbildung 2.7). Dies wird durch den Kontextmenüpunkt „Eclipse Workspace öffnen“ ausgelöst. Gegebenfalls muss ObjectiF im Administratorrechte besitzen, um Eclipse starten zu können.

3. Round-Trip-Engineering für Java-TX

3.1 Vorbereitung

Die erfolgreiche Integration von Eclipse in ObjectiF war die größte Hürde dieser Arbeit. Das korrekte Vorgehen ist in der Dokumentation von ObjectiF nur sehr oberflächlich beschrieben und für einige Punkte nicht korrekt. Schlägt das Aufrufen von Eclipse aus ObjectiF heraus fehl, würde keine aufschlussreiche Fehlermeldung von ObjectiF ausgeben, sondern lediglich die Meldung „Eclipse konnte nicht gestartet werden“.
ObjectiF ist nur in der Lage, Eclipse-Versionen zu nutzen, welche für x86-Architekturen (32 bit) ausgelegt sind. Zudem ist die neuste Version von Eclipse (zu diesem Zeitpunkt Eclipse 4.6 „Neon“) nicht kompatible mit ObjectiF.
Zum Zeitpunkt der Installation ist es nötig, bereits eine kompatible Installation von Eclipse installiert zu haben. Während der Installation von ObjectiF kann auf diese Installation verwiesen werden.
Auf einigen Systemen benötigt ObjectiF für das Starten von Eclipse zusätzlich Administrationsrechte. Die Ursache hierfür bleibt ungeklärt.

3.2 Umgebung

Um einen zentralen Punkt bereitzustellen, an dem die UML-Modelle des Java-TX Quellcodes liegen, existiert ein dedizierter Server. Dieser besitzt eine korrekt konfigurierte ObjectiF installation mit funktionierender Eclipse Integration.
Da das Java-TX Projekt durch GIT verwaltet wird, existiert ein lokales GIT-Repository welches den Quellcode enthält. Durch GIT kann der Quellcode des Java-TX Projekts flexibel über verteilte Entwickler hinweg auf dem aktuellen Stand gehalten werden.
Der Quellcode des Java-TX Projekts ist einem Eclipse Workspace zugeordnet, welcher dem entsprechenden ObjectiF-System zugewiesen ist.

3.3 Klassendiagramme

Alle erstellten Klassendiagramme sind im ObjectiF-Package *de.dhbwstuttgart* zu finden. Die Diagramme sind nach den Packages benannt, die sie enthalten. Sofern ein Subpackage nicht aufgeführt ist, sind die Klassen in dem hierarchisch darüber liegenden Diagramm mit *de.dhbw-stuttgart.syntaxtree.statement.literal* in dem Klassendiagramm *dia_syntaxtree_statement* enthalten.
Der Großteil der Diagramme hat die Beziehungen des Types *dependency* ausgeblendet, um die Übersichtlichkeit zu wahren. Daher ist empfohlen, auf die Funktion, alle Beziehungen anzuzeigen, zu verzichten (siehe Abbildung 2.3, Punkt 8).
Die Beziehungen einer einzelnen Klasse können im Klassenmenü durch den Kontextmenüpunkt dieser Klasse „Beziehungen darstellen“ hinzugefügt werden.

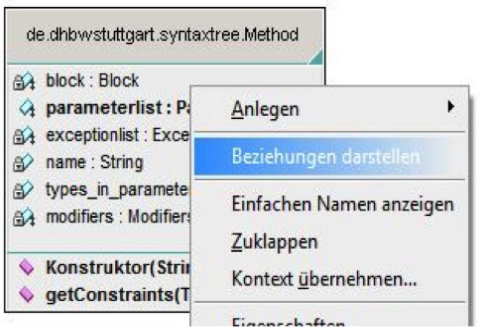


Abbildung 3.1 Beziehung einzelner Klasse darstellen

Da die Beziehungspfeilen beim Verschieben einer Klasse stets neu angeordnet werden, wurde die Anordnung der Pfeile durch die automatische Anordnungsfunktion von ObjectiF durchgeführt (siehe Abbildung 2.3 Klassendiagramm MenüPunkt 9). Dies bietet den Vorteil, dass die Beziehungen neu in ein Diagramm hinzugefügter Klassen ebenfalls durch diese Funktion einfach angeordnet werden können.
Es kann vorkommen, dass die Größe der UML-Klasse im Diagramm zu klein ist, um hinzugefügte Attribute und Methoden anzuzeigen. ObjectiF vergrößert die

Klassendarstellung nicht automatisch. Um sicherzustellen, dass alle Elemente der Klassen angezeigt werden, können im Klassendiagramm alle Klassensymbole zu- und wieder aufgeklappt werden. Anschließend sind alle Elemente garantiert sichtbar. Hiernach können die Beziehungen wieder entsprechend automatisch angeordnet werden.

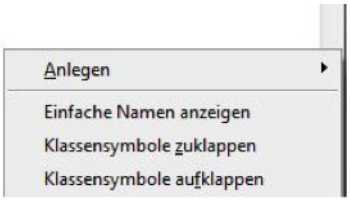


Abbildung 3.2 Klassensymbole auf- und zuklappen

3.4 Fehlerbehebung

Ein Schwachpunkt an ObjectiF ist, dass es keine aufschlussreichen Fehlermeldungen ausgibt. Dies ist vor allem für ein Projekt mit großem Umfang, so wie das Java-TX Projekt, ein Hindernis.
Im folgenden Abschnitt sollen Fehler im Quellcode erläutert werden, die während dieser Studienarbeit aufgetreten sind, und Ansatzpunkte, wie diese behoben werden können.
Wichtigster Ansatzpunkt für die Fehlerbehebung sind die Report-Dateien, die ObjectiF bei jedem zurückführen von Code aus Eclipse, anlegt.
Report-Dateien enthalten jede Aktion, die ObjectiF beim Parsen einer Datei, ausführt.

```
----------------------------------------------------------------

Reverse engineering a file [FILE]...

Class 'ConsoleInterface' ('class ConsoleInterface' ) created

Attribute 'directory' ('static final String directory =
system.getProperty("user.dir")' )    created

Reverse engineering of a file [FILE]

completed with 0 error(s).

................................................................
```

Listing 3.1 Eintrag einer Report-Datei

Ein fehlerfreier Eintrag in einer Report-Datei ist in Listing 3.1 dargestellt. An dieser Stelle wird eine Klasse ‚ConsoleInterface' geparsed und in ObjectiF angelegt, die ein Attribute ‚directory' besitzt.
Report-Dateien werden von ObjectiF im „Temp"-Verzeichnis des aktuellen Windows-Benutzers angelegt (*%userprofile%/AppData/Local/Temp/*). Die Report-Dateien sind nach dem Schema repXXXX.tmp benannt. Da ObjectiF veraltete

Report-Dateien nicht löscht, muss die aktuelle Report-Datei anhand des Erstellungsdatums gewählt werden.

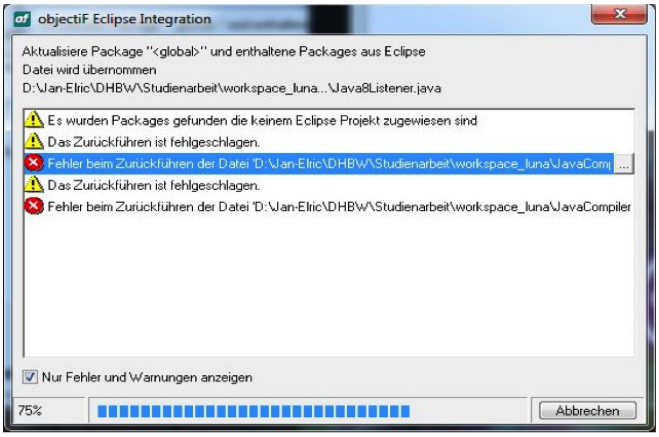


Abbildung 3.3 Nicht fatale Fehler in ObjectiF

Nicht fatale Fehler

Bei der Behandlung vieler Fehler reagiert ObjectiF sehr robust. Syntaxfehler im Quellcode verursacht oft eine Fehlermeldung, diese führt jedoch nicht zum Abbruch des Zurückführens.
Die häufigste Quelle für diese Fehler ist die Instanziierung von generischen Klassen.

```
----------------------------------------------------------------
Reverse engineering a file [FILE]...

Class 'JavaTXCompiler' ('class JavaTXCompiler' )     created

(23):ERROR: unexpected `>`; expected { QUESTIONMARK JAVA_BOOLEAN
JAVA_BYTE CHAR_ DOUBLE_ FLOAT_ INT_ LONG_ SHORT_ VOID_ ID }

Attribute 'sourceFiles' ('List<SourceFile> sourceFiles = new
ArrayList<>()' )     created

Method 'typeInference' ('void typeInference()' )     created

Method 'parse' ('void parse(File sourceFile) throws IOException,
ClassNotFoundException' )   created

Reverse engineering of a file [FILE]

completed with 1 error(s).

................................................................
```

Listing 3.2 Fehlermeldung einer Report-Datei eines nicht-fatalen Fehlers

```
private List<SourceFile> sourceFiles = new ArrayList<>();
```

Listing 3.3 nicht-fatale Fehlerursache

Listing 3.2 zeigt den Eintrag in einer Report-Datei, der einen Fehler enthält. Die konkrete Fehlermeldung ist in roter Schrift hervorgehoben. ObjectiF hat beim Parsen einen vermeintlichen Syntaxfehler im Quellcode gefunden. Listing 3.3 zeigt den Code, der diese Fehlermeldung verursacht hat.
Die Fehlerquelle ist die Deklaration eines privaten, generischen Klassenattributes. Java erlaubt es, für die Instanziierung einer generischen Klasse, die Typparameter entfallen zu lassen, wenn diese aus dem Kontext erschließbar sind. ObjectiF erzeugt für dieses valide Konstrukt jedoch einen Syntaxfehler. Würde man an der entsprechenden Stelle den Typen konkret angeben, entfällt die Fehlermeldung.

```
private List<SourceFile> sourceFiles =
    new ArayList<SourceFile>();
```

Listing 3.4 Behebung des nicht-fatalen Fehlers

Listing 3.4 zeigt die optionale Änderung am Code, um diese Fehlermeldung zu beheben. Da alle generischen Klassen wie in Listing 3.3 instanziiert werden, ist davon abzusehen, den Code dahingehend zu ändern.
Trotz der Fehlermeldung bricht ObjectiF das parsen der Datei und allen restlichen Dateien nicht ab. Ebenso wird der entsprechenden Klasse das Klassenattribut hinzugefügt.

Fatale Fehler

Im Gegensatz zu den nicht fatalen Fehlern verursachen fatale Fehler den Stillstand von ObjectiF und es wird keine Fehlermeldung generiert. Dies kann daran erkannt werden, dass sich ObjectiF unverhältnismäßig lange mit dem Parsen einer einzelnen Datei aufhält. Im Kontext des Java-TX Parsers liegt diese Zeitspanne bei ca. 5 Minuten.
Die Datei, die aktuell von ObjectiF geparsed wird, ist in dem Fenster (Abbildung 3.3 „Nicht fatale Fehler in ObjectiF") abzulesen.
Stellt man fest, dass ObjectiF sich im Stillstand befindet, muss das Zurückführen abgebrochen werden. In der Report-Datei findet sich dann der Grund für den fatalen Fehler. Nachdem der Fehler aus dem Quellcode entfernt wurde, muss das Zurückführen erneut gestartet werden.

Ausgelöst wird dieses Verhalten unter anderem durch validen Java-Code, der ein Konstrukt wie Listing 3.5 fatale Fehlerursache enthält.

```
import examplePackage.* //Enthält die Klasse ExampleClass
[…]
public ExampleClass Method1(){ [...] }
[…]
public examplePackage.ExampleClass Method2(){ [...] }
```

Listing 3.5 fatale Fehlerursache

Dieses Konstrukt, in dem ein Typ über verschiedene Import-Pfade genutzt wird, erzeugt eine Fehlermeldung nach Listing 3.6.

```
[…]
Method 'Method1' ('ExampleClass Method1()' )   created
Method ‘Method2’ (‘examplePackage.ExampleClass Method2()’)
created
(525):ERROR:` ExampleClass`is ambigous.
[…]
```

Listing 3.6 Fehlermeldung einer Report-Datei eines fatalen Fehlers

```
import examplePackage.* //Enthält die Klasse ExampleClass
[...]
public ExampleClass Method1(){ [...] }
[...]
public ExampleClass Method2(){ [...] }
```

Listing 3.7 Behebung des fatalen Fehlers

Durch die Codeänderungen in Listing 3.7 wird dieser fatale Fehler behoben.

4. Zusammenfassung und Ausblick

Diese Arbeit hat die Grundlagen für die Anwendung des Round-Trip-Engineerings für das Java-TX Projekt gelegt.
Es besteht eine korrekt konfiguriertes ObjectiF-System, welches den Java-TX Code erfolgreich einlesen und visualisieren kann. Die Codegenerierung aus ObjectiF und das Zurückführen aus Eclipse heraus kann angewendet werden.
Der Code wurde in sinnvollen Gruppen durch Klassendiagramme dargestellt, welche mit Hilfe von ObjectiF weitestgehend automatisch aktualisiert werden können.
Darüber hinaus ist es mit ObjectiF möglich, dass Round-Trip-Engineering für das Java-TX Projekt anzuwenden.

Für fortführende Arbeiten ist eine solide Basis gebildet worden. Eine Möglichkeit, diese Arbeit fortzusetzen, bestünde in der Untersuchung, inwieweit das Vorgestellte Paradigma des Model-Driven-Development für das Projekt vom Vorteil ist, und wie es für dieses Projekt konkret mit ObjectiF umsetzbar ist.
Eine Funktionalität von ObjectiF, die in dieser Arbeit nicht betrachtet wurde, ist die Möglichkeit, Algorithmen durch Sequenzdiagramme zu modellieren. In wie weit ObjectiF hierfür einsetzbar ist, und ob der Aufwand, bestehende Algorithmen durch Sequenzdiagramme zu modellieren, lohnenswert ist, könnte ebenso ein Gegenstand einer fortführenden Arbeit sein.

References

1. Beydeda, Sami, Book, Matthias und Gruhn, Volker: Model-Driven Software Development. s.l. : Springer Berlin Heidelberg, 2005. 978-3-540-28554-0.
2. Generative Software Engineering Zwickau. Short overview of MDA and MDSD. [Online] [Zitat vom: 20. 05 2017.] http://documentation.genesez.org/en/ch01s01.html#de.Genesez.intro.mdsd.
3. Stahl, Thomas und Völter, Markus. Model-Driven Software Development. s.l. : John Wiley & Sons, Ltd, 2006. 978-0-470-02570-3.
4. Object Modelling Group. Business Process Model and Notation (BPMN). www.omg.org. [Online] 2011. [Zitat vom: 28. 05 2017.] http://www.omg.org/spec/BPMN/2.0/PDF.
5. Plümicke, Martin. Das Java-TX Projekt . [Online] [Zitat vom: 28. 05 2017.] http://www.hb.dhbw-stuttgart.de/~pl/JCC.html.
6. microTOOLS GmbH. ObjectiF Anwenderhandbuch. [PDF] 2015.
7. microTOOL GmbH. ObjectiF - Eigene Modelltransformationen entwickeln.

Functional Array Programming for Cluster Architectures

Thomas Macht[1,2] and Clemens Grelck[2]

[1] VU University Amsterdam
De Boelelaan 1105, 1081 HV Amsterdam, Netherlands
t.macht@student.vu.nl
[2] University of Amsterdam
Informatics Institute
Science Park 904, 1098 XH Amsterdam, Netherlands
c.grelck@uva.nl

Abstract. SAC (Single Assignment C) is a purely functional, data-parallel array programming language that predmoninantly targets compute-intensive applications. Thus, clusters of workstations, or more generally distributed address space supercomputers, form an attractive compilation target. Notwithstanding, SAC today only supports shared address space architectures, graphics accelerators and heterogeneous combinations thereof.

In our current work we aim at closing this gap. At the same time we are determined to uphold SAC's promise of entirely compiler-directed exploitation of concurrency, no matter what the target architecture is. It is well known that distributed memory architectures are going to make this promise a particular challenge.

Despite SAC's functional semantics, it is generally not straightforward to infer exact communication patterns from memory architecture agnostic code. Therefore, we intend to capitalise on recent advances in network technology, namely the closing of the gap between memory bandwidth and network bandwidth. We aim at a solution based on an implementation of software distributed shared memory (SDSM) and large per-node software-managed cache memories. To this effect the functional nature of SAC with its write-once/read-only arrays provides a strategic advantage that we aim to exploit.

Throughout the paper we further motivate our approach, sketch out our implementation strategy and show preliminary experimental evaluation.

1 Introduction

Single Assignment C (SAC) [9] is a functional data parallel language specialised in array programming. The goal of the language is to combine high productivity programming with efficient parallel execution. Data parallelism in SAC is based on array comprehensions in the form of `with`-loops that are used to create immutable arrays and to perform reduction operations. At this point, we can

compile SAC source code into data parallel programs for shared memory architectures, CUDA-enabled graphics accelerators including hybrid systems and the MicroGrid architecture. However, the SAC compiler and runtime system do not yet support symmetric distributed memory architectures like clusters.

Our goal is to add efficient support for distributed memory architectures to the SAC compiler and runtime system. We aim to achieve competitive speedups for high-performance computing applications.

In a shared memory system, all nodes share a common address space. By contrast, in a distributed memory system, each node has a separate address space. In order to access remote data in a distributed memory programming model, the programmer must be aware of the data item's location and use explicit communication. While distributed memory systems can scale to greater size, the shared memory model is simpler to program. Distributed Shared Memory (DSM) aims to combine both models; it provides a shared memory abstraction on top of a distributed memory architecture. DSM can be realised in software or in hardware; hybrid solutions also exist. Partitioned Global Address Space (PGAS) is a programming model that lies in between the local and global view programming models. PGAS logically partitions a global address space such that a portion of it is local to each process, thereby exploiting locality of reference. PGAS is the underlying model of programming languages like Chapel [5].

In the remainder of this paper we will first give an introduction to the SAC language and then motivate our current work, SAC for clusters. Subsequently, we will discuss our implementation and show preliminary performance results.

2 Single Assignment C

Single Assignment C (SAC) is a data parallel language for multi- and many-core architectures. For an introduction to SAC see [9]. The language aims to combine the productivity of high-level programming languages with the performance of hand-parallelized C or Fortran code. As the name suggests, the syntax is inspired by C. Other than C, however, SAC is a functional programming language without side-effects.

SAC is specialised in array programming; it provides multi-dimensional arrays that can be programmed in a shape-independent manner. While the language only includes the most basic array operations it comes with a comprehensive library. Conceptually, SAC's functional semantics requires to copy the full array whenever a single element is updated. To minimise the resulting overhead, SAC uses reference counting. This facilitates in-place updates of data structures when they are no longer referenced elsewhere. See [12] for SAC's memory management.

Array operations are typically implemented by `with`-loops, a type of array comprehension, which comes in three variants. See Figure 1 for examples. Both `genarray` and `modarray` `with`-loops create an array; `modarray` does so based on an existing array. For individual indices or sets of indices, expressions define the value of the corresponding array element(s). Independently for each index,

the associated expression is evaluated and the corresponding array element is initialised. The third `with`-loop variant, `fold`, performs a reduction operation over an index set. As we have do not distribute these `with`-loops, we will not discuss them in this paper.

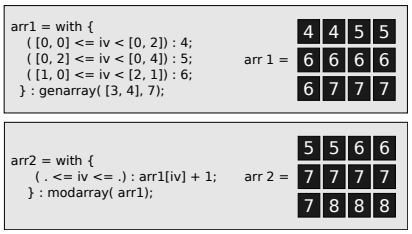


Fig. 1: Examples of genarray and modarray `with`-loops and resulting arrays

All variants of `with`-loops have in common that the compiler may evaluate individual expressions independently of each other in any order and that write-accesses are very restricted. These properties allow us to parallelise `with`-loops in an efficient way. While `with`-loops denote opportunities for parallelism, the decision whether they are actually executed in parallel or not is taken by the compiler and runtime system. At all times, program execution is either sequential or a `with`-loop is processed in parallel.

The SAC compiler is a many-pass compiler and emits platform-specific C code. The compiler spends a lot of effort on combining and optimising `with`-loops [10]. Currently, the compiler includes backends for symmetric multi-cores [8], GPUs (based on CUDA) [14] and the MicroGrid many-core architecture [15]. Heterogeneous systems are supported as well [6] and there have been experiments with OpenMP as a compilation target.

3 Motivation

In this section we argue why it is useful to add support for distributed memory architectures to SAC, why we followed a software DSM-based approach and why we decided to build a custom compiler-integrated DSM system.

3.1 Why support distributed memory architectures?

Distributed memory architectures are more cost-efficient, more scalable, and distributed memory architectures dominate high-performance computing. Currently, 86% of the TOP500 supercomputers are clusters and 14% have a Massively Parallel Processing (MPP) architecture [1] which is also a type of distributed memory system. While they are still predominant in commodity hardware, typical shared memory architectures have long vanished from the TOP500

list: single processors by 1997 and Symmetric multiprocessing (SMP) architectures by 2003 [1].

Message passing, and in particular MPI, is still the prevailing programming model for distributed memory systems [7]. While such a local view or fragmented programming model meets the performance requirements, it lacks programmability [5]. The programmer is responsible for the decomposition and distribution of data structures. Algorithms operate on the local portion of data structures and require explicit communication to access remote data. Data distribution and communication statements obscure the core algorithm.

By contrast, global view programming represents a higher-level alternative. In this model, the programmer works with whole data structures and writes algorithms that operate on these whole data structures. Data transfers and work distribution are handled implicitly. The algorithm is specified as a whole and not interleaved with communication. SAC offers a global view of computation to the programmer. By adding support for distributed memory architectures to SAC, we can utilise its global view programming model to make programming for distributed memory systems more efficient.

3.2 Why a software DSM-based solution?

Distributed Shared Memory (DSM) provides a shared memory abstraction on top of a physically distributed memory. An overview of issues of Distributed Shared Memory (DSM) systems can be found in [17]. DSM can be realised in software or hardware; hybrid systems also exist. In the context of this work, we focus on software solutions. According to [19], the first software DSM system was Ivy which appeared in 1984. Until the early 1990's, several other software DSM systems were proposed. Examples include Linda, Munin and Shiva [17].

These early DSM systems have not been adopted on a large scale due to shortcomings in performance. Explicit message passing, and in particular MPI, remain the predominant programming model for clusters. However, Bharath et al. suggest that it is time to revisit DSM systems [18]. They argue that early DSM systems were not successful because of slow network connections at the time. In the meantime, the picture has changed. Network bandwidth is comparable to main memory bandwidth and network latency is only one order of magnitude worse than main memory latency. According to Bharath et al. these developments reduce DSM to a cache management problem. They propose to use the improved network bandwidth to hide latency. As we will discuss in Section 4, our implementation uses that trick as well.

3.3 Why a custom DSM system?

In order to support distributed memory systems, we could run a SAC program on top of an existing software DSM system. Instead, we decided to integrate a custom DSM system into the SAC compiler and runtime system. This allows us to exploit SAC's functional semantics and its very controlled parallelism in `with`-loops. Since variables in `sac` have write-once/read-only semantics, we do

not have to take into account that they could change their value. Furthermore, parallelism only occurs in `with`-loops and while arbitrary variables can be read in the body `with`-loop, only a single variable is written to.

4 Implementation of our distributed memory backend

We added support for distributed memory architectures to the SAC compiler and runtime system based on a page-based software DSM system. Every node owns part of each distributed array and the owner computes principle applies. All accesses to remote data are performed through a local cache. To abstract from the physical network and provide portability, we utilise existing one-sided Remote Direct Memory Access (RDMA) communication libraries. Currently, we support GASNet [4], GPI-2 [13], ARMCI [16] and MPI-3. In order to add support for a communication library, one only has to provide implementations for a small set of operations. These include initialisation and shut down of the communication system, an operation to load a memory page from a remote node and barriers.

4.1 Distributed arrays and memory model

Distributed execution is triggered by `with`-loops that generate distributed arrays. The runtime system decides whether an array is distributed based on the size of the array, the number of compute nodes and the execution mode at allocation time (see Section 4.4 for execution modes). Arrays are always distributed block-wise along their first dimension. The minimum number of elements per node such that an array gets distributed can be configured at compile time.

In memory, a distributed array does not form one contiguous block, but instead it is split into *number-of-nodes* blocks of memory corresponding to the elements that are owned by each node. We will motivate the choice for this memory model in Section 4.5.

For an illustrative example of the memory model, see Figure 2. The example uses two arrays, denoted by different colours, with fourteen and eight elements, respectively, and four compute nodes. The numbers in the boxes denote the element indices. Every node owns a share of each distributed array. The portion of the array that is owned by a node is located in that node's shared segment (e.g. elements 0 - 3/0 - 1 of the first/second array on Node 0). Note that the array sizes were chosen to simplify the example; in practise only arrays that are some orders of magnitudes bigger would be distributed. Furthermore, we assume for this example that a memory page can hold three array elements only.

Each node's DSM memory consists of memory for the shared segment and memory for the local caches. At program startup, each node pins a configurable amout of memory for its shared segment and reserves an address space of the same size for the caches of each other node's data. (De-)allocation of distributed arrays in DSM memory is taken care of by an adapted version of the SAC private heap manager [11]. When a distributed array is allocated, the runtime system

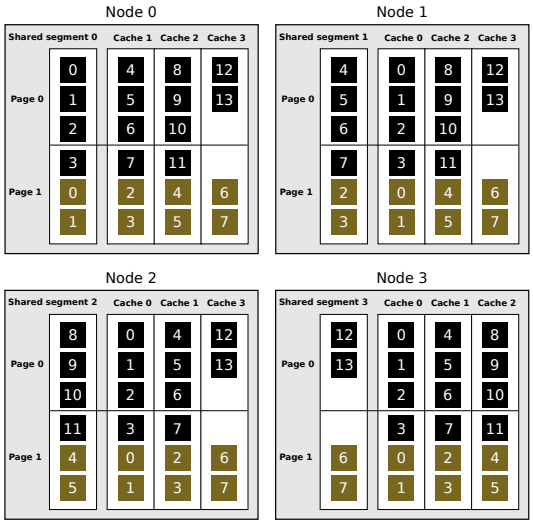


Fig. 2: Memory model for two distributed arrays (distinguished by different colours)

also reserves an address space of the same size within the local caches for all other nodes. To simplify locating array elements, the shared segment and caches are aligned. In the example, the second array starts at offset 4 in the shared DSM segment and all three cache segments on all four compute nodes. Non-distributed arrays, scalars and array descriptors are not allocated in DSM memory.

4.2 Array element pointer calculations

SAC supports multi-dimensional arrays; the translation of multi-dimensional array indices into vector offsets for memory accesses is taken care of by the compiler [3]. For the remainder of this paper, we assume that this conversion has already taken place. As explained in Section 4.1, a distributed array does not form a contiguous block of memory. The runtime system, therefore, needs to translate an offset to an array element to a pointer to the actual location of the element. This section describes how this is done and how we optimise this process.

In SAC, arrays have descriptors that hold a reference counter and, if not known at compile time, shape information. For each distributed array, we add two fields to the array descriptor: `first_elems` and `arr_offs`. The value of `first_elems` is the number of elements that are owned by each node except for the last node, which owns the remaining elements. The value of `arr_offs` is the offset at which the array starts within the shared segment of its owner node and within the cache for the owner at each other node. The formula for the pointer

calculation is shown in Listing 1. The variable `segments` contains pointers to the local shared segment and the local caches; the rank of a node is the index of its segment within `segments`. The value of `elem_offs` is the offset of the requested element within the array assuming that the array would be allocated as one contiguous block of memory.

```
(segments[elem_offs / first_elems] + arr_offs) + (elem_offs % first_elems)
```

Listing 1: Formula for array element pointer calculations

In a naive implementation we would have to perform this pointer calculation for every access to an array element. However, we implemented three optimisations for write accesses, remote read accesses and local read accesses, respectively, so that the calculation can be avoided in most cases.

When writing distributed arrays we know that the elements we are writing to are local to the writing node because of the owner computes principle. We, therefore, simply keep a pointer to the start of the local portion of the array.

For remote read accesses we implemented a pointer cache. For each distributed array, we keep a pointer to the start of the array within the local cache for the node that owns the least recently accessed remote element of that array. In addition, we keep the offset of the first and last element that are owned by the same node.

For local read accesses we use the same pointer to the start of the local portion of the array that we use for write accesses. In addition, we keep the offset of the first and the last element that are local to the current node.

When a read access to an array element occurs, we first check whether the element is local to the current node by comparing its offset to the offsets of the first and last node that are local to the current node. If the element is local, we can use the pointer to the start of the local portion of the array in the local shared segment.

If the element is not local, we check whether it is owned by the same node as the last remote element of the same array that was accessed by comparing the offsets. If that is the case, we can use the pointer to the start of the array in the local cache for that node. Otherwise, we have to perform a pointer calculation as shown in Listing 1 to update the pointer cache.

4.3 Communication model and cache

According to the owner computes rule, a node only writes array elements that it owns. By contrast, every node can read all elements of a distributed array, including remote elements. This section describes the required communication for read accesses to remote array elements.

As mentioned in Section 4.1, the address space for the caches of remote elements is reserved when a distributed array is allocated. Initially, the caches

are protected page-wise against all accesses by means of the `mprotect` system call. When a node tries to access remote data through its local cache, a `SIGSEGV` signal is raised. A custom handler then copies the appropriate memory page from the remote node into the local node's cache and allows accesses to it. Subsequent accesses to the same memory page can then be served directly from the local cache. The signal handler can calculate the requested array element and its location from the address where the segfault occurred. See Listing 1 for how to calculate the memory location of array elements.

When part of the cache becomes outdated, the corresponding memory pages are protected again. Distributed arrays are written in `with`-loops and we do not need any communication to trigger the required cache invalidations. Every node participates in the write operation and, therefore, knows that it has to invalidate the cache for that array on completion.

When a remote element is not in the local cache yet, we always load entire memory pages rather than single array elements. For an example, see Figure 2. When Node 0 first accesses Element 8 of the first array, Elements 9 and 10 will also be fetched from Node 2. Likewise, when Node 1 accesses Element 4 of the second array for the first time, Element 5 of the second and Element 11 of the first array will also be fetched from Node 2.

The rationale for loading entire pages is that thanks to advances in network technology, available bandwidth has increased so much that we can use it to hide latency [18]. Furthermore, the page-based approach allows us to use the operation system's memory page protection mechanism to decide whether an element is present in the cache or not.

4.4 Execution modes and barriers

A distributed memory SAC program is always in one out of three execution modes: replicated, distributed or side effects execution mode. See Figure 3 for an illustrating example. In the following, we call the node with rank 0 master node and the remaining nodes worker nodes.

Program execution starts in replicated execution mode in which every node executes the same instructions on the same data. This way all nodes maintain the same execution environment without requiring communication.

In distributed execution mode, each node works on its share of the data. Currently, `genarray` and `modarray` `with`-loops are distributed iff the result array is distributed. Distributed memory SAC supports one level of distribution, an array and the `with`-loop that writes that array are not distributed if the program is already in distributed execution mode when the array is allocated.

In side effects execution mode, only the master node is executing and the workers are waiting until it is done. This is important because functions that have side effects, such as I/O, must not be executed more than once. If functions with side-effects yield any results, they are broadcast to the workers when the master is done.

In some cases we need barriers to preserve the correctness of the program in a distributed environment. For examples see Figure 3; the horizontal bars

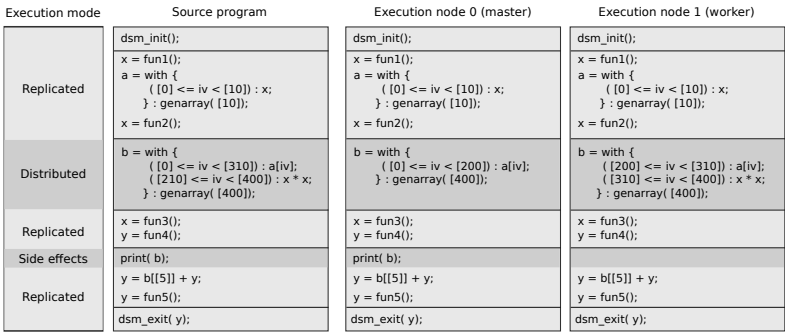


Fig. 3: Execution modes and barriers (horizontal bars)

denote barriers. In general, we require barriers after program startup and before program termination, before and after a distributed `with`-loop and before a function application with side effects.

The barrier after a distributed `with`-loop ensures that no stale data is read by other nodes because there were write accesses to the distributed array in the `with`-loop. The barrier before a distributed `with`-loop ensures that, in case memory is reused (see [12] for SAC's memory management), no other node needs to read the old data anymore before it is overwritten.

4.5 Motivation for memory model

As described in Section 4.1, distributed arrays do not form one contiguous block, but instead are split into *number-of-nodes* blocks of memory corresponding to the elements that are owned by each node. We explained in Section 4.2 that it is relatively expensive to calculate pointers to array elements with this memory model and proposed a pointer cache as a solution. Given this disadvantage, why do we propose the described memory model? For our argumentation we will assume that we use a *page-based* DSM system. We will, therefore, first elaborate on the reasons why we decided to build a *page-based* DSM system: to hide latency and to avoid overheads when checking whether an element is present in the cache.

On a cache miss, we fetch a whole memory page rather than a single element from the remote node that owns the element. Subsequent accesses to neighbouring elements can then be served from the cache. This allows us to use the available bandwidth to hide latency. In addition, if we fetch whole memory pages, we can use the operating system's page protection mechanism to decide whether a page is present in the cache or not. If a page is not present in the cache, a `SIGSEGV` signal is raised when we try to access it and the fetch from the remote node is taken care of by our custom signal handler. If a page is present in

the cache, however, the access simply returns the data. The alternative to using the page protection mechanism would be to keep track of the cached elements ourselves, but that would involve a search in a possibly large data structure. This search would incur additional overheads, also in the case that an element is already present in the cache.

Having decided that we want to use a page based DSM system, why do we use the described memory model? SAC supports multi-dimensional arrays and `with`-loops that generate multi-dimensional arrays are compiled to complex nested loop structures with a loop for each array dimension. We need to make sure that the distribution happens along a single dimension; in practise along the outermost dimension. Otherwise, the iteration of the index space becomes impractically complex, especially when considering that the size and dimensionality of arrays is often not known at compile time.

We have established that we want to use a page-based DSM system and that the distribution of the array should happen along the outermost dimension. If an array was to form a single contiguous form of memory we would then have to partition it at memory page borders. However, we have also established that the distribution should happen along the outermost dimension. Unfortunately, these two demands generally cannot be met at the same time.

Another benefit of our memory model is that it allows us to solve larger problems. With contiguous arrays, we would need to allocate the entire array within the DSM segment so that remote nodes can read the local portion of it. Unfortunately, the size of the DSM segment is limited by hardware constraints. In any case, it cannot be larger than the node's physical memory. By contrast, in our memory model, we only allocate the local part of the array within the DSM segment. The caches are allocated outside of the DSM segment using `mmap`. Until a memory page is accessed for the first time, only an address space is reserved but no phyiscal memory is provided.

5 Evaluation of our distributed memory backend

We evaluate the performance of our distributed memory backend for SAC by means of experiments in the areas of image convolution, matrix multiplication and N-body simulation. In the following, we will first describe the experimental setup and then discuss the results of the individual experiments.

5.1 Experimental setup

All experiments were performed on the VU cluster side of the DAS-4 supercomputer system [2]. The VU cluster side consists of 74 dual quad-core 2.4 GHz compute nodes with 24 GB of memory each. The nodes are interconnected by Gigabit Ethernet as well as high speed InfiniBand. We used the following versions of the supported communication libraries for our experiments: GASNet 1.24.0, GPI-2 1.1.1, ARMCI as included in Global Arrays 5.3 and the Open MPI 1.6.5 implementation of MPI-3.

In our experiments, we compare the runtimes of the program compiled for our distributed memory backend (`dm`) to the runtimes of the sequential SAC program (`seq`). With the distributed program, we start each process on a separate compute node. For $N \leq 8$ (as the nodes of the DAS-4 system have eight cores), we also compare the performance of our distributed memory backend program run by multiple processes on a single node (`dm-sn`) to the performance of the multi-threaded SAC program `mt`.

For all included measurements, we compared the output of the distributed memory backend program to the output of the sequential program to ensure that the program yields correct results. We measure the kernel execution time of the calculations and not the total execution time of the program. The reason is that the setup of the communication libraries and the printing of the result arrays to check the correctness take a considerable amount of time and that would otherwise distort our results. For real-world applications, the compute time would be much longer, whereas the setup time remains nearly constant and, thus, can be neglected.

We performed all experiments at least three times or more often if there was a high variance in the results. From all measurements, we take the minimum execution time for each program version rather than the average execution time. Our justification is that there may be background processes running on the compute nodes that have an influence on our experiments. All reported speedups are with respect to the sequential SAC program (`seq`).

5.2 Image convolution

First, we present our image convolution experiments. We include image convolution in our evaluation because it is a simple application where array element accesses show a high degree of locality. We have optimised our implementation for that by fetching entire pages on a cache miss and by using optimisations such as array pointer caches (see Section 4.2).

The `gaussBlurOpt` test program performs twenty iterations of a 3 x 3 kernel Gaussian blur on a 50,000 x 8,000 = 400,000,000 elements integer array. Figure 4 shows the performance results for `gaussBlurOpt`. For this program, we achieve speedups of more than 80% of linear for up to sixteen nodes.

5.3 Matrix multiplication

We also include experiments with matrix multiplication, because, compared to image convolution, it requires more communication. In this way, the matrix multiplication experiments are a stress test for the communication performance of our distributed memory backend for SAC.

The `matmulBigDiff` program performs ten iterations of a multiplication of two matrices with 2,000 x 2,000 = 4,000,000 double-precision floating point elements each. Implementation-wise, we first transpose the second matrix before we calculate the result matrix. Figure 5 shows our measurements for the

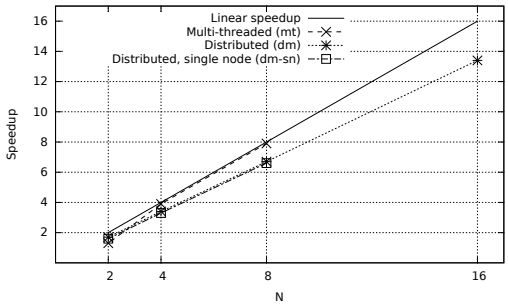


Fig. 4: Speedups for the `gaussBlurOpt` program (twenty iterations of a 3 x 3 kernel Gaussian blur on a 50,000 x 8,000 = 400,000,000 elements integer array)

`matmulBigDiff` program: for eight nodes we achieve a speedup of 3.2 (40% of linear) and for sixteen nodes a speedup of 4.2 (26% of linear).

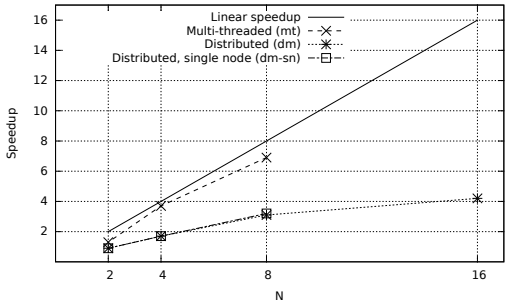


Fig. 5: Speedups for the `matmulBigDiff` program (ten iterations of a multiplication of two matrices with 2,000 x 2,000 = 4,000,000 double-precision floating point elements each)

5.4 N-Body simulation

Finally, we present the measurements for our all-pairs N-body problem experiments. The SICSA N-body challenge simulates the movements of a system of planets in three-dimensional space over time. Our program is based on the SAC implementation proposed in [20].

The `nbodyBig` program performs fifty iterations for 16,384 planets. Figure 6 show the measurements for the `nbodyBig` program. We achieve approximately 50% of linear speedups for up to sixteen nodes.

For the `nbodyBig` program, we also compare the minimum runtimes with the different communication libraries GASNet, ARMCI, GPI-2 and MPI-3. In Figure 7, we can see that MPI shows the weakest overall performance. Overall, GASNet is slightly faster than ARMCI and GPI-2.

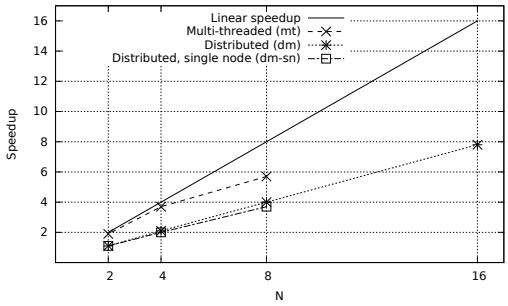


Fig. 6: Speedups for the `nbodyBig` program (N-body simulation: movements of 16,384 planets, 50 iterations)

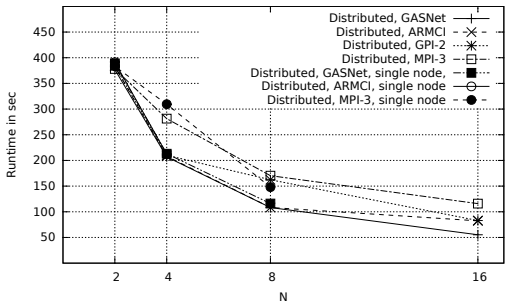


Fig. 7: Minimum runtimes with different communication libraries for the `nbodyBig` program (N-body simulation: movements of 16,384 planets, 50 iterations)

6 Conclusions and future work

6.1 Conclusions

In this paper, we have presented our implementation of a new compiler backend for SAC that supports symmetric distributed memory architectures like clusters of workstations. A particular challenge in doing so is upholding SAC's promise of entirely compiler-directed exploitation of concurrency.

We propose a DSM-based implementation where all accesses to remote data go through large local caches. Initially, the caches are protected by means of the `mprotect` system call. When a memory page is first accessed, a `SIGSEGV` signal is raised. A custom signal handler fetches the requested data from the remote node and subsequent accesses to the same data can be served directly from the cache.

While there is a lot of work to be done, our first results are promising. For our convolution experiments, we achieve 80% of linear speedups, for our N-body simulation approximately 50% of linear speedups and for matrix multiplication about one third of linear speedups.

6.2 Future Work

Possible future research directions lie in the areas of general performance improvements, the combination with multi-threading, cache eviction and distributed I/O. In the following, we briefly elaborate on these topics.

We want to improve overall performance by reducing the number of barriers. Furthermore, we want to make read operations to distributed arrays more efficient by avoiding locality checks and/or reducing overheads caused by them.

To fully utilise clusters of multi-core compute nodes, we want to combine the distributed memory backend with SAC's multi-threaded execution facilities [8]. We expect that we can achieve higher speedups with a hybrid solution that combines distributed execution and multi-threading.

Other than speeding up program execution, distributed execution has another advantage: It allows us to solve problems that do not fit into the memory of a single node. This is already possible to some extent in our solution, but to support the general case, we would need to add a cache eviction scheme.

Currently, functions that have side effects including I/O are only executed by the master node. We decided for this implementation to ensure that existing SAC libraries work correctly with the distributed memory backend. However, in many situations it would be more efficient to distribute I/O operations.

References

1. TOP500 supercomputer sites (2014), `http://top500.org/`, accessed on 19 February 2015
2. DAS-4: Distributed ASCI supercomputer 4 (2015), `http://www.cs.vu.nl/das4/home.shtml`, accessed on 25 July 2015

3. Bernecky, R., Herhut, S., Scholz, S.B., Trojahner, K., Grelck, C., Shafarenko, A.: Index Vector Elimination - Making Index Vectors Affordable, pp. 19–36. Implementation and Application of Functional Languages, Springer (2007)
4. Bonachea, D.: GASNet specification, v1. 1. Tech. rep., University of California at Berkeley (2002)
5. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications 21(3), 291–312 (2007)
6. Diogo, M., Grelck, C.: Towards heterogeneous computing without heterogeneous programming, pp. 279–294. Trends in Functional Programming, Springer (2013)
7. Dongarra, J.J., der Steen, A.V.: High-performance computing systems: Status and outlook. Acta Numerica 21, 379–474 (2012)
8. Grelck, C.: A multithreaded compiler backend for high-level array programming. In: Applied Informatics. pp. 478–484 (2003)
9. Grelck, C.: Single Assignment C (SAC): High Productivity Meets High Performance, pp. 207–278. Central European Functional Programming School, Springer (2012)
10. Grelck, C., Scholz, S.B.: SAC: off-the-shelf support for data-parallelism on multicores. In: Proceedings of the 2007 workshop on Declarative aspects of multicore programming. pp. 25–33. ACM (2007)
11. Grelck, C., Scholz, S.B.: Efficient heap management for declarative data parallel programming on multicores. In: 3rd Workshop on Declarative Aspects of Multicore Programming (DAMP 2008), San Francisco, CA, USA. pp. 17–31 (2008)
12. Grelck, C., Trojahner, K.: Implicit memory management for SAC. In: Implementation and Application of Functional Languages, 16th International Workshop, IFL. vol. 4, pp. 335–348 (2004)
13. Grünewald, D., Simmendinger, C.: The GASPI API specification and its implementation GPI 2.0. In: 7th International Conference on PGAS Programming Models. vol. 243 (2013)
14. Guo, J., Thiyagalingam, J., Scholz, S.B.: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: Proceedings of the sixth workshop on Declarative aspects of multicore programming. pp. 15–24. ACM (2011)
15. Herhut, S., Joslin, C., Scholz, S.B., Grelck, C.: Truly nested data-parallelism: compiling SAC for the Microgrid architecture. Draft proceedings of the 21st International Symposium on Implementation and Application of Functional Languages (2009)
16. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems, pp. 533–546. Parallel and Distributed Processing, Springer (1999)
17. Nitzberg, B., Lo, V.: Distributed shared memory: A survey of issues and algorithms. Distributed Shared Memory-Concepts and Systems pp. 42–50 (1991)
18. Ramesh, B., Ribbens, C.J., Varadarajan, S.: Is it time to rethink distributed shared memory systems? In: Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. pp. 212–219 (2011), iD: 1
19. Ramesh, B.: Samhita: Virtual shared memory for non-cache-coherent systems (2013)
20. Šinkarovs, A., Scholz, S.B., Bernecky, R., Douma, R., Grelck, C.: SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. Concurrency and Computation: Practice and Experience 26(4), 952–971 (2014)

Autorenverzeichnis