DHBW
Duale Hochschule
Baden-Württemberg

20. Kolloquium Programmiersprachen und Grundlagen der Programmierung KPS 2019 Tagungsband

Martin Plümicke, Fayez Abu Alia (Hrsg.)

Baiersbronn,
23.–25. September 2019

Martin Plümicke
Duale Hochschule Baden-Württemberg, Campus Horb
Florianstr. 15, 72160 Horb
pl@dhbw.de

Fayez Abu Alia
Duale Hochschule Baden-Württemberg, Campus Horb
Florianstr. 15, 72160 Horb
abualia@hb.dhbw-stuttgart.de

Vorwort

Das 20. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2019) setzt eine Reihe von Arbeitstagungen fort, die ursprünglich von den Forschungsgruppen der Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel) ins Leben gerufen wurde. Das erste Kolloquium fand 1980 in Tannenfelde im Naturpark Aukrug in der Nähe von Neumünster in Schleswig-Holstein statt. Die Tagungen finden seitdem in etwa zweijährigen Rhythmus statt. Aus den ursprünglich drei Arbeitsgruppen sind in der Zwischenzeit weitere Forschungsgruppen in ganz Deutschland und darüber hinaus hervorgegangen. Heute präsentiert sich das Kolloquium als offenes Forum für alle interessierten deutschsprachigen Wissenschaftler und bietet einen zwanglosen Rahmen zum Austausch neuer Ideen und Ergebnisse aus den Bereichen Entwurf und Implementierung von Programmiersprachen sowie Grundlagen und Methodik des Programmierens. Die inzwischen 39-jährige Tradition wird sichtbar in der Liste der bisherigen Tagungsorte und Veranstalter:

2017	Weimar	Uni Jena
2015	Pörtschach am Wörthersee	TU Wien
2013	Lutherstadt Wittenberg	Uni Halle-Wittenberg
2011	Schloss Raesfeld	Uni Münster
2009	Maria Taferl	TU Wien
2007	Timmendorfer Strand	Uni Lübeck
2005	Fischbachau	LMU München
2004	Freiburg-Munzingen	Uni Freiburg
2001	Rurberg in der Eifel	RWTH Aachen
1999	Kirchhundem-Heinsberg	FernUni Hagen
1997	Avendorf auf Fehmarn	Uni Kiel
1995	Alt-Reichenau	Uni Passau
1993	Garmisch-Partenkirchen	UniBw München
1992	Rothenberge bei Steinfurt	Uni Münster
1989	Hirschegg	Uni Augsburg
1987	Midlum auf Föhr	Uni Kiel
1985	Passau	Uni Passau
1982	Altenahr	RWTH Aachen
1980	Tannenfelde im Naturpark Aukrug	Uni Kiel

Das 20. Kolloquium Programmiersprachen und Grundlagen der Programmierung wird vom Studiengang Informatik des Campus Horb der DHBW Stuttgart organisiert. Wir freuen uns darauf, etwa 30 Teilnehmende vom 23. bis 25. September 2019 in Baiersbronn begrüßen zu dürfen. Besonders freuen wir uns, dass Gerhard Goos das Kolloquium wiederum mit einem Beitrag zur Geschichte der Informatik bereichern wird.

Der vorliegende Tagungsband ist im Shakerverlag in der Verlagsreihe Informatik erschienen. Die darin enthaltenen 22 Beiträge zeigen, zum Teil in Form von Kurzzusammenfassungen, die Breite der wissenschaftlichen Forschung zu Programmiersprachen und Grundlagen der Programmierung im deutschsprachigen Raum. Unser ausdrücklicher Dank gilt allen Autorinnen und Autoren für ihre Beträge, ohne die dieser Tagungsband natürlich nicht möglich wäre. Unser Dank gilt darüber hinaus dem Campus Horb der DHBW Stuttgart und den Mitarbeitenden des Hotels Sackmann in Baiersbronn. Ein besonderer Dank gilt der Firma Dr. Hornecker Softwareentwicklung und IT-Dienstleistungen in Freiburg, die das Kolloquium mit einer großzügigen Spende unterstützt hat.

Martin Plümicke
Fayez Abu Alia

Inhaltsverzeichnis

Minimal set of generated generics in Java-TX **1**
Fayez Abu Alia und Martin Plümicke

A Refactoring Tool for the Safe Parallelization of Array-Centric For-Loops **15**
Christoph Amrein und Luc Bläser

Sharing Nogoods between Portfolio Instances in Choco Solver **31**
Ilja Becker, Sven Löffler und Petra Hofstedt

Eigenschaften von Mustern zur Herleitung geordneter Attributgrammatiken aus Mustern und Paradigmen **37**
Christian Berg

Zustandsbasiertes Debugging mit abstrakten Zustandsmaschinen **53**
Sebastian Beyer und Wolf Zimmermann

Shape Inference from Memory Graphs **69**
Jan H. Boockmann und Gerald Lüttgen

Muli: Constraint-Logic Object-Oriented Programming mit freien Objekten **73**
Jan C. Dageförde und Herbert Kuchen

Integer Division by Multiplying with the Double-Width Reciprocal **75**
M. Anton Ertl

Proving Termination of Pointer Programs on Top of Graph-based Symbolic Execution **85**
Ira Justus Fesefeldt

Studiengang Informatik — die Anfänge **89**
Gerhard Goos

Compile-time Verification of Declarative Programs with SMT Solvers . . **101**
Michael Hanus

Parallele Programmierung mit MOSTflexiPL **107**
Christian Heinlein

Mining von BPMN-Prozessartefakten auf GitHub **111**
Thomas S. Heinze, Viktor Stefanko und Wolfram Amme

Path-Specific Program Transformation **121**
Jens Knoop

Ein Weg zum verteilten agentenbasierten Big Data Netzwerk im Logistikumfeld **123**
Marianne Mauch

Implementing a Mutation Testing Framework in Rust using Procedural Macros **127**
Samuel Pilz

Von Pizza zu Java–TX **129**
Martin Plümicke

Renaissance: Benchmarking Suite for Parallel Applications on the JVM **141**
Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger und Walter Binder

Improving Stability and Portability of Haskell IDE Engine: A Case Study **145**
Hannes Siebenhandl

Subsidiary Meta-Programming for Higher-Level Programming Concepts **149**
Daniel Speicher und Armin B. Cremers

A type system for javascript **167**
Andreas Stadelmeier

Selbstspezialisierende Parserkombinatoren – Experimente mit praktischer partieller Auswertung in konventionellen Programmiersprachen **171**
Baltasar Trancón y Widemann und Markus Lepper

Minimal set of generated generics in Java-TX

Fayez Abu Alia and Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
abualia@hb.dhbw-stuttgart.de, pl@dhbw.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 general idea of Java-TX is that the programmer can write typeless Java code and the compiler translate it with the inferred types to correct bytecode.
There are some interesting properties which follow from the Java-TX extensions. Global type inference leads to generated generics of methods and classes.
In this paper we present an algorithm which takes the remaining type variables of the type inference output and generates generics.
Therefore we have to divide generics of the class and the methods, respectively. There are some simplification to generate an minimal set of generics.

1 Introduction

Java-TX (Java Type eXtended) is an extension of Java in which a global type inference algorithm and real function types are added. Since the end of the nineties features from functional programming languages have been transferred to Java. Parametric polymorphism extended by wildcards, called generics, were transfered to Java 5.0 [GJSB05]. Higher-order functions and lambda expression were introduced in Java 8 [GJS+15]. Java 8 uses functional interfaces as target types of lambda expressions in contrast to real function types as in functional programming languages.
The powerful feature *type inference* from functional programming languages is incorporated into Java, as into other object-oriented languages, i.e. only in a restricted way called local type inference [PT00,OZZ01]. Local type inference allows certain type annotations to be omitted. For instance, it is often not necessary to specify the type of a variable. Type parameters of classes in the `new`–statement can be left out. Return types of methods can often also be omitted. Local type inference is at its most pronounced in Scala [OAC+17]. In Java 10 an extention of local type inference is introduced (JEP 286: Local-Variable Type Inference [Goe16]), where types of local variables can be replaced by the keyword `var` and inferred automatically during the compilation. In contrast to global type inference, local type inference allows types of recursive methods and lambda expressions not to be omitted.

The Java-TX project contributes to the design of object-oriented languages by developing global type inference algorithms for Java-like languages. In [Plü07] we presented a type inference algorithm for a core Java 5.0 including wildcards. We went on to give a type inference algorithms for a core of Java 8 [Plü15] and for real function types [PS17].
The paper is structured as follows: In the next section we describe the generation of generics for methods and classes. Afterwards, we describe the defined algorithm to determine the smallest set of (bounded) type variables that have to be generated. Finally, we end with a conclusion.

2 Generated generics

The basic functionality of our type inference algorithm is that first all omitted types are assumed as a type placeholder. During the algorithm the type placeholders are replaced by the inferred types step by step. Remaining type placeholders are generic types of the corresponding methods and classes, respectively.
Before generating bytecode, we walk on the abstract syntax tree (`AST`) to collect all remaining type placeholders and all constraints of this type placeholders. We store the type placeholders of the class in the set `TPHS_CLASS` and for each method the method signature and its type placeholders in the set `METHODS_AND_TPHS`. The type placeholders of the class are the type placeholders of the fields and the constructor parameters. Furthermore we store for the class all constraints of all type placeholders such as `TPH1 extends TPH2` in the set `CONSTRAINTS`, where `TPH1` and `TPH2` are type placeholders and `extends` represents the Java subtyping relation.

Let us consider an example.

Example 1. The class `Id` is typeable in Java.

```
class Generics {
    a;
    id(b) { return b; }
    setA(x) {
        a = x;
        return a;
    }
    m(x,y) { x = id(y); }
}
```

The type inference algorithm infers

```
class Generics {
    T1 a;
    T2 id(T3 b) { return b; }
    T4 setA(T5 x) {
```

```
        a = x;
        return a;
    }
    void m(T6 x,T7 y) { x = id(y); }
}
```

with the constraints `T3 extends T2`, `T5 extends T1`, `T1 extends T4`, `T7 extends T3` and `T2 extends T6`.
The table below shows the informations that we use for generating generics:

Class/method	Type placeholders	Constraints
Class	T1	T1 extends T4
id	T2, T3	T3 extends T2, T2 extends T6
setA	T4, T5	T5 extends T1
m	T6, T7	T7 extends T3, T3 extends T2, T2 extends T6

We generate for each type placeholder a (bounded) type variable. The result of generated generics:

```
class Generics <T1 extends T4, T4>{
    T1 a;
    <T3 extends T2, T2 extends T6, T6> T2 id(T3 b) {
        return b;
    }
    <T5 extends T1> T4 setA(T5 x) {
        a = x;
        return a;
    }
    <T7 extends T3, T3 extends T2, T2 extends T6, T6>
       void m(T6 x,T7 y) {
        x = id(y);
    }
}
```

In the result of the generated generics we can see that in methods `id` and `m` the number of the generated type variable is greater than the number of the used type variables. If we consider the generated generics more accurately, we recognize that in both methods there are unnecessary type variables which can be removed:

- For id:
  `T3 extends T2 extends T6` can be replaced with `T3 extends T2`, because `T6` is not used in the method and if we remove it we still have a correct code.
- For m:
  `T7 extends T3 extends T2 extends T6` can be replaced with `T7 extends T6`.

After reducing the number of the generated type variables:

```
class Generics <T1 extends T4, T4>{
    T1 a;

    <T3 extends T2, T2> T2 id(T3 b) {
        return b;
    }
    <T5 extends T1> T4 setA(T5 x) {
        a = x;
        return a;
    }
    <T7 extends T6, T6> void m(T6 x,T7 y) {
        x = id(y);
    }
}
```

This is a correct Java program without any type error and it is equivalent to the program before reducing the number of type variables.

We define an algorithm to determine the smallest set of the type variables with their bounds which have to be generated.

3 The algorithm

This section contains the description of the defined algorithm that determine the smallest set of the type variables with their bounds which have to be generated.

The algorithm consists of two parts. One is simplification of constraints, the other is generation of the smallest set of constraints, which will be used for generating generics. In the simplification of constraints all constraints with the same left-hand sides, such as `T extends R` and `T extends S`, will be replaced by one constraint like `T extends X`, where `X = R` and `X = S`, because in Java a class can only extend one parent class. Furthermore, in this part all cycles, if they exist, will be removed. In the second part of the algorithm we use the simplified constraints to create the smallest set of constraints for the class and for each method. The smallest set of constraints for the class contains only the constraints, which consist of the class type placeholders. Each constraint in the generated set of constraints for a method consists of type placeholders of this method.
The two parts of the algorithm will be described in more details.

PART 1: Simplify constraints

Input:

– Set of all type placeholders of the class: `TPHS_CLASS`.

- Set of all methods of the class and for each method the type placeholders of its return type and its parameters: `METHODS_AND_TPHS`.
- Set of all constraints of the class: `CONSTRAINTS`.

Step 1: Eliminate infima.
If there are constraints in `CONSTRAINTS` that have the same left-hand sides such as `C` = {`(T extends R)`, `(T extends S)`, ... } then

- Create a new name (e.g. `X`) and create a new constraint `(T extends X)`.
- Add the new constraint `(T extends X)` to `CONSTRAINTS`.
- Remove all constraints of `C` from `CONSTRAINTS`.
- Substitute the names on the right-hand side of all constraints in `C` with `X` in `TPHS_CLASS` and `METHODS_AND_TPHS`.

Repeat this step until no such constraints exist.

Step 2: Remove cycles.
If there is any cycle (e.g. `T extends K extends G extends ... extends T`) then

- Substitute all type placeholders of the cycle with T in `CONSTRAINTS`, `TPHS_CLASS` and `METHODS_AND_TPHS`.
- Remove all constraints that build the cycle from `CONSTRAINTS`.

After executing part 1 the set `CONSTRAINTS` only contains the simplified constraints.

PART 2: Generate constraints for generating generics

Input:

- All type placeholders of the class `TPHS_CLASS`.
- All methods of the class and for each method the type placeholders of its return type and its parameters `METHODS_AND_TPHS`.
- The set of simplified constraints `CONSTRAINTS`.

Output:

- `CLASS_CONSTRAINTS` - the smallest set of the type variables with their bounds of the class.
- `METHODS_AND_CONSTRAINTS` - a map contains the names of methods and the associated smallest set of the type variables with their bounds.

Step 1: Determine class type parameters.
Get all type placeholders `T` from `METHODS_AND_TPHS`, where `(K extends T)` and `K` $\in$ `TPHS_CLASS`. Remove these type placeholders from `METHODS_AND_TPHS` and add them to `TPHS_CLASS`. Repeat this step until there are no more type placeholders `T` $\in$ `METHODS_AND_TPHS` with `(K extends T)` and `K` $\in$ `TPHS_CLASS`.

Step 2: Generate class type constraints.

– Mark all type placeholders from TPHS_CLASS as not visited.
– For each type placeholder T ∈ TPHS_CLASS if T not visited:
  a. C = Create_Constraint(T, TPHS_CLASS, {} ,CONSTRAINTS)[1].
  b. Put C in CLASS_CONSTRAINTS.

Step 3: Generate unbounded class type parameters.
For each type placeholder T ∈ TPHS_CLASS if there is no constraint in CLASS_CONSTRAINTS that has T on its left-hand side, then T is unbounded type variable. Therefore, create the constraint C = (T extends java.lang.Object) and put it in CLASS_CONSTRAINTS.

Step 4: Generate type constraints for methods.
For each method M of the class get all type placeholders of M (TPH_METHOD):

– Mark all type placeholders as not visited and define CONSTRAINTS_M = {}.
– For each type placeholder T ∈ TPHS_METHOD if T not visited:
  a. C = Create_Constraint(T, TPHS_CLASS, TPHS_METHOD, CONSTRAINTS).
  b. Put C in CONSTRAINTS_M.
– For each type placeholder T ∈ TPHS_METHOD if there is no constraint in CONSTRAINTS_M that has T on its left-hand side, then T is unbounded type variable. Therefore, create the constraint C = (T extends java.lang.Object) and put it in CONSTRAINTS_M.
– Add the pair (M,CONSTRAINTS_M) to METHODS_AND_CONSTRAINTS.

Below we describe how a constraint for a given type placeholder will be generated. If the type placeholder T is an unbounded type variable, then the constraint T extends java.lang.Object will be generated. That means there is no relation between T and the other type placeholders. If T is a bounded type variable, then the constraint T extends K will be generated, where K is the upper bound.

Create_Constraint: Generate a constraint for generating generics

Input:

– Type placeholder T, for which the constraint will be generated.
– The set of type placeholders of the class: TPHS_CLASS.
– The set of type placeholders of a certain method: TPHS_METHOD. If we generate a constraint for a type placeholder of the class, TPHS_METHOD is empty.
– The set of simplified constraints: CONSTRAINTS.

Output: The generated constraint.

Step 1: Generate transitive closure upward.

a. Create an empty list L
b. Mark T as visited and add it to L.

[1] For function Create_Constraint see below.

c. Let `X` be the last element of `L`. If `X` $\in$ `TPHS_CLASS` $\cup$ `TPHS_METHOD` and size of `L` > 1, then go to **step 3.** Otherwise, if there is any constraint (`X extends K`), then add `K` at the end of `L` and repeat **c.**

Step 2: Generate transitive closure downward.
Let E_{first} be the first element of `L`:

- If there is any constraint (`X extends` E_{first}) and `X` is not visited, then add `X` at the beginning of `L` and mark `X` as visited. Otherwise, go to **Step 3.**
- If `X` $\notin$ `TPHS_CLASS` $\cup$ `TPHS_METHOD`, then repeat **Step 2.**

Step 3: Transform transitive closure to constraints.
Remove all elements `E` at the beginning and end of `L`, which are not in `TPHS_CLASS` $\cup$ `TPHS_METHOD`. Let E_{first} be the first element of `L`:

- If the size of `L` $= 1$, then E_{first} is an unbounded type variable. We create for it the constraint `C` $=$ (E_{first} `extends java.lang.Object`) and return `C`.
- If the size of `L` > 1, then E_{first} is a bounded type variable and E_{last} is the upper bound of E_{first}, where E_{last} the last element of `L`. We create the constraint `C` $=$(E_{first} `extends` E_{last}) and return `C`.

Let us consider the following example to see how the algorithm works.

Example 2. We perform the algorithm for the class `TPHsAndGenerics`.

```
class TPHsAndGenerics {
    id = x -> x;

    id2 (x) {
        return id.apply(x);
    }

    m(a,b){
        var c = m2(a,b);
        return a;
    }

    m2(a,b){
        return b;
    }
}
```

The type inference algorithm infers

```
class TPHsAndGenerics {
    Fun1<T1,T2> id = x -> x;

    T3 id2 (T4 x) {
        return id.apply(x);
```

```
    }

    T5 m(T6 a, T7 b){
        m2(a,b);
        return a;
    }

    T8 m2(T9 a, T10 b){
        return b;
    }
}
```

with the constraints CONSTRAINTS = {(T1 extends T2), (T2 extends T3), (T4 extends T1), (T6 extends T5), (T6 extends T9), (T7 extends T10),(T10 extends T8)}.
Table 1 shows the informations that we use for generating generics. The first row in the table 1 represents the TPHS_CLASS and the rows below of it represent the METHODS_AND_TPHS. These informations are the inputs of the algorithm.

Class/method	Type placeholders
Class (TPHsAndGenerics)	T1, T2
id2	T3, T4
m	T5, T6, T7
m2	T8, T9, T10

Table 1. Type placeholders that will be used for generating generics

We start with part 1 of the algorithm (simplify constraints):

Step 1: In the set of constraints there are two constraints with the same left-hand sides C = {(T6 extends T5), (T6 extends T9)}. We create the constraint (T6 extends R) and put it in CONSTRAINTS. Then we remove the both constraints of C from CONSTRAINTS and replace T5 and T9 by R. Below, we can see the result of this step.

CONSTRAINTS = {(T1 extends T2), (T2 extends T3), (T4 extends T1), (T6 extends R), (T7 extends T10),(T10 extends T8)}.

Class/method	Type placeholders
Class (TPHsAndGenerics)	T1, T2
id2	T3, T4
m	R, T6, T7
m2	T8, R, T10

In CONSTRAINTS there are no more constraints with the same left-hand sides. Therefore, we go to the next step.
Step 2: There are no constraints in CONSTRAINTS that build a cycle.

We are finish with the first part of the algorithm. Now we perform the second part:

Step 1: Because of the constraint (T2 extends T3), we remove T3 from the type placeholders of the method id2 and add it to the type placeholders of the class:

Class/method	Type placeholders
Class (TPHsAndGenerics)	T1, T2, T3
id2	T4
m	R, T6, T7
m2	T8, R, T10

Step 2: We generate a type constraint for each type placeholder in TPHS_CLASS:

- T1 is not visited:
  a. C = Create_Constraint(T1, TPHS_CLASS, {}, CONSTRAINTS).
  Create_Constraint(T1, TPHS_CLASS, {}, CONSTRAINTS):
  - **Step 1.**
    a. L = { }
    b. We mark T1 as visited and add it to L: L = {T1}.
    c. Because of the constraint (T1 extends T2), we add T2 at the end of L: L = {T1→T2}.
    c. T2 ∈ TPHS_CLASS and size of L > 1. Therefore, we go to **step 3.**
  - **Step 3.** The size of L > 1 ⇒ we create the constraint C = (T1 extends T2).

  b. Add C to CLASS_CONSTRAINTS ⇒ CLASS_CONSTRAINTS = {(T1 extends T2)}

- T2 is not visited:
  a. C = Create_Constraint(T2, TPHS_CLASS, {}, CONSTRAINTS).
  Create_Constraint(T2, TPHS_CLASS, {}, CONSTRAINTS):
  - **Step 1.**
    a. L = { }
    b. We mark T2 as visited and add it to L: L = {T2}.
    c. Because of the constraint (T2 extends T3), we add T3 at the end of L: L = {T2→T3}.
    c. T3 ∈ TPHS_CLASS and size of L > 1. Therefore, we go to **step 3.**
  - **Step 3.** The size of L > 1 ⇒ we create the constraint C = (T2 extends T3).

b. Add C to CLASS_CONSTRAINTS ⇒ CLASS_CONSTRAINTS = {(T1 extends T2),(T2 extends T3)}

– T3 is not visited:
  a. C = Create_Constraint(T3, TPHS_CLASS, {}, CONSTRAINTS).
  Create_Constraint(T2, TPHS_CLASS, {},CONSTRAINTS):
    - **Step 1.**
      a. L = { }
      b. We mark T3 as visited and add it to L: L = {T3}.
      c. There is no constraint ∈ CONSTRAINTS that has T3 on its left-hand side. We go to **step 2.**
    - **Step 2.** We found the constraint (T2 extends T3) but T2 is already visited. Therefore, we do nothing here and go to **Step 3.**
    - **Step 3.** The size of L = 1 ⇒ we create the constraint C = (T3 extends java.lang.Object).
  b. Add C to CLASS_CONSTRAINTS ⇒ CLASS_CONSTRAINTS = {(T1 extends T2),(T2 extends T3),(T3 extends java.lang.Object)}

Step 3: Each type placeholder T ∈ TPHS_CLASS is located on the left-hand side of a constraint C ∈ CLASS_CONSTRAINTS ⇒ Do nothing and continue with **step 4.**
Step 4: We visit all methods of the class to generate constraints for their type placeholders:

– Method **id2** ⇒ TPHS_id2 = {T4}. We Mark all type placeholders as not visited and define CONSTRAINTS_id2 = {}.
  - T4 is not visited:
    a. C = Create_Constraint(T4, TPHS_CLASS, TPHS_id2, CONSTRAINTS).
    Create_Constraint(T4, TPHS_CLASS, TPHS_id2, CONSTRAINTS):
      * **Step 1.**
        a. L = { }
        b. We mark T4 as visited and add it to L: L = {T4}.
        c. Because of the constraint (T4 extends T1), we add T1 at the end of L: L = {T4→T1}.
        c. T1 ∈ TPHS_CLASS and size of L > 1. Therefore, we go to **step 3.**
      * **Step 3.** The size of L > 1 ⇒ we create the constraint C = (T4 extends T1).
    b. Add C to CONSTRAINTS_id2 ⇒ CONSTRAINTS_id2 = {(T4 extends T1)}

– T4 is on the left-hand side of (T4 extends T1, so continue with the next step.
– Add (id2,CONSTRAINTS_id2) to METHODS_AND_CONSTRAINTS.
– Method **m** ⇒ TPHS_m = {R,T6,T7}. We mark all type placeholders of as not visited and define CONSTRAINTS_m = {}.

- R is not visited:
  a. C = Create_Constraint(R, TPHS_CLASS, TPHS_m, CONSTRAINTS).
  Create_Constraint(R, TPHS_CLASS, TPHS_m, CONSTRAINTS):
     * **Step 1.**
       a. L = { }
       b. We mark R as visited and add it to L: L = {R}.
       c. There is no constraint ∈ CONSTRAINTS that has R on its left-hand side. We go to **step 2.**
     * **Step 2.** We found the constraint (T6 extends R) and T6 is not visited. We mark T6 as visited and add it to L: L = {T6→R}. Because T6 ∈ TPHS_m, we go to **step 3.**
     * **Step 3.** The size of L > 1 ⇒ we create the constraint C = (T6 extends R).
  b. Add C to CONSTRAINTS_m ⇒ CONSTRAINTS_m = {(T6 extends R)}

- T6 is visited. Therefore, we do nothing and continue with the next type placeholder.
- T7 is not visited:
  a. C = Create_Constraint(T7, TPHS_CLASS, TPHS_m, CONSTRAINTS).
  Create_Constraint(T7, TPHS_CLASS, TPHS_m, CONSTRAINTS):
     * **Step 1.**
       a. L = { }
       b. We mark T7 as visited and add it to L: L = {T7}.
       c. Because of the constraint (T7 extends T10), we add T10 at the end of L: L = {T7→T10}.
       c. T10 ∉ TPHS_CLASS ∪ TPHS_m and there is the constraint (T10 extends T8). Therefore, we add T8 at the end of L: L = {T7→T10→T8}.
       c. T8 ∉ TPHS_CLASS ∪ TPHS_m and There is no constraint ∈ CONSTRAINTS that has T8 on its left-hand side, so we continue with **step 2.**
     * **Step 2.** We did not find any constraint that has T7 on its left-hand side.Therefore, we go to **step 3.**
     * **Step 3.** We remove T10 and T8 from L ⇒ L = {T7}. Now the size of L = 1 ⇒ we create the constraint C = (T7 extends java.lang.Object).
  b. Add C to CONSTRAINTS_m ⇒ CONSTRAINTS_m = {(T6 extends R),(T7 extends java.lang.Object)}

– We create the constraint C = (R extends java.lang.Object) and add to CONSTRAINTS_m.
– Add (m,CONSTRAINTS_m) to METHODS_AND_CONSTRAINTS.
– Method **m2** ⇒ TPHS_m2 = {T8,R,T10}. We mark all type placeholders of as not visited and define CONSTRAINTS_m2 = {}.
  - T8 is not visited:
    a. C = Create_Constraint(T8, TPHS_CLASS, TPHS_m2, CONSTRAINTS).
    Create_Constraint(T8, TPHS_CLASS, TPHS_m2, CONSTRAINTS):

* **Step 1.**
  a. `L = { }`
  b. We mark `T8` as visited and add it to `L`: `L = {T8}`.
  c. There is no constraint ∈ `CONSTRAINTS` that has `T8` on its left-hand side. We go to **step 2.**
* **Step 2.** We found the constraint `(T10 extends T8)` and `T10` is not visited. We mark `T10` as visited and add it to `L`: `L = {T10→T8}`. Because `T10` ∈ `TPHS_m2`, we go to **step 3.**
* **Step 3.** The size of `L` > 1 ⇒ we create the constraint `C = (T10 extends T8)`.

b. Add `C` to `CONSTRAINTS_m2` ⇒ `CONSTRAINTS_m2 = {(T10 extends T8)}`

- `R` is not visited:

a. `C = Create_Constraint(R, TPHS_CLASS, TPHS_m2, CONSTRAINTS)`. `Create_Constraint(R, TPHS_CLASS, TPHS_m2, CONSTRAINTS)`:

* **Step 1.**
  a. `L = { }`
  b. We mark `R` as visited and add it to `L`: `L = {R}`.
  c. There is no constraint ∈ `CONSTRAINTS` that has `R` on its left-hand side. We go to **step 2.**
* **Step 2.** We found the constraint `(T6 extends R)` and `T6` is not visited. We mark `T6` as visited and add it to `L`: `L = {T6→R}`. Because `T6` ∉ `TPHS_m`, we repeat **step 2.**
* **Step 2.** There is no constraint ∈ `CONSTRAINTS` that has `T6` on its right-hand side and `T6` ∉ `TPHS_CLASS` ∪ `TPHS_m2`. Therefore, continue with the next step.
* **Step 3.** We remove `T6` from `L` ⇒ `L = {R}`. Now the size of `L` = 1 ⇒ we create the constraint `C = (R extends java.lang.Object)`.

b. Add `C` to `CONSTRAINTS_m2` ⇒ `CONSTRAINTS_m2 = {(T10 extends T8),(R extends java.lang.Object)}`

- `T10` is visited. Therefore, we do nothing.

– We create the constraint `C = (T8 extends java.lang.Object)` and add to `CONSTRAINTS_m2`.
– Add (`m2`,`CONSTRAINTS_m2`) to `METHODS_AND_CONSTRAINTS`.

The output of the algorithm:

– The smallest set of the type variables with their bounds of the class {`(T1 extends T2),(T2 extends T3),(T3 extends java.lang.Object)`}.
– A map contains the names of methods and the associated smallest set of the type variables with their bounds

The generated bytecode for the class `TPHsAndGenerics` corresponds to the following Java class:

Method name	Type variables and their bounds
id2	{(T4 extends T1)}
m	{(T6 extends R),(R extends java.lang.Object),(T7 extends java.lang.Object)}
m2	{(T10 extends T8),(T8 extends java.lang.Object),(R extends java.lang.Object)}

```
class TPHsAndGenerics<T1 extends T2, T2 extends T3,
    T3> {
     Fun1<T1,T2> id = x -> x;

     <T4 extends T1> T3 id2 (T4 x) {
         return id.apply(x);
     }

     <T6 extends R, R, T7> R m(T6 a, T7 b){
         m2(a,b);
         return a;
     }

     <T10 extends T8, T8, R> T8 m2(R a, T10 b){
         return b;
     }
}
```

We can see in the class above that for each method of the class only type variables are generated that are used in the method and there are no unnecessary defined type variables.

4 Conclusion

In this paper we described the generation of generics for the type placeholders, which can not be replaced by the inferred types. Then we presented an algorithm, which determine the smallest set of type variables with their bounds using the transitive relationship between the type placeholders.

References

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Language Specification*. The Java series. Java SE 8 edition, 2015.

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

[Goe16] Brian Goetz. Jep 286: Local-variable type inference, 2016.

[OAC+17] Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, St?hane Micheloud, Nikolay Mihaylov, Adriaan Moors,

Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2017. Version 2.12.

[OZZ01] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. *Proc. 28th ACM Symposium on Principles of Programming Languages*, 36(3):41–53, 2001.

[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.

[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.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

A Refactoring Tool for the Safe Parallelization of Array-Centric For-Loops

Christoph Amrein and Luc Bläser

HSR Hochschule für Technik Rapperswil, Switzerland
christoph.amrein@hsr.ch, luc.blaeser@hsr.ch

Abstract. We present a software refactoring tool for loop parallelization. By using a light-weight static analysis, the tool conservatively detects sequential for-loops with mutually independent loop steps that can be safely transformed to corresponding parallel for-loops. We implemented this tool for C# as a Visual Studio IDE plugin and applied it to several open-source projects. Our evaluation shows that the analysis is fast and effective in detecting parallelizable array-centric for-loops.

Keywords: Concurrency · Parallelization · Refactoring · For-Loops · Loop-Carried Dependence · C# · Dataflow Analysis

1 Introduction

Modern software is still unnecessarily sequentially programmed, even if today's programming languages feature light-weighted parallelization constructs and efficient parallelization libraries, such as for example the task parallel library (TPL) [20] in .NET with its parallel invocation or parallel loop statements. A reason for this is that programmers have to explicitly opt-in for parallelization by selecting the corresponding parallel program idioms, instead of having a system or IDE tool that automatically assists in parallelizing the right code fragments. For this reason, various research has been conducted on the topic of automatic parallelization, at the level of the compiler [1, 5, 4, 23, 28] or the runtime system [26, 27].

In this work, we focus on a specific case of automatically inferred parallelization, namely *for-loop parallelization*. Our goal to is develop an IDE tool that interactively assists programmers in identifying for-loops that can be safely parallelized, i.e., without altering the functional behavior of the loop, except its execution speed. The tool eventually transforms such for-loops to their corresponding parallel versions. We concentrate on for-loops that have mutually independent loop steps and are therefore straightforward to be replaced by their parallel counterpart, without introducing synchronization in the parallel loop bodies.

To identify parallelizable loops, we focus ourselves on independent loops. This is also known as proving the absence of loop-carried dependencies [2]. A loop-carried dependency is given if an instruction of an iteration depends on an

instruction of another iteration. Conversely, loop iteration steps are independent if they do not access the same memory locations except for read-only accesses, and if they do not perform any side effects via external function calls, such as I/O. In many programming languages, including C#, memory locations are defined at the granularity of variables or array elements. Therefore, loops are guaranteed to be independent if the write accesses per loop step target distinct memory locations, and if none of the loop steps calls side-effecting functions.

A particular challenge in the analysis of loop independence is the verification that there is no array access intersection when at least one element of the array is written in the loop, i.e., each iteration accesses a distinct set of elements of the array. For example, consider the code sketched in Listing 1. This example computes the sum of all entries of the array a up to the position i and writes it to the position i. A loop-carried dependency occurs in the line 6 because it depends on the current iteration i and the previous iteration $i-1$. This loop hence has dependent iterations steps, and will not be parallelized by our tool.

```
1  int[] a = new int[] {
2    1, 2, 3, 4, 5, 6
3  };
4  int m = a.Length;
5  for(int i=1; i < m; i++) {
6    a[i] = a[i - 1] + a[i];
7  }
```

Listing 1. Example of a dependent for-loop that will not be parallelized.

```
1  int[] a = new int[] {
2    1, 2, 3, 4, 5, 6
3  };
4  int m = a.Length;
5  for(int i=0; i < m; i++) {
6    a[i] = a[i] * a[i];
7  }
```

Listing 2. Example of an independent loop that can be parallelized.

In contrast, Listing 2 sketches an independent and thus parallelizable loop. This for-loop squares the value at the array position i and stores it again at the same position. All accesses in the line 6 access a distinct region of the array a per iteration, such that this loop is free of array access intersections.

There exist well-known static loop dependency analysis methods, such as the polyhedral model [1, 5], or the GCD dependence test [4, 23, 28]. While these methods can detect specific complex cases, our motivation was to design a simple alternative algorithm, that is both fast and effective in practice. This is particularly important in an IDE refactoring tool that should respond fast to code changes. Our method also supports parallelization of array-centric loops with non-linear array index functions. For this purpose, we designed a light-weight dataflow analysis that infers potential array access intersections, with a relatively limited, cheap context information. This analysis has been integrated into our parallel refactoring tool for C# that works as a plugin in Visual Studio IDE. It

highlights for-loops that can be safely parallelized, and on the consent of the developer, parallelizes these for-loops, by using the .NET TPL's `Parallel.For`.

In summary, this paper makes the following contributions:

- Description of a new dataflow analysis to determine the absence of array access intersections, that experimentally proves to be fast and effective for array-centric for-loops.
- Report of the implementation and evaluation of a practical refactoring tool for loop parallelization in C# and Visual Studio based on this analysis.

The remainder of this paper is structured as follows: Section 2 presents the dataflow analysis method. Section 3 descibes the implementaton of our refactoring tool based on this analysis. Section 4 reports on the experimental evaluation of the analysis and the refactoring tool. Section 5 discusses related work. Section 6 finally concludes this paper.

2 Analysis

Instead of applying potentially expensive computations, the analysis uses a set of simple transfer rules on the static single assignment (SSA) form. These transfer rules are designed to provide sufficient information to decide whether an array index expression is an injective function. In other words, it helps deciding if each iteration in the loop computes unique array indices. One essential property of the analyzed array index expressions is that they are dependent on the loop index. For instance, if the function $f(x)$ is injective, using the loop index as function argument yields a unique value for each iteration. Therefore, an array access such as $a[f(i)]$, with i representing the loop index, is independent in each loop step. Consequently, the identity function $f(x) = x$ is the simplest function that satisfies this property.

This section outlines our dataflow-based loop dependency analysis: Section 2.1 introduces an example that is used throughout the sections to explain the different steps of the analysis. Section 2.2 specifies requirements for the for-loops that are covered by the analysis. Section 2.3 introduces our notation to encode analysis properties. Section 2.4 summarizes the property inference used in the dataflow analysis. Section 2.5 specifies the dataflow analysis that computes the properties for loop code. Section 2.6 explains how the dataflow analysis output allows determining potential array access intersections. Section 2.7 finally combines the computed information to determine whether the loop is independent and can be safely parallelized. Section 2.8 describes further static analyses that can increase the precision in our loop dependency analysis.

2.1 Example

The analysis procedure is introduced with the help of the example sketched in Listing 3. The illustrated code computes the factorial of every odd array index and stores the factorial at this position. The outer loop declared in line 5 iterates

```
1  long[] f = new long[] {
2    15, 15, 18, 18, 20, 20
3  };
4  int m = f.Length / 2;
5  for(int i = 0; i < m; i++) {
6    int o = 1;
7    int p = o + i * 2;
8    long s = f[p];
9    long c = 1;
10
11   for(int j = 1; j <= s; j++) {
12     c = c * j;
13   }
14
15   f[p] = c;
16 }
```

Listing 3. A loop which computes the factorial of every odd array index.

```
1   o1 ← 1
2   p1 ← o1 + i * 2
3   s1 ← f[p1]
4   c1 ← 1
5   j1 ← 1
6   label 1
7   if φ(j1, j2) <= s1 goto 2
8   goto 3
9   label 2
10  c2 ← φ(c1, c2) * φ(j1, j2)
11  j2 ← φ(j1, j2) + 1
12  goto 1
13  label 3
14  f[p1] ← φ(c1, c2)
```

Listing 4. Body of the loop that computes the factorials in the SSA form.

over every second element, while the inner loop in line 11 computes the factorial of the current element.

Listing 4 shows the body of the loop that iterates over the array elements in the SSA form. The loop index i and the array f have no value numbering since their write accesses and definitions only occur outside of the body. Instead of denoting the SSA form as a graph, we here sketch the code with labels and goto instructions. The line numbers are later used as labels to refer to the specific basic blocks.

2.2 Prerequisites

For our analysis, we focus on for-loops that have the form as sketched in Listing 5. Conceptually, other loop patterns, e.g. reverse or multi-step loops, could be handled as well, by transforming them to our required form. The `type` of the loop index has to be an integer, such as `int` or `long`. The identifier `low` defines the lower bound and `up` the upper bound of the loop. The implementation also supports the logical operator `<=` to define the upper bound as well as `i+=1` for the single step increment. An iteration may start with a negative value and end with a positive one. We currently concentrate on this specific template of single-step forward for-loop as this corresponds to the only supported variant of the `Parallel.For` program construct of .NET. Moreover, we require that the analyzed sequential for-loops are correct, e.g., have no integer overflows, no concurrency issues, etc. inside the loop bodies. We need this assumption, as we

```
1  for(type i = low; i < up; i++)
```

Listing 5. Definition of the for-loop structure.

do not prove these elementary correctness conditions and could otherwise effect subsequent errors through the parallelization.

2.3 Notation

This section introduces some elementary notation that we use in our analysis and it is outlined in Table 1. We generally denote sets with capital letters. The identifier ℓ stands for the label of a particular basic block of the loop body in the SSA form and s_ℓ the basic block with this label. The set S represents the current state of the analysis. This set contains tuples of the form $(x\ p)$, stating that the variable x has the property p.

Table 1. Analysis notation

Notation	Description
ℓ	The label ℓ of the basic block.
s_ℓ	The basic block with the label ℓ.
S	The current analysis state.
x	The variable x of the SSA form.
p	The property p.
$(x\ p)$	The variable x has the property p.

Table 2. Property notation

Notation	Description
$x\mid$	x is loop-dependent.
$x\nmid$	x is not loop-dependent.
$x = 0$	x is zero.
$x \not\simeq 0$	x is at most once zero.
$x = 1$	x is one.
$x+$	x is positive.
$x-$	x is negative.

Table 2 introduces the set of properties a variable may have. A variable can have multiple properties associated with it. Only a few properties are mutually exclusive, e.g., a positive variable cannot be negative at the same time. Moreover, the absence of a property does not mean its negation, it only means that the property has not been proved by the conservative analysis. The primary property is the loop dependence denoted as the vertical bar[1] $x\mid$. This expresses that the variable x is either the loop index itself or a variable computed with an injective function making use of the loop index. Therefore, these variables have unique values for each iteration. This property eventually helps to prove the absence of array access intersections, along with other rules that we describe in subsequent sections. The counterpart $x\nmid$ expresses that the variable x will have the same value for all iterations. The property $x = 0$ states that the value is exactly 0,

[1] The idea for the bar comes from the moving value of the loop index. It moves from one bound to the other.

whereas $x \not\simeq 0$ specifies that the value may only be zero for a single iteration. This may be the case if the variable x is loop-dependent. $x = 1$ means that x is exactly 1. Property $x+$ denotes that the value of x is positive or increases in each loop step, if x is loop-dependent. Analogously, $x-$ has the opposite characteristics as $x+$, i.e., represents negative values or a loop-decreasing value.

2.4 Property Inference

This section introduces the rules to derive property sets from expressions and statements in the program code.

Loop Index The loop index has a set of properties, comprising the loop-dependence property | and the at-most-once-zero property $\not\simeq 0$. As we currently only address increasing loop directions, the additional + property is also associated. If we support decreasing loop directions in future, we would have to associate − for them instead. As for the introductive example of Section 2.1, the loop index has the property set of (1).

$$Prop(i) = \{|, \not\simeq 0, +\} \tag{1}$$

Constants The properties derived from integer constants are self-explanatory, thus only listed in Table 3 for brevity.

Table 3. Properties generated by constants

Constant	Generated Properties
0	$\{\nmid, = 0\}$
1	$\{\nmid, \not\simeq 0, = 1, +\}$
$2, 3, 4, ...$	$\{\nmid, \not\simeq 0, +\}$
$-1, -2, -3, ...$	$\{\nmid, \not\simeq 0, -\}$

Phi-Functions The ϕ-functions conservatively yield the empty property set. As ϕ represents multiple potential values and the analysis aims to identify injective index expressions, we cannot take the intersection. For example, if $x_1 = 2$ and $x_2 = 3$, the intersection of the properties would be $\{\nmid, \not\simeq 0, +\}$. However, this information would wrongly classify an assignment like $a[i + \phi(x_1, x_2)] \leftarrow i$ as not intersecting between the iterations which is not necessarily the case. Therefore, we stick to the conservative rule of (P-PHI).

$$Prop(\phi(x_1, x_2, ..., x_n)) = \emptyset \tag{P-PHI}$$

Variables The dataflow analysis keeps track of all variables and the associated properties and stores these tuples in the analysis state S. This state is queried to retrieve all state tuples with the desired variable to get the associated properties as specified by (P-VAR).

$$Prop(x) = \{p \mid (x\ p) \in S\} \tag{P-VAR}$$

For example, consider the analysis state sketched in (2).

$$S = \{i\ |, i \not\simeq 0, i+, o_1\ \nmid, o_1 \not\simeq 0, o_1 = 1, o_1+\} \tag{2}$$

(3) and (4) illustrate the properties of the variables i and o_1, respectively.

$$Prop(i) = \{|, \not\simeq 0, +\} \tag{3}$$
$$Prop(o_1) = \{\nmid, \not\simeq 0, = 1, +\} \tag{4}$$

Binary Expressions Tables 4 to 8 list the property inference rules for binary expressions. The columns *Left* and *Right* express the conditions that have to be satisfied when deriving the properties. More specifically, both columns specify a set of properties that have to be associated with the corresponding operand as sketched in (P-BINEXP).

$$Prop(e_1\ \text{op}\ e_2) = \bigcup_{r \in Rows(\text{op})} \{Result_r \mid Left_r \subseteq Prop(e_1) \wedge Right_r \subseteq Prop(e_2)\} \tag{P-BINEXP}$$

For example, if the column *Right* has the set $\{|, +\}$, it requires that the properties $|$ and $+$ are associated with the right operand. If this condition is not satisfied, the properties of the current row cannot be derived. The column *Result* denotes which properties are derived if the associated conditions are fulfilled. The rows captioned with *Copy* identify opportunities where the properties can be safely copied from the specified operand. In summary, each row of the rule table has to be checked whether the corresponding result property of the row can be derived. Therefore, it is possible to derive multiple properties for a single binary expression that are eventually unified the property set of that binary expression.

The defined rules follow arithmetics. For example, the addition of two positive numbers results in a positive number. Similarly, a multiplication with zero results in zero. However, these rules do not respect integer overflows/underflows and may yield incorrect properties in such a case. As mentioned in Section 2.2, we presume that the sequential implementation is free of overflows, or alternatively, suggest enabling overflow runtime checks in .NET.

As a general example of how to infer the properties of a binary expression, consider the expression in (5), an extract of the statement in line 2 of the example's SSA form.

$$o_1 + i * 2 \tag{5}$$

Table 4. Inference rules for subtractions

Left	Right	Result
$\{\mid\}$	$\{\lightning\}$	$\mid$
$\{\lightning\}$	$\{\mid\}$	$\mid$
$\{\mid, +\}$	$\{-\}$	$\mid$
$\{-\}$	$\{\mid, +\}$	$\mid$
$\{\mid, -\}$	$\{+\}$	$\mid$
$\{+\}$	$\{\mid, -\}$	$\mid$
$\{\lightning\}$	$\{\lightning\}$	$\lightning$
$\{+\}$	$\{-\}$	$+$
$\{-\}$	$\{+\}$	$-$
$\{+\}$	$\{-\}$	$\not\simeq 0$
$\{-\}$	$\{+\}$	$\not\simeq 0$
Covered by copy rule.		$= 0$
Covered by copy rule.		$= 1$
$\emptyset$	$\{= 0\}$	Copy from left

Table 5. Inference rules for additions

Left	Right	Result
$\{\mid\}$	$\{\lightning\}$	$\mid$
$\{\lightning\}$	$\{\mid\}$	$\mid$
$\{\mid, +\}$	$\{+\}$	$\mid$
$\{+\}$	$\{\mid, +\}$	$\mid$
$\{\mid, -\}$	$\{-\}$	$\mid$
$\{-\}$	$\{\mid, -\}$	$\mid$
$\{\lightning\}$	$\{\lightning\}$	$\lightning$
$\{+\}$	$\{+\}$	$+$
$\{-\}$	$\{-\}$	$-$
$\{+\}$	$\{+\}$	$\not\simeq 0$
$\{-\}$	$\{-\}$	$\not\simeq 0$
Covered by copy rules.		$= 0$
Covered by copy rules.		$= 1$
$\emptyset$	$\{= 0\}$	Copy from left
$\{= 0\}$	$\emptyset$	Copy from right

Table 6. Inference rules for multiplications

Left	Right	Result
$\{\mid\}$	$\{\not\simeq 0\}$	$\mid$
$\{\not\simeq 0\}$	$\{\mid\}$	$\mid$
$\{\lightning\}$	$\{\lightning\}$	$\lightning$
$\{+\}$	$\{+\}$	$+$
$\{-\}$	$\{-\}$	$+$
$\{+\}$	$\{-\}$	$-$
$\{-\}$	$\{+\}$	$-$
$\{\not\simeq 0\}$	$\{\not\simeq 0\}$	$\not\simeq 0$
$\{= 0\}$	$\emptyset$	$= 0$
$\emptyset$	$\{= 0\}$	$= 0$
Covered by copy rules.		$= 1$
$\emptyset$	$\{= 1\}$	Copy from left
$\{= 1\}$	$\emptyset$	Copy from right

Table 7. Inference rules for divisions

Left	Right	Result
Covered by copy rules.		$\mid$
$\{\lightning\}$	$\{\lightning\}$	$\lightning$
Covered by copy rules.		$= 0$
Covered by copy rules.		$= 1$
$\emptyset$	$\{= 1\}$	Copy from left
$\{= 0\}$	$\emptyset$	Copy from left

Table 8. Inference rules for modulo

Left	Right	Result
$\{\lightning\}$	$\{\lightning\}$	$\lightning$
$\{= 0\}$	$\emptyset$	$= 0$
$\emptyset$	$\{= 1\}$	$= 0$

(6) to (8) show the properties of the operands.

$$Prop(o_1) = \{\nmid, \not\simeq 0, = 1, +\} \tag{6}$$
$$Prop(i) = \{|, \not\simeq 0, +\} \tag{7}$$
$$Prop(2) = \{\nmid, \not\simeq 0, +\} \tag{8}$$

Since the variable i is the loop index, it features the loop-dependence property $|$. The application of the rules within nested expressions follows conventional evaluation rules of the programming language. In this example, the multiplication is evaluated before the addition, i.e., the corresponding expression tree is evaluated bottom-up. As a consequence of the evaluation order, the multiplication rules are applied first. Table 6 denotes the necessary inference rules for multiplications. The $|$ property is derived because the left operand contains the property $\{|\}$, and the right contains $\{\not\simeq 0\}$. Moreover, $+$ and $\not\simeq 0$ are derived by the fact that both operands contain $\{+\}$ and $\{\not\simeq 0\}$. This finally leads to the properties of (9).

$$Prop(i * 2) = \{|, \not\simeq 0, +\} \tag{9}$$

Subsequently, the rules for additions listed in Table 5 have to be applied. Since the right operand has the properties $\{|, +\}$ and the left has an associated $\{+\}$, the $|$ property is obtained. Moreover, the properties $+$ and $\not\simeq 0$ are derived because the $\{+\}$ condition for both operands is satisfied. (10) denotes the resulting properties after the application of the addition rule.

$$Prop(o_1 + i * 2) = \{|, \not\simeq 0, +\} \tag{10}$$

Other Operations As a conservative approach, expressions without rules generate the empty property set. For example, floating point operations are not supported and yield no properties. Since our analysis is inter-procedural, the use of method invocations is however supported and propagates properties.

2.5 Dataflow Analysis

The previous sections described how to infer properties for a given expression or statement. This section now describes the application of these properties in the dataflow analysis. The properties have to be computed per basic block. Anything but variable assignments does not generate new properties; thus, the corresponding *Gen* functions are equal to the empty set as (11) to (14) show.

$$Gen(x[e_1] \leftarrow e_2) = \emptyset \tag{11}$$
$$Gen(\text{label } n) = \emptyset \tag{12}$$
$$Gen(\text{goto } n) = \emptyset \tag{13}$$
$$Gen(\text{if } e \text{ goto } n) = \emptyset \tag{14}$$

Assignments combine the properties retrieved from the expression e with the variable x to form the state tuples as illustrated in (G-ASGMT).

$$Gen(x \leftarrow e) = \{(x\ p) \mid p \in Prop(e)\} \tag{G-ASGMT}$$

The dataflow transfer function in (S-TRANS) eventually merges the current analysis state with the generated states of the statement s_ℓ by applying a unification.

$$Transfer(s_\ell) = Gen(s_\ell) \cup S \qquad \text{(S-TRANS)}$$

One way to solve the analysis is with the help of an iterative worklist algorithm [15, 23, 24]. The worklist can be initialized with all basic blocks of the SSA form. Although the initial order of the statements is irrelevant, a top-down initialization requires fewer iterations to converge. The algorithm will terminate in any case because of the limited size of the monotonously growing analysis state. This is because the transfer function introduced in (S-TRANS) does not discard any information, the number of state tuples may only increase. At most, it will have the size of the number of variables times the number of properties.

2.6 Array Intersection Analysis

Before answering the general question of the loop independency, we first have to determine potential array intersections. The output of the preceding dataflow analysis provides corresponding information, namely whether array index expressions are disjoint across all iteration steps of a loop. We additionally need an alias analysis [23] to identify potential aliases of an array, since this may be additional source for array access intersections. To deal with array aliases, we conservatively treat potential array aliases as if they would refer to the same array instance.

For each array and its potential aliases, we collect all array element access expressions. In the introduced example, this are the read from $f[p_1]$ in line 3 and the write to $f[p_1]$ in line 14. We then concentrate on the index expressions of the array accesses: If an array has accesses with different index expressions, we conservatively consider these accesses as potentially intersecting. We later relax this condition in Section 2.8. If the same index expression is used in all accesses of an array, we continue the intersection analysis for this array: In our example, this is the case since each access to the array f is made with the same index expression p_1.

In the case of a common access index expression, we look up the property set of this expression and check whether it contains the loop dependency property. This is the case for the index expression p_1 since the state S contains the tuple $p_1 \mid$. Due to the presence of the loop dependency property, it is ensured that each iteration computes a distinct index and thus no access intersections happen on the array f.

2.7 Loop Dependency Analysis

If a for loop contains I/O operations, potentially side-effecting calls, or writes to variables (not array elements!) declared outside the loop body, they are likely to be loop-dependent and are immediately rejected as parallelization candidates. Otherwise, we examine array accesses in the loops: We check for each array (and its aliases), whether it is read-only accessed or the array accesses do not intersect.

The latter information is obtained from the preceding array intersection analysis. If all these checks are fulfilled, the loop is guaranteed to be independent and can be parallelized.

The factorial example has obviously no array access intersections. Moreover, the loop performs neither I/O calls, nor side effects, nor writes to variables declared outside the loop body. Consequently, the loop is parallelizable.

2.8 Accuracy Improvements

A variety of optimization techniques can improve the results of the introduced analysis. For example, consider the situation sketched in Listing 6 that constructs the array indices with an offset. The lines 4 and 5 both access the same array

```
1 var m = a.Length - 1;
2 for(var i=0; i < m; i++) {
3   var x = i;
4   var o = a[x + 1];
5   a[x + 1] = o + 1;
6 }
```

Listing 6. Indices as common sub-expressions

```
1  var m = a.Length - 1;
2  var b = ...; // computed
3  for(var i=0; i < m; i++) {
4    if(b) {
5      a[i] = a[i] * 2;
6    } else {
7      var x = i + 1;
8      a[x] = a[x] * 3;
9    }
10 }
```

Listing 7. Intersecting array access with branches

element. However, they compute the index in two distinct locations. Therefore, there is no guarantee that the value of the variable x did not change in between. To cover such situations, we implement a common sub-expression elimination [23] together with a copy propagation [23] prior to the loop dependency analysis.

Another situation in which we miss the parallelization opportunity is sketched in Listing 7. In this example, two different branches are present. Each branch uses a different function to compute the array indices. One single branch would be detected as parallelizable but not their combination. However, since the condition and its result is the same for every iteration, the loop itself is parallelizable. One approach to aid the analysis to handle such situations is the application of program slicing [6, 11, 29] and analyzing each slice separately. Although, our tool does currently not implement this.

3 Implementation

The refactoring tool[2] is a static code analyzer implemented in C# with the help of the .NET Compiler Platform (Roslyn) [19]. It is a Visual Studio 2017 Version 15.6 [21] plugin that collects all for-loops within the source code and reports opportunities for parallelization. Our array access intersection analysis is based on interprocedural context-insensitive dataflow analysis as described in the previous section. To avoid cases where methods are overridden with polymorphism, our inter-procedural analysis only follows non-virtual methods.

Internally, the C# code is transformed into a three-address intermediate representation. During the conversion process, semantic checks prevent ambiguities of shadowed variables. Moreover, a copy propagation and common subexpression elimination are used to improve the precision of the array access analysis. We also implemented an alias analysis to detect potential array aliases. Although an SSA form is the best baseline for our analysis method, we did not implement SSA transformation for performance reasons (short refactoring tool reaction time) but used an over-approximation, namely reaching definition analysis instead. This simplification does not violate soundness of array access intersection analysis.

The implementation currently supports the following language features: 1) auto-properties 2) binary and unary expressions 3) coalesce expressions 4) compound assignments 5) conditional expressions 6) continue and break statements for inner loops 7) for-, while-, and do-statements 8) method invocations 9) multi-dimensional arrays 10) string interpolation. The unlisted language features are not implemented at this time and conservatively prevent parallelization of the for-loop.

Moreover, we support a small set of white-listed methods of the .NET framework that are side-effect-free and can be safely parallelized, such as `System.Math.Abs`. Our analysis currently analyzes each source code file in isolation and does not perform an inter-document full-solution analysis.

4 Experimental Evaluation

The refactoring tool was applied to a selection of open source projects available on GitHub. The list comprises well-known C# projects in general as well as projects by search strings such as image, algorithms, machine learning, and database are included. A project must have had at least five for-loops to be taken into account. Different software maturity levels are considered by selecting the projects with different user rankings (measured as the number of stars awarded by users in GitHub): from 56 up to 7,063. It is important to note that test code of the selected projects was excluded from the analysis to only consider production code in the evaluation.

Each scan was run on a computer with an Intel Core i7-7700HQ CPU with four 2.80 GHz cores and 16 GB RAM. The source code of the projects was located on an SSD (SK hynix SC308). On average, a scan of a project took 4.59

[2] The source code is available on `https://github.com/camrein/RefactorToParallel`

seconds. The measured times are the average out of three subsequent runs. The analysis automatically skips files without for loops hence the scan time does not necessarily correlate with the lines of code. Moreover, large files require more time because the analysis eagerly collects and optimizes methods for the interprocedural analysis even if they are not used by the for-loops.

During this evaluation, a for-loop was considered parallelizable if it can be replaced with its `Parallel.For` counterpart without breaking the as-if-serial property as well as without changing its body, i.e., not introducing synchronization in the loop body of the parallel counterpart. We focus on a specific type of for-loops where we apply array intersection analysis, so-called array-centric loops. These are for-loops with their bodies accessing arrays and engaging the elementary language features as listed in Section 3. We, therefore, exclude for-loops that work on collections, provoke virtual method calls, or use object accesses (except arrays and strings). Our motivation is to assess the quality of the array intersection analysis, by concentrating on mere array-accessing for-loops with language features implemented by our analysis. Twenty projects were analyzed, as listed in Table 9.

Table 9. Projects used for the analysis coverage evaluation

Project	LoC (C#)	Stars	Avg. Time (s)
AForge.NET	20,684	405	4.04
BrainSimulator	189,858	224	5.34
CSCore	45,325	860	4.75
C-Sharp-Algorithms	16,921	1,657	2.52
GeneticSharp	13,581	320	1.74
ImageProcessor	18,974	1,859	4.27
ImageSharp	27,966	1,790	2.13
Inbox2 desktop client	88,731	435	5.07
LiteDB	15,404	2,806	1.99
Microsoft BotBuilder	41,998	5,037	4.08
Naiad	28,886	371	4.59
Nancy	34,874	6,034	4.57
NHibernate	360,776	1,423	8.08
Popcorn	18,971	601	2.30
RavenDB	638,467	1,898	13.25
SharpBrain	1,330	53	2.11
SignalR	16,941	7,063	2.89
Spring.NET	52,208	456	9.18
Structure.Sketching	17,536	56	2.48
Veldrid	17,496	361	2.02

We scanned all projects for the described type of for-loops and counted 223 such array-centric for-loops. Ten of these loops make use of a computed array index and the other 183 use the loop index itself. We eventually run our loop

independence analysis on all these projects. The tool proves to be relatively fast, taking about 4.5 seconds on average per project. Our tool reported 193 of the selected 223 for-loops to be independent and parallelizable. 30 of 223 for-loops could not be identified as independent and would therefore not be parallelized. We eventually manually reviewed all the 223 loops for loop independence. As expected, all 193 reported parallelizable loops were indeed independent (no false positives). As for the 30 for-loops that the tool did not approve for parallelization, we identified six loops to be independent, i.e., 3% false negatives). To support these remaining 3%, we observed that an analysis supporting accesses to array ranges — such as the GCD dependence test — would be necessary. In summary, our tool shows effective detection for the case of "low-level" array-centric loops.

5 Related Work

There already exists a variety of utilities aiding engineers in the development of parallel code. This section briefly introduces a selection of different approaches.

Hydra [8] operates on the intermediate representation of LLVM [7, 16, 17]. Its underlying idea can be seen as C#'s async/await [18, 22] programming model. However, instead of manually declaring fragments with `async` and `await` respectively, it infers them automatically. To accomplish this, Hydra searches for parallelizable code fragments and computes an estimated cost of the sequential and the parallel code. If the estimated cost of the parallel code is lower than the original sequential, the function invocation is offloaded to a thread pool.

Sambamba [26, 27] aims at large-scoped automatic parallelization. Besides a conservative compile-time code analysis on LLVM's bitcode, it also incorporates a runtime system that optimistically parallelizes execution. For this purpose, the runtime system collects further information and replaces method code where appropriate. The collected information is thereby stored in a persistent storage that can be reused in later program executions. At runtime, Sambamba involves a software transactional memory [14] system to guard optimistically parallelized code. This STM system allows, different to the previous two approaches, the parallelization of code fragments that require synchronization.

Baar [3, 9, 10] is an approach that transparently offloads compute-intense program parts to a server. It provides an environment to allow the execution of programs in LLVM's IR. The goal is the runtime identification of code hotspots. Identified hotspots are offloaded to a server that applies additional processing such as parallelization and vectorization. The automatic parallelization is thereby accomplished with the LLVM subproject Polly [13, 25], based on the polyhedral model. The Message Passing Interface (MPI) [12] is used for inter-process communication; reducing the negative impact of expensive data transfers between client and server.

In summary, none of the introduced projects actively informs the user about the parallelization opportunities. In contrast, our implementation aids the developer with the parallelization by actively informing them inside the IDE, through-

out the entire source code. The developer may then decide if the parallelization is justified and apply the refactoring directly in the source code.

Besides the already introduced tools, there are also various analysis approaches. For example, the GCD dependence test [4, 23, 28] is a technique that allows deciding if array indices computed with linear expressions may intersect with other iterations. Although it does not support non-linear expressions like our approach, it is capable of identifying non-intersecting ranges.

Another powerful analysis technique is the polyhedral model (or polytope model) [1, 5]. It allows the transformation of loops which — in their original form — are not parallelizable but are in a restructured way. The detection of intersections is accomplished by the setup of inequalities and solving for an integer solution with the help of an ILP solver which can become very expensive for complex programs.

6 Conclusion

This paper introduced a light-weight conservative dataflow analysis to prove the absence of array access intersections in for-loops, covering linear and non-linear array element access patterns. It proves to be effective in terms of speed and accuracy for "low-level" for-loops that merely work on arrays. For this frequent case of for-loops, we achieved 97% precision with 4.5 second analysis time per multi KLOC project. We integrated the analysis in a practical Visual Studio IDE plugin operating on C# code. The tool actively informs the user about safe parallelization opportunities in the context of array-centric for-loops.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools, 2/e. Addison Wesley Boston (2003)
2. Akhter, S., Roberts, J.: Multi-core programming, vol. 33. Intel press Hillsboro (2006)
3. Baar (binary acceleration at runtime). `https://github.com/pc2/baar`, accessed: 2019-06-03
4. Banerjee, U.: Dependence testing in ordinary programs. Master's thesis, University of Illinois, Department of Computer Science (1976)
5. Bondhugula, U.K.: Effective automatic parallelization and locality optimization using the polyhedral model. Ph.D. thesis, The Ohio State University (2008)
6. Canfora, G., Cimitile, A., De Lucia, A.: Conditioned program slicing. Information and Software Technology **40**(11-12), 595–607 (1998)
7. Chansler, R., Bryant, R., Bryant, R., Canino-Koening, R., Cesarini, F., Allman, E., Bostic, K., Brown, T.: The architecture of open source applications (2011)
8. Chicken, J.: Hydra: Automatic parallelism using llvm (2014), homerton College
9. Damschen, M., Plessl, C.: Easy-to-use on-the-fly binary program acceleration on many-cores. arXiv preprint arXiv:1412.3906 (2014)
10. Damschen, M., Riebler, H., Vaz, G., Plessl, C.: Transparent offloading of computational hotspots from binary code to xeon phi. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015. pp. 1078–1083. IEEE (2015)

11. Danicic, S., Fox, C., Harman, M., Hierons, R.M.: Consit: A conditioned program slicer. In: IEEE International Conference on Software Maintenance (ICSM'00). pp. 216–226 (2000)
12. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, vol. 1. MIT press (1999)
13. Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., Pouchet, L.N.: Polly-polyhedral optimization in llvm. In: Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT). vol. 2011 (2011)
14. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2011)
15. Khedker, U., Sanyal, A., Sathe, B.: Data flow analysis: theory and practice. CRC Press (2009)
16. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004)
17. The llvm compiler infrastructure. `http://llvm.org`, accessed: 2019-06-03
18. Microsoft: Asynchronous programming with async and await (c# and visual basic). `https://msdn.microsoft.com/library/hh191443(vs.110).aspx`, accessed: 2019-06-03
19. Microsoft: .net compiler platform (roslyn). `https://github.com/dotnet/roslyn`, accessed: 2019-06-03
20. Microsoft: Task parallel library (tpl). `https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl`, accessed: 2019-06-03
21. Microsoft: Visual studio. `https://www.visualstudio.com`, accessed: 2019-06-03
22. Microsoft: C# language specification version 5.0. `https://www.microsoft.com/en-us/download/details.aspx?id=7029` (2015), accessed: 2019-06-03
23. Muchnick, S.S.: Advanced compiler design implementation. Morgan Kaufmann (1997)
24. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (2015)
25. Polly - llvm framework for high-level loop and data-locality optimizations. `https://polly.llvm.org`, accessed: 2019-06-03
26. Streit, K., Hammacher, C., Zeller, A., Hack, S.: Sambamba - adaptive optimization and parallelization of general purpose programs. `http://www.sambamba.org`, accessed: 2019-06-03
27. Streit, K., Hammacher, C., Zeller, A., Hack, S.: Sambamba: Runtime adaptive parallel execution. In: Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems. pp. 7:1–7:6. ADAPT '13, ACM, New York, NY, USA (2013)
28. Towle, R.A.: Control and Data Dependence for Program Transformations. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1976)
29. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering. pp. 439–449. ICSE '81, IEEE Press, Piscataway, NJ, USA (1981)

Sharing Nogoods between Portfolio Instances in Choco Solver

Extended Abstract

Ilja Becker, Sven Löffler, and Petra Hofstedt

Brandenburg University of Technology Cottbus – Senftenberg,
03046 Cottbus, Germany
{ilja.becker,sven.loeffler,hofstedt}@b-tu.de

1 Introduction

Parallelisation and learning have both been extensively researched in Constraint Programming (CP) throughout the past decades. Most parallelisation techniques in CP either split the search space or run independent instances in portofolios, focusing parallel work power either on a single view or using it seperatedly in independent views. This paper describes an attempt of sharing nogoods across portfolio instances in the Choco Solver, hence utilising the learnings from several views in each seperate instance. We briefly describe the learning infrastructure in Choco 4.0.9 with its particularities. Furthermore we present the implemented prototype for sharing nogoods and its limits. Our experiments show that sharing nogoods across instances can provide improvements over non-sharing learning strategies in Choco portfolio instances.

2 Related Work

There is a lot of current research concerning the parallel aspects of Constraint Programming [4]. While there has been research on parallel propagation, e.g. on parallel arc-consistency [11, 8] and on parallel propagation algorithms [17] most research focuses on parallel search. There are several methods to parallel search, two of them realising search space splitting either by work stealing [15, p. 342] or by Embarrassingly Parallel Search [16]. Another method are portfolio algorithms [15, p. 8]. For portfolio algorithms the main research focus is to generate diverse enough views for the portfolio instances [6, 5, 7, 9] in order to be able to profit from many instances.

In 2016 Chu et al. presented research on parallelising learning [3] within *Chuffed*, a lazy clause generation solver. We research whether their findings can be recreated in a lazy explanation learning based solver.

3 Sharing Choco Nogoods

Following we will set some fundamental definitions, describe the Choco solver and its learning infrastructure and furthermore our prototype implementation.

3.1 Preliminaries

A *Constraint Satisfaction Problem* (CSP) is a triple $M = \langle X, D, C \rangle$ where X is a set of variables $\{x_1, \ldots, x_n\}$, D is set of domains $D = \{D_1, \ldots, D_n\}$ associated respectively with these variables and C is a set of constraints. The constraints are formally represented by a pair $c = \langle \sigma, \rho \rangle$ where σ is the constraint's scope, a subset of the variables involved, and ρ, the constraint relation, a subset of the cartesian product of their domains [18, p. 376].

We define a nogood as follows: Given a CSP $M = \langle X, D, C \rangle$, any partial instantiation that does not appear in any solution of M is called a *nogood*. *Minimal* nogoods have no nogood subtuples [2, p. 153].

3.2 Choco Solver

The experiments we report on were conducted with version 4.0.9 of the Choco Solver [12]. In the following release 4.10.0 [13] major changes to the learning subsystem were introduced, so the following description is not representative of the current Choco implementation.

In Choco both modelling and search are organised and contained in the `Model`. The `Model` holds the variables, their associated domains, all constraints as well as a `Solver` object. The `Solver` is responsible for navigating through the search tree, propagation and learning, each managed by their dedicated `Move`, `Propagate` and `Learn` objects, respectively. Choco offers a variety of search strategies with static and dynamic heuristics as well as restarts.

The Choco Solver furthermore offers a `ParallelPortfolio` infrastructure, that enables the user to create a portfolio of `Model` instances that can be searched in parallel.

The solver also allows for "plugging in" a so-called `ExplanationEngine`. This objects manages the information necessary to deduce explanations in a lazy, asynchronous, low-intrusive and reverse manner [12, p. 30]. The implementation of the `ExplanationEngine` is based upon the *Event SElection Rules* (ESER) algorithm [14].

This explanation engine is then utilised by learn strategies like `LearnCBJ` and `LearnDBT` which implement conflict-directed and dynamic backtracking algorithms. They also allow to deduce and learn nogoods in the process. For learning nogoods the `Model` keeps a central "Nogood Store", represented by a dedicated nogood propagator. This nogood propagator internally maintains a dedicated SAT solver whose literals are mapped to their representative variables and their assignment. This allow leveraging SAT solver techniques such as *watched literals* originally described in [10].

3.3 Implementation Details

We implemented a search strategy called `LearnCBJSharedNogoods` based upon the strategy `LearnCBJ` provided by Choco. This allows to reuse the code necessary to maintain the `ExplanationEngine` and only adapt the code responsible for

the actual nogood learning. The prototype then follows a rather simple idea: The instances of `LearnCBJSharedNogoods` register upon construction with a global list in a synchronized manner. Then, whenever an instance learns a nogood, it shall be shared with the other `LearnCBJSharedNogoods` instances.

Since Choco uses trailing as a means of state recreation and each portfolio instance is its own `Model` the nogoods are passed as a list of tuples containing the variable name as a `String` and the nogood assignment value. Since naming is automated in Choco `Model`s the variable name is sufficient to map the nogood to its corresponding variable in each Portofolio instance. To reduce the effort each `LearnCBJSharedNogoods` instance maintains a `HashMap` that maps variable names to variables and is filled lazily during the search.

To avoid conflicts due to data races in the fairly complicated trailing infrastructure in Choco's centralised `Model` infrastructure the nogoods are initially only passed to the sharing `Learn` instance and only learned when the instance enters its own learning phase.

Note: One can configure the different portfolio instances' search parameters such that the search trees result to be the same across all instances. All instances then have the same search space, which they search with different heuristics. In that case sharing nogoods between those instances, crossing out certain parts of the search tree, in fact implements a form of nogood-based search space splitting.

4 Preliminary Results

Tests were run across a selection of problem instances from the 2018 Minizinc challenge [19] on quad-CPU system sporting AMD Opteron 6276 CPUs [1] and 512 GB of RAM. The testing setup utilised Choco's internal measurement system with a few additions to better judge the nogood utilisation.

One key insight is that the per-thread performance drops quickly when increasing the number of threads/portfolio instances. There are a number of possible causes to this, most likely this is caused by either an increase in JVM garbage collection or memory (cache) saturation.

Strategy \Threads	1	4	16	32
No Learning	2063	2009	1875	1570
CBJ (No Learning)	1450	1270	872	509
CBJ (Learning, unltd.)	993	431	124	115
CBJ (Sharing, unltd.)	600	170	40	67

Table 1. Processed Nodes per Second, Problem: "Elitserien Handball 16" from the 2018 MiniZinc Challenge, single run searching for all solutions, 60s timeout

The second key insight is that while many problems do not profit from nogoods at all, some instances profit not only from nogoods in general but from shared nogoods especially.

Strategy \Threads	1	4	16	32
No Learning	–	–	–	–
CBJ (No Learning)	–	–	–	–
CBJ (Learning, unltd.)	14.4	15.7	20.8	35.8
CBJ (Learning, nogood size max = 1000)	14.1	15.0	20.4	31.0
CBJ (Sharing, unltd.)	47.8	15.1	17.1	17.7
CBJ (Sharing, nogood size max = 1000)	49.2	16.0	14.5	8.6

Table 2. Time to find first solution in seconds ("–" if none found within timelimit), Problem: "Elitserien Handball 16" from the 2018 MiniZinc Challenge, single run, 60s timeout

5 Conclusion and Further Work

Further we shall investigate whether sharing nogoods provides advantages when searching for optimal solutions. Also of big interest would be to identify the attributes of instances that profit from nogoods, and whether this can be done automatically.

References

1. Advanced Micro Devices, Inc. (AMD): 6276 AMD. https://www.amd.com/de/products/cpu/6276 (2019), accessed: 13.08.2019
2. Dechter, R.: Constraint processing. Elsevier Morgan Kaufmann (2003)
3. Ehlers, T., Stuckey, P.J.: Parallelizing constraint programming with learning. In: Quimper, C. (ed.) Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9676, pp. 142–158. Springer (2016). https://doi.org/10.1007/978-3-319-33954-2
4. Gent, I.P., Miguel, I., Nightingale, P., McCreesh, C., Prosser, P., Moore, N.C.A., Unsworth, C.: A review of literature on parallel constraint solving. TPLP **18**(5-6), 725–758 (2018). https://doi.org/10.1017/S1471068418000340
5. Gomes, C.P., Selman, B.: Algorithm portfolio design: Theory vs. practice. In: Geiger, D., Shenoy, P.P. (eds.) UAI '97: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, Brown University, Providence, Rhode Island, USA, August 1-3, 1997. pp. 190–197. Morgan Kaufmann (1997)
6. Gomes, C.P., Selman, B.: Search strategies for hybrid search spaces. In: 11th IEEE International Conference on Tools with Artificial Intelligence, ICTAI '99, Chicago, Illinois, USA, November 8-10, 1999. pp. 359–364. IEEE Computer Society (1999). https://doi.org/10.1109/TAI.1999.809823

7. Gomes, C.P., Selman, B.: Hybrid search strategies for heterogeneous search spaces. International Journal on Artificial Intelligence Tools **9**(1), 45–57 (2000). https://doi.org/10.1142/S0218213000000057
8. Hamadi, Y.: Optimal distributed arc-consistency. Constraints **7**(3-4), 367–385 (2002). https://doi.org/10.1023/A:1020594125144
9. Kautz, H.A., Horvitz, E., Ruan, Y., Gomes, C.P., Selman, B.: Dynamic restart policies. In: Dechter, R., Kearns, M.J., Sutton, R.S. (eds.) Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada. pp. 674–681. AAAI Press / The MIT Press (2002), http://www.aaai.org/Conferences/AAAI/aaai02.php
10. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
11. Nguyen, T., Deville, Y.: A distributed arc-consistency algorithm. Sci. Comput. Program. **30**(1-2), 227–250 (1998). https://doi.org/10.1016/S0167-6423(97)00012-9
12. Prud'homme, C., Fages, J.G., Lorca, X.: Choco solver documentation. Tech. rep., TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (October 2018), https://github.com/chocoteam/choco-solver/blob/4.0.9/user_guide.pdf
13. Prud'homme, C., Fages, J.G., Lorca, X.: Choco solver documentation. Tech. rep., TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (December 2018), https://github.com/chocoteam/choco-solver/blob/4.10.0/user_guide.pdf
14. Prud'homme, C., Lorca, X., Jussien, N.: Event selection rules to compute explanations. CoRR **abs/1608.08015** (2016), http://arxiv.org/abs/1608.08015
15. Régin, J., Malapert, A.: Parallel constraint programming. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning., pp. 337–379. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3_9
16. Régin, J.C., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming. pp. 596–610. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
17. Rolf, C.C., Kuchcinski, K.: Parallel consistency in constraint programming. In: Arabnia, H.R. (ed.) Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2009, Las Vegas, Nevada, USA, July 13-17, 2009, 2 Volumes. pp. 638–644. CSREA Press (2009)
18. Smith, B.M.: Modelling. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2, pp. 377–406. Elsevier (2006). https://doi.org/10.1016/S1574-6526(06)80015-5
19. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The minizinc challenge 2008-2013. AI Magazine **35**(2), 55–60 (2014)

Eigenschaften von Mustern zur Herleitung geordneter Attributgrammatiken aus Mustern und Paradigmen

Christian Berg[1]

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
http://informatik.uni-halle.de
berg@informatik.uni-halle.de

Zusammenfassung Zur Verringerung der Geschwätzigkeit von Attributgrammatiken gibt es viele mögliche Paradigmen. Keine der Arbeiten, die diese Paradigmen vorgestellt haben, haben untersucht, welche formalen Eigenschaften diese Paradigmen einhalten oder wie dieses Einhalten sichergestellt werden kann. In dieser Arbeit werden die notwendigen Definitionen und Lemmata sowie entscheidende Sätze vorgestellt, die für die Konstruktion und Untersuchung der Paradigmen notwendig sind. Neben einer Verringerung des Spezifikationsumfangs um mindestens eine Größenordnung bietet die Untersuchung der Paradigmen darüber hinaus einen Einblick in die inkrementelle Konstruktion geordneter Attributgrammatiken.

Keywords: Geordnete Attributgrammatik · OAG · Übersetzerbau · Compiler Construction · Paradigmen · Patterns · Muster

1 Hintergrund und Motivation

Eine Methode um Programmiersprachen und Domänen-spezifische Sprachen zu implementieren, ist die Verwendung von Attributgrammatiken [19,32]. Attributgrammatiken (siehe [24]) sowie Erweiterungen auf Attributgrammatiken, wie bspw. [8,11,12,33,34] und unterschiedliche Interpretationen und Paradigmen von und auf Attributgrammatiken (siehe dazu [29]), wurden bereits vor langer Zeit eingeführt. Weiterhin ist bekannt, dass aus Attributgrammatiken, insbesondere geordneten Attributgrammatiken (siehe [18]), schnelle Übersetzer generiert werden können [4,20].

Während viele Arbeiten die grundlegende Bestimmung einer Strategie zur Auswertung einer Attributgrammatik widmeten – siehe dazu z. B. [8,9,31], konzentrierten sich andere Arbeiten auf die Verringerung des Umfangs der Spezifikation einer Sprachsemantik unter Verwendung von Attributgrammatiken – siehe dazu z. B. [11,19,29].

Die Motivation für die Einführung neuer Konstrukte in Attributgrammatiken – in dieser Arbeit als Muster bezeichnet, in [29] als Paradigma – begründet sich in der häufig vorgebrachten Kritik der „Geschwätzigkeit“ von Attributgrammatiken. So schrieb Koskimies in [26], dass „*Attributgrammatiken zu primitiv sind und nicht mehr als ein grundlegendes Rahmenwerk ... für die Implementierung der Sprachsemantik zu sein*“[1] – er verglich also Attributgrammatiken Anfang der 1990er mit Assembler: einfach, umfangreich und nicht unbedingt angenehm zu programmieren. Die Einführung der bereits erwähnten Paradigmen ist in der Lage diese Kritik zu beseitigen, wie die Gegenüberstellung in Abbildung 1 zeigt.

Die Verwendung von Paradigmen ermöglicht die Verringerung des Aufwands zur Spezifikation der Sprachsemantik erheblich. So wird in [3] berichtet, dass unter Ausnutzung bekannter Muster die Sprachsemantik reiner Attributgrammatiken *mindestens* um den Faktor 6 für ausgewählte Domänen-spezifische Sprachen größer ist.

Die Arbeit [27] ist dabei die einzige Arbeit, die diese Paradigmen bzgl. formaler Eigenschaften untersucht. Lämmel und Riedewald beschreiben Paradigmen dabei als Meta-Programme auf Attributgrammatiken und sind in diesem Aspekt ähnlich [23]. Obwohl die Darstellung in [27] sehr mächtig ist, beschränken sich die Autoren auf die Prüfung der Vollständigkeit und Eindeutigkeit in Attributgrammatiken. Weitere Eigenschaften,

[1] Im Original: *The concept of an attribute grammar is too primitive to be nothing more but a basic framework, the 'machine language' of language implementation.*

```
1  def_before_use PDef.bind, EDef.bind, VDef.bind
2                 of PDef.sym, Edef.sym, Vdef.sym
3                    PRef.bind, ERef.bind, VRef.bind
4                 of PRef.sym, Eref.sym, VRef.sym
5    in Program.proc_names, Program.event_names,
6       Program.var_names
7  with error unknown, error unknown, error unknown
8  with error predef, error predef, error predef
```

(a) Variante mit Mustern

```
1   rule Program ::= Process
2   attr Process.proc_namesIn ← ∅
3
4   rule Process ::= PDef PStats
5   attr PDef.proc_namesIn ← Process.proc_namesIn
6        PStats.proc_namesIn ← PDef.proc_namesOut
7        Process.proc_namesOut
8          ← PStats.proc_namesOut
    ...
220 rule EDef ::= ID
221 attr EDef.var_namesOut ← EDef.var_namesIn
222
223 rule ERef ::= ID
224 attr ERef.var_namesOut ← ERef.var_namesIn
225
226 rule VDef ::= ID
227 attr VDef.bind ← bindKey(VDef.var_namesIn, VDef.sym)
228      VDef.var_namesOut ← VDef.var_namesIn ↞VDef.bind
229 cond VDef.sym ∉ VDef.var_namesIn
230    ⇒ error "Already Defined " ++ VDef.sym
```

(b) Variante ohne Muster

Abbildung 1: Ausschnitt aus [3] zur Implementierung der Namensanalyse mit geordneten Attributgrammatiken gegenübergestellt mit derselben Variante unter Ausnutzung von Mustern für die Beispielsprache aus [2].

wie Zerlegbarkeit, die Herstellung einer berechenbaren Reihenfolge oder ob eine Attributgrammatik geordnet ist, gehören nicht zu den untersuchten Eigenschaften.

In dieser Arbeit werden die Eigenschaften von Mustern und Paradigment sowie deren Aufbau (Abschnitt 3) sowie, welche Eigenschaften diese einhalten (Abschnitt 3.2), vorgestellt. Eine Beschreibung der aus der Verwendung von Paradigmen und dem Beweis zu deren Aufbau und die Konstruktion geordneter Attributgrammatiken aus diesen gewonnenen Erkenntnisse folgt in Abschnitt 4 Eine Auswertung der Ergebnisse und eine Zusammenfassung folgt in Abschnitt 5. Eine Einleitung zu Attributgrammatiken und den gewünschten Eigenschaften zum Erhalt schneller Übersetzer erfolgt in Abschnitt 2

2 Attributgrammatiken und Berechnungsreihenfolgen

Mit einer Attributgrammatik kann der kontextsensitive Teil einer Programmiersprache beschrieben werden – eine Erkenntnis, die Knuth in [24] vorgestellt hat. Attributgrammatiken erweitern damit eine kontextfreie Sprache – die (abstrakte) Syntax einer Programmiersprache – um Attribute und deren Berechnungsvorschrift. Die abstrakte Syntax einer Sprache gegeben als kontextfreie Grammatik lässt sich wie folgt definieren:

Definition 1. *(siehe u. a. [1,35]) Ein Tupel* $G \triangleq (N, T, P, Z)$ *wobei*

- N *eine endliche Menge an Nichtterminalen ist,*
- T *eine endliche Menge an Terminalen ist,*
- $P \subseteq N \times \Sigma^*$ *eine endliche Menge an Produktionen ist und*
- $Z \in N$

heißt ***kontextfreie Grammatik****, wobei* $\Sigma = N \cup T$ *die Menge möglicher Symbole ist und* Z *ein ausgezeichnetes Nichtterminal, welches als Startsymbol bezeichnet wird.*

Nur aus Terminalsymbolen bestehende Worte, die aus dem Startsymbol hergeleitet werden können, bilden die durch G *erkannte Sprache*. Ein Symbol `X` ist aus einem Nichtterminal `Y` herleitbar, wenn es eine Produktion gibt, für die `Y` auf der linken Seite der Produktion vorkommt und `X` auf der rechten Seite oder, wenn es andere Symbole gibt, aus denen `X` herleitbar ist und die aus `Y` hergeleitet werden können. Dann heißt `Y` auch *Vorfahr* von `X` und analog `X` *Nachfahr* von `Y`. Die Anzahl der Vorkommen eines Symbols `X` für eine gesamte Produktion p (linke und rechte Seite) wird mit $\lceil \mathtt{X} \rceil_p$ bezeichnet, die Anzahl der Vorkommen auf der rechten Seite der Produktion mit $|\mathtt{X}|_p$.

Attributgrammatiken werden üblicherweise wie folgt definiert:

Definition 2. *(siehe u. a. [18,24,35]) Ein Tupel* $AG \triangleq (G, A, R, B)$ *heißt* ***Attributgrammatik****, wobei*

- $G = (N, T, P, Z)$ *eine kontextfreie Grammatik ist,*
- $A = \biguplus_{\mathsf{X} \in \Sigma} A_{\mathsf{X}}$ *eine endliche Menge von* ***Attributen****,*
- $R = \biguplus_{p \in P} R_p$ *eine endliche Menge von* ***Attributierungsregeln*** *der Form* $a_0 \leftarrow f(a_1, \cdots, a_n)$ *ist, wobei* f *eine Funktion ist und*
- $B = \biguplus_{p \in P} B_p$ *eine endliche Menge von* ***Bedingungen*** *der Form* $\phi(a_1, \cdots, a_n)$ *ist, wobei* ϕ *ein Prädikat ist,*

wobei für alle $a_0 \leftarrow f(a_1, \cdots, a_k) \in R_p$ *einer Produktion* $p \in P, p = \mathsf{X}_0 ::= \mathsf{X}_1 \ \cdots \ \mathsf{X}_n$ *muss* $a_i \in A_{\mathsf{X}_0} \cup \cdots \cup A_{\mathsf{X}_n}$, $i = 0, \cdots, k$ *sein,* a_0 *ist das in* p ***definierte Attribut****, geschrieben* $a_0 \in AD_p$*; für alle Bedingungen* $\phi(a_1, \cdots, a_k) \in B_p$ *für die Produktion* $p \in P, p = \mathsf{X}_0 ::= \mathsf{X}_1 \ \cdots \ \mathsf{X}_n$ *muss analog* $a_i \in A_{\mathsf{X}_0} \cup \ \cdots \cup A_{\mathsf{X}_n}$ *sein.*

Um die Reihenfolge der Berechnung von Attributen zu bestimmen, werden Attribute eingeteilt:

Definition 3. *(siehe [24,35]) Sei* $AG \triangleq (G, A, R, B)$ *eine Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$*. Seien* $u, v, w \in \Sigma^*$ *und* e *beliebiger Ausdruck. Ein Attribut* $a \in A$ *heißt*

ererbt *gdw.* $a \in A_{\mathsf{X}}$ *und für eine Produktion* $p \in P$, $p = \mathsf{Y} ::= u \ \mathsf{X} \ v$ *die Attributierungsregel* $\mathsf{X}.a \leftarrow e \in R_p$ *ist und*

synthetisiert *gdw.* $a \in A_{\mathsf{X}}$ *und für eine Produktion* $p \in P$, $p = \mathsf{X} ::= w$ *die Attributierungsregel* $\mathsf{X}.a \leftarrow e \in R_p$ *ist.*

Die Bestimmung der Berechnungsreihenfolge bezieht sich üblicherweise auf „konkrete“ Programme und damit auf, sogenannte, abstrakte Syntaxbäume:

Definition 4. *(siehe [35])Sei* $G \triangleq (N, T, P, Z)$ *eine kontextfreie Grammatik und* $w \in L(G)$*. Sei* $AST = (E, K)$ *ein markierter, geordneter Baum mit Wurzel* k_0 *und* $k_1, \cdots, k_n$ *für* $n \in \mathbb{N}$ *und* $n > 0$ *unmittelbare Nachfolger von* k_0 *sowie eine Markierungsfunktion* $f \colon E \to M$*, dann heißt AST* ***abstrakter Syntaxbaum*** *des Wortes* w *bzgl.* G *genau dann, wenn folgende Bedingungen gelten:*

1. $M \subseteq \Sigma \cup \{\varepsilon\}$
2. $f(k_0) = Z$
3. $Z ::= f(k_1) \cdots f(k_n) \in P$
4. *ist* $f(k_i) \in T$ *oder* $n = 1$ *und* $f(k_i) = \varepsilon$ *so ist* k_i *ein Blatt*
5. *ist* $f(k_i) \in N$ *dann ist* k_i *Wurzel eines abstrakten Syntaxbaums bezüglich der Grammatik* $G' = (T, N, P, f(k_i))$ *für ein Wort* $w' \in L(G')$

Dabei heißt k_i i***-ter Nachfahre*** *von* k_0 *und für jedes* k_i *heißt* k_0 ***Vorfahr*** *bzgl. des abstrakten Syntaxbaums AST. Für einen Knoten* k *heißt* $f(k)$ ***Typ des Knotens*** k *falls* $f(k) \neq \varepsilon$*. Statt Nachfahr oder Vorfahr werden diese auch als Kindknoten oder Elternknoten bezeichnet.*

Analog der Markierungsfunktion bildet $Typ(k)$ einen Knoten auf das zugrunde liegende Symbol der abstrakten Syntax ab.

Definition 5. *(siehe u. a. [24,35]) Eine Attributgrammatik* $AG \triangleq (G, A, R, B)$ *heißt* ***konsistent*** *genau dann, wenn für alle abstrakten Syntaxbäume AST folgende Bedingungen erfüllt sind:*

- *für jedes Attribut* $a \in A$ *in jedem Knoten* k *höchstens eine definierende Regel* $\mathsf{X}.a \leftarrow e \in R_p$ *existiert, für einen beliebigen Ausdruck* e *und*
- *der* $Typ(k) = \mathsf{X}$ *ist.*

Die Attributgrammatik heißt ***vollständig*** *wenn in jedem Knoten* k *mindestens eine Regel* $\mathsf{X}.a \leftarrow e \in R$ *existiert.*

Bestimmung der Konsistenz und Vollständigkeit werden u. a. in [24,35] beschrieben. Die verschiedenen Arten der Bestimmung in welcher Reihenfolge die Attribute bestimmt werden können, würde den Rahmen dieser Einführung sprengen. An dieser Stelle sei daher auf die Arbeiten [7,9,16,18,28,31,35] verwiesen.

In dieser Arbeit werden geordnete Attributgrammatiken verwendet, bei denen eine topologische Sortierung der Attributgabhängigkeiten zur Bestimmung der Reihenfolge verwendet wird. Ein definiertes Attribut hängt von allen Attributen, die auf der rechten Seite der Berechnungsvorschrift vorkommen, ab. Eine Liste von Attributen für einen abstraten Syntaxbaum ist eine berechenbare Reihenfolge für diesen Baum, wenn jedes Attribut eines Knotens genau einmal vorkommt und alle Attribute von denen dieses Attribut abhängt, vorher in dieser Liste stehen. Die folgenden Definitionen sind im wesentlichen [18] und [35] entnommen.

Definition 6. *Sei* $AG \triangleq (G, A, R, B)$ *eine Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$. *Für jede Produktion* $p : \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n \in P$ *mit Attributierungsregel* $\mathtt{X}_j.b \leftarrow f(\ \cdots \ \mathtt{X}_i.a \ \cdots)$ *heißt* $\mathtt{X}_i.a \rightarrow \mathtt{X}_j.b \subseteq A \times A$ ***lokale Abhängigkeit der Produktion*** *p. Für eine Produktion* $p \in P$ *heißt* $DG_p \triangleq (A_p, DDP_p)$ *mit* $A_p \triangleq A_{\mathtt{X}_0} \cup \ \cdots \ \cup A_{\mathtt{X}_n}$ *und* $DDP_p \triangleq \{\mathtt{X}_i.a \rightarrow \mathtt{X}_j.b \colon \mathtt{X}_j.b \leftarrow f(\cdots \mathtt{X}_i.a \ \cdots) \in R_p\}$ ***lokaler Abhängigskeitsgraph zu*** *p*

Ist DG_p *azyklisch für alle* $p \in P$, *dann heißt AG* ***lokal azyklisch****.*

Für die Bestimmung einer topologischen Sortierung, die unabhängig von abstrakten Syntaxbäumen durchgeführt werden kann, werden mittels folgender Definition Graphen konstruiert, sodass alle Kontexte in Abhängigkeitsgraphen vorhanden sind:

Definition 7. *Sei* $AG \triangleq (G, A, R, B)$ *eine Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$. *Für alle Produktionen* $p \in P$ *mit* $p = \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n$ *heißt* $NDDP_p = DDP_p^+ \setminus \{\mathtt{X}_i.a \rightarrow \mathtt{X}_j.b) \colon \mathtt{X}_i.a$ *und* $\mathtt{X}_j.b$ *werden in p definiert* $\}$ ***normalisierte direkte Abhängigkeiten der Produktion*** *p. Für alle Symbole* $\mathtt{X} \in \Sigma$ *sind die* ***induzierten Attributabhängigkeiten*** *das kleinste System von Mengen* $IDS_{\mathtt{X}} \subseteq A_{\mathtt{X}} \times A_{\mathtt{X}}$, $IDP_p \subseteq A \times A$ *welches folgende Gleichungen erfüllt*

1. $NDDP_p \subseteq IDP_p$
2. $IDS_{\mathtt{X}} = \{\mathtt{X}.a \rightarrow \mathtt{X}.b \colon \exists q \in P$ *sodass* $\mathtt{X}.a \rightarrow \mathtt{X}.b \in IDP_q^+\}$
3. $IDP_p = IDP_p \cup IDS_{\mathtt{X}_0} \cup \ \cdots \ \cup IDS_{\mathtt{X}_n}$

IDS sind die ***induzierten Abhängigkeiten zwischen Symbolattributen****, IDP die* ***induzierten Abhängigkeiten zwischen Attributvorkommen****.*

Die Reihenfolge in der Attribute dann berechnet werden können, teilt sich in Gruppen synthetisierter und ererbter Attribute. Bei geordneten Attributgrammatiken können diese Gruppen in Teilen auch leer sein. Im Folgenden sind AS und AI die Mengen synthetisierter bzw. ererbter Attribute, ggf. mit Index des dazugehörigen Symbols.

Definition 8. *Für eine Attributgrammatik* $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und allen Symbolen* $\mathtt{X} \in \Sigma$ *heißt die Partitionierung von* $A_{\mathtt{X}} = A_{\mathtt{X}}(1) \uplus \ \cdots \ \uplus A_{\mathtt{X}}(m_{\mathtt{X}})$ ***zulässige Zerlegung*** *genau dann, wenn für alle Symbole* $\mathtt{X}$ *gilt* $A_{\mathtt{X}}(i) \subseteq AS_{\mathtt{X}}$ *für* $i = m_{\mathtt{X}}, m_{\mathtt{X}} - 2, \cdots$ *und* $A_{\mathtt{X}}(i) \subseteq AI_{\mathtt{X}}$ *für* $i = m_{\mathtt{X}} - 1, m_{\mathtt{X}} - 3, \cdots$.

Ist AG lokal azyklisch und existiert für jedes Symbol $\mathtt{X}$ *eine zulässige Zerlegung, sodass für jeden abstrakten Syntaxbaum AST eine berechenbare Reihenfolge existiert so, dass die Attribute des Knoten K mit* $Typ(K) = \mathtt{X}$ *in der Reihenfolge* $A_{\mathtt{X}}(1), \cdots, A_{\mathtt{X}}(m_{\mathtt{X}})$ *berechnet werden können, dann heißt AG* ***zerlegbar****.*

Eine zerlegbare Attributgrammatik mit zulässiger Zerlegung heißt ***zerlegt****.*

Dies erlaubt die Definition einer geordneten Attributgrammatik:

Definition 9. *(siehe [18]) Sei* $AG \triangleq (G, A, R, B)$ *eine Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$. *Für alle Symbole* $\mathtt{X} \in \Sigma$ *seien*

1. $T_{\mathtt{X}}(-1) \triangleq \emptyset$
2. $T_{\mathtt{X}}(0) \triangleq \emptyset$
3. $T_{\mathtt{X}}(2k-1) \triangleq \{a \in AS_{\mathtt{X}}$ *so, dass* $\forall b \in A_{\mathtt{X}} \colon \mathtt{X}.a \rightarrow \mathtt{X}.b \in IDS_{\mathtt{X}} \implies \exists\ j \leq 2k-1$ *mit* $\mathtt{X}.b \in T_{\mathtt{X}}(j)\}$

4. $T_{\mathsf{X}}(2k) \triangleq \{a \in AI_{\mathsf{X}}$ *so, dass* $\forall b \in A_{\mathsf{X}} \colon \mathsf{X}.a \to \mathsf{X}.b \in IDS_{\mathsf{X}} \implies \exists\, j \leq 2k$ *mit* $\mathsf{X}.b \in T_{\mathsf{X}}(j)\}$
5. $A_{\mathsf{X}} \triangleq T_{\mathsf{X}}(m-i+1) \setminus T_{\mathsf{X}}(m-i-1)$

wobei $m \in \mathbb{N}$ *minimal und* $T_{\mathsf{X}}(m-1) \cup T_{\mathsf{X}}(m) = A_{\mathsf{X}}$. *Die Attributgrammatik heißt* ***geordnet*** *(OAG) genau dann, wenn sie mit den Partitionen* $A_{\mathsf{X}} = A_{\mathsf{X}}(1) \uplus \cdots \uplus A_{\mathsf{X}}(m_{\mathsf{X}})$ *zerlegt ist und für alle Produktionen* $p \in P$ *der erweiterte Abhängigkeitsgraph* $EDP_p \triangleq IDP_p \cup \{\mathsf{X}.a \to \mathsf{X}.b \colon \exists h,k \in \mathbb{N}, \mathsf{X}.a \in A_{\mathsf{X}}(h) \wedge \mathsf{X}.b \in A_{\mathsf{X}}(k) \wedge h < k\}$ *azyklisch ist.*

3 Muster und deren Konstruktion

Grundsätzlich ist ein Muster eine Funktion, die eine geordnete oder zerlegbare Attributgrammatik in eine anderere geordnete oder zerlegbare Attributgrammatik überführt. In dieser Arbeit wird diese Funktion über die Mengendifferenz und Mengenvereinigung der Bestandteile einer Attributgrammatik formuliert. Weiterhin wird darauf verzichtet Bedingungen explizit zu betrachten. In der Literatur ist dies ein übliches Vorgehen [9,19,33].

Die folgenden Definitionen und Lemmata sind [3] entnommen und, soweit möglich und das Verständnis nicht behindernd, gekürzt.

Definition 10. *Sei* $\mathcal{AG}_G$ *die Menge aller Attributgrammatiken mit abstrakter Syntax* G. *Ein* ***Muster*** *ist eine Abbildung* $\mathcal{M}_G \colon \mathcal{AG} \to \mathcal{AG}$. *Seien* $AG, AG' \in \mathcal{AG}_G$ *Attributgrammatiken mit* $AG' = \mathcal{M}_G(AG)$, *dann heißt* AG' *die* ***resultierende Attributgrammatik*** *nach Anwendung des Musters* $\mathcal{M}_G$. *Sind* AG *und* AG' *zerlegbar so heißt das Muster* $\mathcal{M}_G$ ***zerlegungserhaltend****, sind beide Attributgrammatiken geordnet* ***ordnungserhaltend****.*

Die Konstruktion der resutlierenden Attributgrammatik erfolgt mittels, sogenannter, Änderungsmengen. Das Hinzufügen und Entfernen von Elementen in Mengen ist so grundlegend, dass dies in vielen Arbeiten in einer Form oder einer anderen wiederkehrt.

Definition 11. *Sei* $\mathcal{AG}_G$ *die Menge aller Attributgrammatiken mit abstrakter Syntax* G *und* $AG \in \mathcal{AG}_G$, $AG \triangleq (G, A, R, B)$ *eine zerlegbare Attributgrammatik. Ein Tupel* $\mathsf{M}_{AG} \triangleq (\mathsf{A}_-, \mathsf{A}_+, \mathsf{R}_-, \mathsf{R}_+)$ *definiert die resultierende Attributgrammatik* $AG' \triangleq (G', A', R', B')$ *mit*

- $G' = G$
- $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$
- $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$
- $B' = B$.

M_{AG} *heißt dann* ***Attributgrammatik-abhängige Musterdefinition*** *zu* AG, *die Mengen des Tupels* $(\mathsf{A}_-, \mathsf{A}_+, \mathsf{R}_-, \mathsf{R}_+)$ *heißen* ***Änderungsmengen (einer Attributgrammatik)****.*

Die Frage, wie diese Änderungsmengen konstruiert sein müssen, damit eine zerlegbare Attributgrammatik in eine zerlegbare Attributgrammatik überführt wird, ist eine der wesentlichen Fragestellungen dieser Arbeit.

Paradigmen, für die [29] einen Überblick gibt, wurden bisher nicht in Form von Mustern als Änderungsmengen dargestellt. Lämmel und Riedewalt zeigen jedoch in [27] über eine Meta-Sprache, dass diese Meta-Sprache für einige Paradigmen aus [29], azyklische Attributgrammatiken erzeugt. Auf Zerlegbarkeit oder Ordnung der Attributgrammatik wird nicht eingegangen. Für die Überprüfung ob Änderungsmengen, wie sie im Rahmen dieser Arbeit verwendet werden, die Zerlegbarkeit erhalten wird folgende Hilfsdefinition benötigt.

Definition 12. *Sei* $AG \in \mathcal{AG}_G$, $AG \triangleq (G, A, R, B)$ *eine zerlegbare Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$, *für ein Attribut* $a \notin AD_p$ *einer Produktion* $p \in P$ *und einem Symbol* $\mathsf{X} \in \Sigma$, *wobei* $\lceil X \rceil_p \geq 1$, *Attributen* $b_1, \cdots, b_n \in A$ *und für alle* b_i, $1 \leq i \leq n$ *mit* $n \in \mathbb{N}$ *gilt, dass* $b_i \neq a$ *und es existieren* $q_i \in P$, *sodass* $b_i \in AD_p$ *und* $b_i \in A(Y_i)$ *mit* $\lceil Y_i \rceil_{q_i} - |Y_i|_{q_i} = 1$ *für* $Y_i \in \Sigma$ *und* $\lceil Y_i \rceil_p > 0$. *Eine zu* AG ***vereinbar hinzufügbare*** *Attributierungsregel* $r^{(a,p)}$ *für das Attribut* a *zu* p, *sodass* $AG' = (G, A', R', B)$ *mit* $A' = A \cup \{a\}$, $R' = R \cup \{r^{(a,p)}\}$ *ist induktiv wie folgt definiert:*

1. $r^{(a,p)} = a \leftarrow c$ *für eine Konstante* c *ist vereinbar hinzufügbar*
2. $r^{(a,p)} = a \leftarrow f(b_1, \cdots, b_n)$ *ist vereinbar hinzufügbar, falls* $r^{(a,p)} \in R'_p$ *und* DG_p *bzgl.* R' *azyklisch ist.*

Grundsätzlich ist, trotz Definition 12 möglich, dass durch Hinzufügen vereinbar hinzufügbarer Attributierungsregeln ein Zyklus zu AG hinzugefügt wird. Dies kann dann passieren, wenn a bereits in AG existiert, jedoch in einer an p „angrenzenden" Produktion definiert wird, bspw. in einem der Y_i nach Definition 12. Ist a bspw. synthetisiert für ein Symbol der rechten Seite der Produktion p und wird in p ererbt definiert, dann lässt sich leicht ein Zyklus in DG_{q_i} erzeugen, der Abhängigkeitsgraph in p bleibt allerdings nach Definition 12 azyklisch. Gleichwohl ist AG' dann inkonsistent, da es zwei definierende Attributierungsregeln, die nicht identisch sind, für a gibt. Auf den Beweis des entsprechenden Lemmas wird verzichtet. In [3] wird auf diese und weitere Probleme genauer eingegangen.

Änderungsmengen, die auf Basis vereinbar hinzufügbarer Attributierungsregeln konstruiert werden, müssen folgende Eigenschaften einhalten:

Theorem 1. *Sei* $AG \in \mathcal{AG}_G$, $AG \triangleq (G, A, R, B)$ *eine zerlegbare Attributgrammatik mit abstrakter Syntax* G. *Die Änderungsmengen* A_-, A_+, R_- *und* R_+ *erzeugen eine zerlegbare resultierende Attributgrammatik* $AG' = (G, A', R', B)$ *wobei* $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$ *und* $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$, *wenn folgende Bedingungnen bzgl. der Änderungsmengen gelten:*

1.
$$\begin{aligned}
&\{b \colon \exists \langle a, \cdots, b\rangle \in DG_p,\ a \in AD_p\ \wedge\ a \in \mathsf{A}_-\ \wedge\ \neg(\exists r \in \mathsf{R}_+ : r = a \leftarrow f(\cdots, b, \cdots))\}\\
&\cup \{a' \colon \exists r \in \mathsf{R}_-,\ r = a' \leftarrow f(\cdots)\ \wedge\ \neg(\exists r \in \mathsf{R}_+, r = a' \leftarrow f(\cdots))\}\\
&\subseteq \mathsf{A}_- \subseteq A\\
&\textit{und}\\
&\{r \colon r \in R\ \wedge\ \exists a \in \mathsf{A}_-,\ r = a \leftarrow f(\cdots)\}\\
&\cup \{r \colon r \in R, r = a \leftarrow f(\cdots) \wedge (\exists r' = a \leftarrow f(\cdots) \in \mathsf{R}_- \wedge \forall r'' \in \mathsf{R}_+ : r'' \neq a \leftarrow f(\cdots))\}\\
&\cup \{r \colon r \in R, r = a \leftarrow f(\cdots, b, \cdots) : b \in \mathsf{A}_- \wedge \neg(\exists r' \in \mathsf{R}_+, r' = a \leftarrow f(\cdots, b, \cdots))\}\\
&\subseteq \mathsf{R}_- \subseteq R
\end{aligned}$$

2.
$$\begin{aligned}
\mathsf{R}_{p,+,a,s,X} = \{&r \colon r \in \mathsf{R}_{p,+}, r = a \leftarrow f(\cdots), a \in \mathsf{A}_+, a \in A'(X), a\ \textit{synthetisiert}\ ,\\
&p = X ::= u \in P, u \in \Sigma^*\}\\
\mathsf{R}_{+,X,a,s} = &\biguplus_{p \in P, p = X\ ::=\ u, u \in \Sigma^*} \mathsf{R}_{p,+,a,s,X}
\end{aligned}$$

und $\left|\mathsf{R}_{p,+,a,s,X}\right| = 1$ *und* $\left|\mathsf{R}_{+,X,a,s}\right| = \left|\{p \colon p \in P, p = X ::= u, u \in \Sigma^*\}\right|$. *Sowie*

$$\{a \colon r = a \leftarrow f(\cdots), r \in \mathsf{R}_{p,+,a,s,X}\} \subseteq \mathsf{A}_+$$

ist.

3.
$$\begin{aligned}
\mathsf{R}_{p,+,a,i,X} = \{&r \colon r \in \mathsf{R}_{p,+}, r = X_i.a \leftarrow f(\cdots), a \in \mathsf{A}_+, a \in A'(X), a\ \textit{ererbt}\ ,\\
&p = Y ::= u\ X\ v, X \in \Sigma, Y \in N, u, v \in \Sigma^*, 1 \leq i \leq |X|_p, r \neq Y.a \leftarrow f(\cdots)\}\\
\mathsf{R}_{+,X,a,i} = &\biguplus_{p \in P, p = Y\ ::=\ u\ X\ v, u, v \in \Sigma^*, Y \in N} \mathsf{R}_{p,+,a,i,X}
\end{aligned}$$

wobei $\left|\mathsf{R}_{p,+,a,i,X}\right| = |X|_p$, $r_i \neq r_j, r_i, r_j \in \mathsf{R}_{p,+,a,i,X}$, $i \neq j, 1 \leq i \leq \left|\mathsf{R}_{p,+,a,i,X}\right|$, $1 \leq j \leq \left|\mathsf{R}_{p,+,a,i,X}\right|$ *und* $\left|\mathsf{R}_{+,X,a,i}\right| = \sum_{p \in P, p = \mathsf{Y}\ ::=\ u\ \mathsf{X}\ v} |\mathsf{X}|_p$ *für* $\mathsf{Y} \in N, u, v \in \Sigma^*$ *Sowie*

$$\{a \colon r = a \leftarrow f(\cdots), r \in \mathsf{R}_{p,+,a,i,X}\} \subseteq \mathsf{A}_+$$

ist.

4. *Für alle* $r \in \mathsf{R}_+$ *gilt* r *ist vereinbar hinzufügbar zu* R'' *mit* $R'' = R \setminus \mathsf{R}_-$
5. *Für alle Produktionen* $p \in P$*, alle Symbole* $X \in \Sigma$*, alle Attribute* $a \in \mathsf{A}_+$ *gilt: ist* $r = a \leftarrow f(\cdots) \in \mathsf{R}_{p,+,a,s,X}$*, dann ist* $r \notin \mathsf{R}_{q,+,a,i,X}$ *für* $q \in P$*; analog ist* $r = a \leftarrow f(\cdots) \in \mathsf{R}_{p,+,a,i,X}$ *dann* $r \notin \mathsf{R}_{q,+,a,s,X}$ *für ein* $q \in P$*.*
6. $\mathsf{A}_+ \subseteq \{a \colon r = a \leftarrow f(\cdots) \in \mathsf{R}_+\}$ *und*

$$\mathsf{R}_+ = \biguplus_{p \in P, a \in \mathsf{A}_+, X \in \Sigma} (\mathsf{R}_{p,+,a,i,X} \uplus \mathsf{R}_{p,+,a,s,X})$$

Satz 1 gibt einen Hinweis auf den Aufbau der Änderungsmengen. Da der Beweis eine Reihe zusätzlicher Hilfslemmata benötigt sei auf [3] verwiesen. Die erste Eigenschaft drückt aus, dass, sollte ein Attribut entfernt werden und es gibt keine „Redefinition“ des Attributs, dann auch alle Attribute, die dieses Attribut lesend benutzen entfernt werden müssen. Analoges für Attributierungsregeln. Dies natürlich auch in jedem Kontext, in dem solch ein Attribut vorkommt. Beim Hinzufügen von Attributierungsregeln (Eigenschaften 2 und 3) werden in jedem Kontext genau so viele Attributierungsregeln hinzugefügt, wie Symbolvorkommen zu diesem Attribut in der Produktion vorhanden sind, d. h. genau eins für die linke Seite einer Produktion wenn das Attribut synthetisiert für dieses Symbol vorkommt. Für ererbte Attribute, dann für jedes Symbol der rechten Seite einer Produktion in der das Attribut zu dem Symbol ererbt vorkommt. Zusammen mit Eigenschaft 4 und 5 stellt dies dann sicher, dass keine Inkonsistenzen, Unvollständigkeit und Zyklen der Attributgrammatik hinzugefügt werden können. Die letzte Eigenschaft, Eigenschaft 6, stellt eine interne Konsistenz bzgl. den Änderungsmengen dar, sodass beim Hinzufügen von Attributierungsregeln, die definierten Attribute in der Attributgrammatik vorhanden sind.

Die Eigenschaften aus Satz 1 können mittels Fixpunktiteration gelöst werden. Für die Konstruktion der Zerlegung der Attributgrammatik werden die gelöschten Attribute aus den Partitionen entfernt und in einem weiteren Schritt für jedes hinzugefügte Attribut zwei neue Partitionen erstellt und ein einzelnes hinzugefügtes Attribut in eine der beiden Mengen einsortiert.

Die Beweise der einzelnen Eigenschaften werden in [3] so geführt, dass eine der folgenden Aussagen zum Widerspruch geführt wird: die Attributgrammatik ist zerlegbar und die Änderungsmengen halten die Eigenschaften von Satz 1 ein.

3.1 Musterkonstruktion

Die Bestimmung von Mustern – Änderungsmengen – erfolgt auf Basis verschiedener Arten von Termen, die, sortentreu, mit Attributen und Termen aus der Attributgrammatik, für die das Muster angewendet werden soll, substitutiert werden. Zur Auswahl der zu attributierenden Produktionen und Symbole, bzw. zur Auswahl der zu entfernenden Attribute und Regeln werden Prädikate genutzt.

Definition 13. *Seien* a_i*,* $0 \leq i \leq n, n \in \mathbb{N}$ *Variablen der Sorte* $\mathfrak{A}$ *– der Musterattribute,* $f \in \mathfrak{F}$ *eine Variable der Sorte der Musterfunktionen und* $c \in \mathfrak{C}$ *eine Variable der Sorte der Musterkonstanten. Ein* ***Attributwertterm*** $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ *ist induktiv wie folgt definiert:*

a) c *ist ein Attributwertterm;*
b) a_i *ist ein Attributwertterm und*
c) sind $t^{(0)}_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}, \cdots, t^{(n)}_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ *Attributwertterme, dann ist auch* $f(t^{(0)}_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}, \cdots, t^{(n)}_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}})$ *ein Attributwertterm.*

Bei der Subsitution wird im Folgenden davon ausgegangen, dass die Substitution sortentreu geschieht: Variablen der Sorte $\mathfrak{A}$ nur durch Attribute, Variablen aus $\mathfrak{F}$ nur durch Funktionen und Variablen der Sorte $\mathfrak{C}$ nur durch Konstanten subsitituiert werden. Bei der Implementierung mittels rein textuell arbeitender Ersetzungssysteme muss ggf. auf diese Einschränkung geachtet werden.

Im Folgenden werden Prädikate eingeführt, die dazu dienen, die Attributgrammatik in zu löschende und zu erweiternde Teile aufzuteilen. Darüber hinaus können mittels dieser Prädikate Eigenschaften der Attributgrammatik sichergestellt werden, sodass bei der Anwendung von Mustern das Resultat eine geordnete Attributgrammatik ist.

Definition 14. *Seien a_i für $0 \leq i \leq n$, $n \in \mathbb{N}$, a_i der Sorte der Musterattribute $\mathfrak{A}$, c, $true$, $false$ der Sorte der Konstanten $\mathfrak{C}$. Für Variablen erster Ordnung $a_i, c, true, false, f$, einen Term erster Ordnung $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ und Variablen $S_0, S_1, \cdots, S_m$ der Sorte der Symbolvariablen $\mathfrak{V}$ zweiter Ordnung ist ein* ***Prädikatterm*** *$t_{\{\mathcal{P}\}}$ induktiv wie folgt definiert:*

1. *$true$ und $false$ sind (konstante) Prädikatterme;*
2. *ein Attributwertterm $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ ist ein Prädikatterm (zweiter Ordnung);*
3. *für eine Variable a_j der Sorte $\mathfrak{A}$ ist $a_j \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ ein Prädikatterm;*
4. *Seien $t^{(0)}_{\{\mathcal{P}\}}$ und $t^{(1)}_{\{\mathcal{P}\}}$ Prädikatterme, so auch $t^{(0)}_{\{\mathcal{P}\}} \vee t^{(1)}_{\{\mathcal{P}\}}$, $t^{(0)}_{\{\mathcal{P}\}} \wedge t^{(1)}_{\{\mathcal{P}\}}$ und $t^{(0)}_{\{\mathcal{P}\}} = t^{(1)}_{\{\mathcal{P}\}}$;*
5. *Ist $t_{\{\mathcal{P}\}}$ ein Prädikatterm, so auch $\neg t_{\{\mathcal{P}\}}$*
6. *Seien $t^{(0)}_{\{\mathcal{P}\}}$ und $t^{(1)}_{\{\mathcal{P}\}}$ Prädikatterme, so auch $\forall t^{(0)}_{\{\mathcal{P}\}} \bullet t^{(1)}_{\{\mathcal{P}\}}$ und $\exists t^{(0)}_{\{\mathcal{P}\}} \bullet t^{(1)}_{\{\mathcal{P}\}}$;*
7. *$S_0 ::= S_1 \ \cdots \ S_n$ ist ein Prädikatterm, ebenso $p : S_0 ::= S_1 \ \cdots \ S_n$ für einen neuen Bezeichner p; statt einer Variablen kann auch der Platzhalter _ stehen; statt S_n kann der Platzhalter / stehen;*
8. *für eine Variable a der Sorte $\mathfrak{A}$ (Musterattributvariable) sind $a \in AS$ und $a \in AI$ sowie $a \in AS(S_1)$ und $a \in AI(S_1)$ für eine Variable S_1 der Sorte $\mathfrak{V}$;*

Prädikatterme der Form 1, 2, 3 heißen Prädikatterme erster Ordnung, Prädikatterme der Form 6, 7, 8 heißen Prädikatterme zweiter Ordnung und Prädikatterme der Form 4 und 5 heißen Prädikatterme erster oder zweiter Ordnung wenn $t_{\{\mathcal{P}\}}$, $t^{(0)}_{\{\mathcal{P}\}}$ oder $t^{(1)}_{\{\mathcal{P}\}}$ Prädikatterme erster oder zweiter Ordnung sind.

In der Definition des Prädikatterms wurde keine Rücksicht auf Klammerung und Präzedenzen getroffen. Im Folgenden werden die üblichen Klammerungsregeln und Präzedenzen der Operatoren (wie $\vee$, $\wedge$, $\forall$, usw.) verwendet. Auf die übliche Betrachtung freier und gebundener Variablen wird nicht weiter eingegangen.

Auf die Substitution von Prädikattermen und deren Auswertung wird an dieser Stelle nicht weiter eingegangen. Ein solches Prädikat, was bspw. eine Produktion darstellt, wird bzgl. einer Attributgrammatik zu *true*(wahr) ausgewertet, wenn es eine Subsitution der Variablen gibt, sodass eine Produktion in dieser Attributgrammatik existiert. Im Folgenden wird für die Auswertung eines Prädikats *pred* die Auswertung mittels $[\![pred]\!]_{AG}$ dargestellt. Für die Überprüfung ob ein Prädikat zu wahr oder falsch ausgewertet wurde, wird $\doteq$ verwendet – die übliche syntaktische Übereinstimmung zweier Terme wird dann mit dem üblichen Vergleich dargestellt.

Definition 15. *Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik. Seien weiterhin $Pr_+ = \{t^{(0,+)}_{\{\mathcal{P}\}}, \cdots, t^{(n,+)}_{\{\mathcal{P}\}}\}$ und $Pr_- = \{t^{(0,-)}_{\{\mathcal{P}\}}, \cdots, t^{(m,-)}_{\{\mathcal{P}\}}\}$ Mengen von Prädikattermen und σ_{AG} eine Substitution von Prädikattermen bzgl. AG. Das Paar (AG_+, AG_-) mit $AG_+ \triangleq (G_+, A_+, R_+, B)$ und $AG_- \triangleq (G_-, A_-, R_-, B)$ heißt* ***prädikatabhängige Teil-Attributgrammatik*** *(bzgl. AG und Pr_+ und Pr_-) genau dann, wenn die Anzahl der Elemente in $G_+, A_+, R_+, G_-, A_-, R_-$ minimal ist und folgende Bedingungen bzgl. dieser Mengen eingehalten werden, d. h. es gibt keine Menge, die dieselben Bedingungen einhält und weniger Elemente enthält:*

Für alle Prädikatterme $pred \in Pr_+$ und Substitution σ_{AG} mit $[\![pred]\!]_{AG}$ ist $AG_+ = (G_+, A_+, R_+, B)$ mit $G_+ = (N_+, T_+, P_+, Z_+)$ dann halten

$$\begin{aligned}
&\{\mathsf{X}_0 \colon \mathsf{X}_0 \in N \wedge \exists p \in P_+ : p = \mathsf{X}_0 ::= u, u \in \Sigma^* \wedge N_+ \setminus \{\mathsf{X}_0\} \implies [\![pred]\!]_{AG} \doteq false\} \\
\cup&\{\mathsf{Y} \colon \mathsf{Y} \in N, \exists p \in P_+, p = \mathsf{X} :: \ u \ \mathsf{Y} \ v, \mathsf{Y} \in N, u, v \in \Sigma^* \wedge N_+ \setminus \{\mathsf{Y}\} \implies [\![pred]\!]_{AG} \doteq false\} \\
&\subseteq N_+ \subseteq N
\end{aligned}$$

$$\begin{aligned}
&\{\mathsf{X} \colon \mathsf{X} \in T \wedge \exists p \in P_+ : p = \mathsf{Y} ::= u \ \mathsf{X} \ v, u, v \in \Sigma^*, \mathsf{Y} \in N_+ \wedge T_+ \setminus \{\mathsf{X}\} \implies [\![pred]\!]_{AG} \doteq false\} \\
&\subseteq T_+ \subseteq T
\end{aligned}$$

$$\begin{aligned}
&\{q \colon q \in P, q = \mathsf{X} ::= u, u \in \Sigma^*, \mathsf{X} \in N_+ \wedge \{P_+ \setminus \{q\} \implies [\![pred]\!]_{AG} \doteq false\} \\
\cup&\{p \colon p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*, \wedge(\exists q \in P_+ : q = \mathsf{X} ::= v, v \in \Sigma^* \vee a \in A_+ \wedge a \in AS(\mathsf{X})\} \\
\cup&\{p \colon p \in P, p = \mathsf{Y} ::= u \ \mathsf{X} \ v, u, v \in \Sigma^*, \mathsf{X} \in \Sigma, \mathsf{Y} \in N \wedge \exists a \in A_+, a \in AI(\mathsf{X})\} \\
\cup&\{p \colon p \in P \wedge \exists r \in R_+, r \in R_p \wedge R_+ \setminus \{r\} \implies [\![pred]\!]_{AG} \doteq false\} \\
&\subseteq P_+ \subseteq P
\end{aligned}$$

$$Z_+ \in N_+ \wedge \{\mathsf{X} \colon \mathsf{X} \in N_+ \wedge \mathsf{X} \overset{+}{\rightsquigarrow} Z_+\} = \emptyset \wedge \mathsf{X} \in N_+, \mathsf{X} \neq Z_+ \implies Z_+ \overset{+}{\rightsquigarrow} \mathsf{X}$$

$$\{b_i \colon \exists r = a \leftarrow f(b_1, \cdots, b_n) \in R_+ \vee (a \in A_+ \wedge r \in R) \wedge (R_+ \setminus \{r\} \implies [\![pred]\!]_{AG} \doteq false \vee A_+ \setminus \{a\} \implies [\![pred]\!]_{AG} \doteq false)\} \subseteq A_+ \subseteq A$$

$$\{r \colon r = a \leftarrow f(\cdots), a \in A_+ \wedge R_+ \setminus \{r\} \implies [\![pred]\!]_{AG} \doteq false\} \subseteq R_+ \subseteq R$$

ein. Analoges gilt für die jeweiligen G_-, A_-, R_-, sowie N_-, T_-, P_-, Z_-.

Über eine Fixpunktiteration lassen sich die Teil-Attributgrammatiken nach Def. 15 bestimmen. Resultat sind dann konsistente, vollständige Teil-Attributgrammatiken, auf die Muster anzuwenden sind. Im Folgenden werden nun Attributwertterme erweitert; diese erweiterten Attributwertterme stellen bereits implizit Basismuster dar.

Definition 16. *Sei a eine Variable der Sorte der Musterattribute $\mathfrak{A}$, $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ ein Attributwertterm, $S_0, S_1, \cdots, S_n$ Variablen der Sorte der Symbolvariablen $\mathfrak{V}$. Ein Term $t_{\{\mathfrak{P}\}}$ heißt **erweiterter Attributwertterm** genau dann, wenn $t_{\{\mathfrak{P}\}}$ eine der folgenden Formen hat*

1. **symbol** S_0 **attr** $\uparrow.a \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$
2. **symbol** S_0 **attr** $\downarrow.a \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$
3. **rule** $S_0 ::= S_1 \ \cdots \ S_n$ **attr** $S_j.a \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$

   wobei $0 \leq j \leq n$ ist.

Darauf aufbauend lässt sich eine Attributgrammatik-unabhängige Musterdefinition erstellen:

Definition 17. *Das Tupel $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ heißt **Attributgrammatik-unabhängige Musterdefinition** wobei*

- *$V = \mathfrak{V} \uplus \mathfrak{A} \uplus \mathfrak{C} \uplus \mathfrak{F}$ die Menge der Variablen;*
- *$Pr = Pr_+ \cup Pr_-$ die Menge der Prädikate unter Verwendung der Variablen aus V sind und*
- *$At = At_+ \cup At_-$ die Menge erweiterter Attributwertterme über Variablen der Sorten aus V sind.*

Mittels Substitution werden aus einer Attributgrammatik-unabhängigen Musterdefinition prädikatabhängige Teil-Attributgrammatiken erzeugt.

Definition 18. *Sei $S \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ eine Menge von Substitutionen, $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Eine Substitution $\sigma_i = \left[t_{i_1}/v_{i_1} \cdots t_{i_m}/v_{i_m}\right]$, $0 \leq i \leq n$ heißt **passende Substitution** zu AG und $\mathcal{M}_u$ genau dann, wenn für alle Ersetzungen t_{i_j}/v_{i_j} mit $1 \leq j \leq m, m \in \mathbb{N}$ folgende Eigenschaften erfüllt sind:*

1. *$v_{i_j} \in V$, v_{i_j} der Sorte Sort*
2. *t_{i_j} ist ein Wort über dem Alphabet $\Sigma \triangleq N \cup T \cup \{::=, _, /, \leftarrow, \varepsilon\} \cup A \cup F \cup A'$, wobei F eine Menge an Operationssymbolen ist und $A' \cap A = \emptyset$ und t_{i_j} der Sorte Sort;*
3. *$v_{i_k} = v_{i_l} \implies t_{i_k} = t_{i_l}, 0 \leq k \leq m, 0 \leq l \leq m$.*

*σ_i heißt **anwendbare Substition** auf $t_{\{\mathcal{P}\}}$ oder anwendbare Substitution auf $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$, wenn sie passend ist und mindestens eine der Variablen v_{i_j} in $t_{\{\mathcal{P}\}}$ oder $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ vorkommt.*

*Eine Menge Subs heißt **Menge passender Substitutionen** genau dann, wenn jedes $\sigma_i \in Subs$ eine passende Substitution ist und falls $v_{i_j} = v_{k_l}$ dann auch $t_{i_j} = t_{k_l}$ mit $0 \leq i \leq n, 0 \leq j \leq m$ und $0 \leq k \leq n, 0 \leq l \leq o, o \in \mathbb{N}$.*

Eine weitere Definition erweitert Definition 18 um die Anwendung einer Menge passender Substitutionen auf mehreren Termen.

Definition 19. *Sei* $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ *eine Attributgrammatik-unabhängige Musterdefinition und* $AG \triangleq (G, A, R, B)$ *eine zerlegbare Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$*. Die* ***Semantik der Anwendung der Menge passender Substitutionen*** $Subs \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ *auf einem Term* t *ist definiert als* $[\![t\ Subs]\!] \triangleq \{t\sigma_{\pi(1)} \cdots \sigma_{\pi(n)} : \pi \colon \{1, \cdots, n\} \to \{1, \cdots, n\}$ *ist Permutation*$\}$ *wobei* $\sigma_{\pi_i} \in Subs$ *für alle Permutationen* $\pi_1, \cdots, \pi_n$ *von* 1 *bis* n*, wenn* $\pi_i = \pi_j$ *dann auch* $i = j$*,* $1 \leq i \leq n$*,* $1 \leq j \leq n$*.*

3.2 Mustereigenschaften und Musteranwendung

Mit den bisherigen Definitionen lässt sich die Anwendung von Mustern definieren:

Definition 20. *Sei* $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ *eine Attributgrammatik-unabhängige Musterdefinition mit* $Pr = Pr_+ \cup Pr_-$ *und* $AG \triangleq (G, A, R, B)$ *eine zerlegbare Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$*, sowie der prädikatabhängigen Teil-Attributgrammatik* (AG_+, AG_-) *mit* $AG_+ \triangleq (G_+, A_+, R_+, B)$ *und* $AG_- \triangleq (G_-, A_-, R_-, B)$*, wobei* $G_+ \triangleq (N_+, T_+, P_+, Z_+)$ *und* $G_- \triangleq (N_-, T_-, P_-, Z_-)$*. Sei* $Subs \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ *eine Menge passender Substitutionen zu* AG *und* $\mathcal{M}_u$*. Das Tupel* $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$*, wobei* $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ *und* $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ *heißt* ***Musteranwendung*** *bzgl.* AG *und* $\mathcal{M}_u$ *genau dann, wenn sich* $\mathcal{M}_{+,A}, \mathcal{M}_{+,R}, \mathcal{M}_{-,A}$ *und* $\mathcal{M}_{-,R}$ *mit folgenden Regeln bilden:*

1. *Die Menge* $\mathcal{M}_{+,A}$ *sind alle Attribute für alle variablenfreien, aus erweiterten Attributwerttermen aus* At_+*, durch passende Substitutionen, erzeugten Attribute für jede der möglichen Formen des erweiterten Attributwertterms:*

$$
\begin{aligned}
\mathcal{M}_{+,A} = \{a \colon\ & a \in [\![a'\ Subs]\!]_{AG}, a \text{ variablenfrei } \wedge a \notin A_+ \wedge t^{(i)}_{\{\mathfrak{P}\}} \in At_+ \text{ mit} \\
& t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \uparrow .a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \vee \\
& t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \downarrow .a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \\
& f(b_1, \cdots, b_d) \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG} \text{ mit } b_g \in \mathcal{M}_{+,A} \cup A_+, 1 \leq g \leq d, d \in \mathbb{N} \\
& f \text{ variablenfrei}, \exists X \in [\![S_0\ Subs]\!]_{AG} \wedge X \in \Sigma\} \\
\cup \{a \colon\ & a \in [\![S_j.a'\ Subs]\!]_{AG} \wedge a \text{ variablenfrei } \wedge t^{(i)}_{\{\mathfrak{P}\}} \in At_+ \text{ mit} \\
& t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1\ \cdots\ S_n \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \\
& f(b_1, \cdots, b_d) \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG} \text{ mit } b_g \in \mathcal{M}_{+,A} \cup A_+, 1 \leq g \leq d, d \in \mathbb{N}\}
\end{aligned}
$$

2. *Die Menge* $\mathcal{M}_{-,A}$ *sind alle Attribute für alle variablenfreien, aus erweiterten Attributwerttermen aus* At_-*, durch passende Substitutionen, erzeugten Attribute für jede der möglichen Formen des erweiterten Attributwertterms:*

$$
\begin{aligned}
\mathcal{M}_{-,A} = \{a \colon\ & a \in [\![a'\ Subs]\!]_{AG}, a \text{ variablenfrei } \wedge a \in A_- \wedge t^{(i)}_{\{\mathfrak{P}\}} \in At_- \text{ mit} \\
& t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \uparrow .a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \vee \\
& t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \downarrow .a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \\
& \exists X \in [\![S_0\ Subs]\!]_{AG} \wedge X \in \Sigma\} \\
\cup \{a \colon\ & a \in [\![S_j.a'\ Subs]\!]_{AG} \wedge a \text{ variablenfrei } \wedge t^{(i)}_{\{\mathfrak{P}\}} \in At_- \text{ mit} \\
& t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1\ \cdots\ S_n \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}\} \\
\subseteq A
\end{aligned}
$$

3. *Die Menge $\mathcal{M}_{+,R,s}$ sind alle Attributierungsregeln für alle variablenfreien, aus erweiterten Attributwerttermen aus At_+, durch passende Substitutionen, erzeugten Attributierungsregeln deren Resultat synthetisierte Attributierungsregeln sind:*

$$\begin{aligned}\mathcal{M}_{+,R,s} = \{&r\colon r = \ \textbf{rule}\ \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n\ \textbf{attr}\ \mathsf{X}_0.a \leftarrow t'\ \textit{und}\ \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_+\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol}\ S_0\ \textbf{attr}\ \uparrow.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}} \wedge\\ &\mathsf{X}_0 \in [\![S_0\ Subs]\!]_{AG} \wedge \exists p \in P_+, p = \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n \wedge\\ &a \in [\![a'\ Subs]\!]_{AG} \wedge t' \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG} \wedge a \in \mathcal{M}_{+,A}, \mathsf{X}_0 \in N,\\ &\mathsf{X}_k \in \Sigma_+, 1 \le k \le n,\ a, t'\ \textit{variablenfrei}\}\\ \cup\{&r\colon r = \ \textbf{rule}\ \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n\ \textbf{attr}\ \mathsf{X}_0.a \leftarrow t'\ \textit{und}\ \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_+\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule}\ S_0 ::= S_1\ \cdots\ S_m\ \textbf{attr}\ S_j.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ \textit{mit}\\ &\mathsf{X}_0 \in [\![S_0\ Subs]\!]_{AG}, \mathsf{X}_l \in [\![S_k\ Subs]\!]_{AG}, 1 \le l \le n, 1 \le k \le m, m \ge n,\\ &\exists p \in P_+, p = \mathsf{X} ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n \wedge \mathsf{X}_0.a \in [\![S_j.a']\!]_{AG}, t' \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG},\\ &\mathsf{X}_0.a \in \mathcal{M}_{+,A}, \mathsf{X}_0 \in N_+, \mathsf{X}_l \in \Sigma_+, \mathsf{X}_0.a, t'\ \textit{variablenfrei}\}\end{aligned}$$

4. *Die Menge $\mathcal{M}_{+,R,i}$ sind alle Attributierungsregeln für alle variablenfreien, aus erweiterten Attributwerttermen aus At_+, durch passende Substitutionen, erzeugten Attributierungsregeln deren Resultat ererbte Attributierungsregeln sind:*

$$\begin{aligned}\mathcal{M}_{+,R,i} = \{&r\colon r^{(k)} = \ \textbf{rule}\ \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_j\ \cdots\ \mathsf{X}_n\ \textbf{attr}\ \mathsf{X}^{(k)}_j.a \leftarrow t'\ \textit{und}\ \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_+\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol}\ S_0\ \textbf{attr}\ \downarrow.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}} \wedge\\ &\mathsf{X}_j \in [\![S_0\ Subs]\!]_{AG} \wedge \exists p \in P_+, p = \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_j\ \cdots\ \mathsf{X}_n \wedge\\ &a \in [\![a'\ Subs]\!]_{AG} \wedge t' \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG} \wedge a \in \mathcal{M}_{+,A}, \mathsf{X}_0 \in N,\\ &\mathsf{X}_v \in \Sigma_+, 1 \le v \le n,\ a, t\ \textit{variablenfrei}, 1 \le k \le \left|\mathsf{X}_j\right|_p\}\\ \cup\{&r\colon r = \ \textbf{rule}\ \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_h\ \cdots\ \mathsf{X}_n\ \textbf{attr}\ \mathsf{X}_j.a \leftarrow t'\ \textit{und}\ \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_+\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule}\ S_0 ::= S_1\ \cdots\ S_m\ \textbf{attr}\ S_j.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ \textit{mit}\\ &\mathsf{X}_0 \in [\![S_0\ Subs]\!]_{AG}, \mathsf{X}_l \in [\![S_k\ Subs]\!]_{AG}, 1 \le l \le n, 1 \le k \le m, m \ge n,\\ &\mathsf{X}_h \in [\![S_j\ Subs]\!]_{AG} \exists p \in P_+, p = \mathsf{X} ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n \wedge \mathsf{X}_h.a \in [\![S_j.a']\!]_{AG},\\ &t' \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG}, \mathsf{X}_h.a \in \mathcal{M}_{+,a}, \mathsf{X}_0 \in N_+, \mathsf{X}_l \in \Sigma_+, \mathsf{X}_0.a, t'\ \textit{variablenfrei}\}\end{aligned}$$

5. *Die Menge $\mathcal{M}_{-,R,s}$ sind alle Attributierungsregeln für alle variablenfreien, aus erweiterten Attributwerttermen aus At_-, durch passende Substitutionen, erzeugten Attributierungsregeln deren Resultat synthetisierte Attributierungsregeln sind:*

$$\begin{aligned}\mathcal{M}_{-,R,s} = \{&r\colon r = \ \textbf{rule}\ \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n\ \textbf{attr}\ \mathsf{X}_0.a \leftarrow t'\ \textit{und}\ \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_-\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol}\ S_0\ \textbf{attr}\ \uparrow.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}} \wedge\\ &\mathsf{X}_0 \in [\![S_0\ Subs]\!]_{AG} \wedge \exists p \in P_-, p = \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n \wedge\\ &a \in [\![a'\ Subs]\!]_{AG} \wedge t' \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG} \wedge a \in \mathcal{M}_{-,A}, \mathsf{X}_0 \in N,\\ &\mathsf{X}_k \in \Sigma_-, 1 \le k \le n,\ a, t\ \textit{variablenfrei}\}\\ \cup\{&r\colon r = \ \textbf{rule}\ \mathsf{X}_0 ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n\ \textbf{attr}\ \mathsf{X}_0.a \leftarrow t'\ \textit{und}\ \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_-\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule}\ S_0 ::= S_1\ \cdots\ S_m\ \textbf{attr}\ S_j.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ \textit{mit}\\ &\mathsf{X}_0 \in [\![S_0\ Subs]\!]_{AG}, \mathsf{X}_l \in [\![S_k\ Subs]\!]_{AG}, 1 \le l \le n, 1 \le k \le m, m \ge n,\\ &\exists p \in P_-, p = \mathsf{X} ::= \mathsf{X}_1\ \cdots\ \mathsf{X}_n \wedge \mathsf{X}_0.a \in [\![S_j.a']\!]_{AG}, t' \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG},\\ &\mathsf{X}_0.a \in \mathcal{M}_{-,a}, \mathsf{X}_0 \in N_-, \mathsf{X}_l \in \Sigma_-, \mathsf{X}_0.a, t'\ \textit{variablenfrei}\}\end{aligned}$$

6. *Die Menge $\mathcal{M}_{-,R,i}$ sind alle Attributierungsregeln für alle variablenfreien, aus erweiterten Attributwerttermen aus At_-, durch passende Substitutionen, erzeugten Attributierungsregeln deren Resultat ererbte Attributierungsregeln sind:*

$$\begin{aligned}
\mathcal{M}_{R,i,-} = \{r\colon r^{(k)} = \ & \textbf{rule } \mathsf{X}_0 ::= \mathsf{X}_1 \ \cdots \ \mathsf{X}_j \ \cdots \ \mathsf{X}_n \textbf{ attr } \mathsf{X}_j^{(k)}.a \leftarrow t' \textit{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_- \\
& t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \downarrow.a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \wedge \\
& \mathsf{X}_j \in [\![S_0 \ Subs]\!]_{AG} \wedge \exists p \in P_-, p = \mathsf{X}_0 ::= \mathsf{X}_1 \ \cdots \ \mathsf{X}_j \ \cdots \ \mathsf{X}_n \wedge \\
& a \in [\![a' \ Subs]\!]_{AG} \wedge t' \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \ Subs]\!]_{AG} \wedge a \in \mathcal{M}_{-,A}, \mathsf{X}_0 \in N, \\
& \mathsf{X}_v \in \Sigma_-, 1 \leq v \leq n, \ a,t \textit{ variablenfrei}, 1 \leq k \leq \left|\mathsf{X}_j\right|_p\} \\
\cup \{r\colon r = \ & \textbf{rule } \mathsf{X}_0 ::= \mathsf{X}_1 \ \cdots \ \mathsf{X}_h \ \cdots \ \mathsf{X}_n \textbf{ attr } \mathsf{X}_j.a \leftarrow t' \textit{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_- \\
& t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1 \ \cdots \ S_m \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \textit{ mit} \\
& \mathsf{X}_0 \in [\![S_0 \ Subs]\!]_{AG}, \mathsf{X}_l \in [\![S_k \ Subs]\!]_{AG}, 1 \leq l \leq n, 1 \leq k \leq m, m \geq n, \\
& \mathsf{X}_h \in [\![S_j \ Subs]\!]_{AG} \exists p \in P_-, p = \mathsf{X} ::= \mathsf{X}_1 \ \cdots \ \mathsf{X}_n \wedge \mathsf{X}_h.a \in [\![S_j.a']\!]_{AG}, \\
& t' \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \ Subs]\!]_{AG}, \mathsf{X}_h.a \in \mathcal{M}_{-,a}, \mathsf{X}_0 \in N_-, \mathsf{X}_l \in \Sigma_-, \mathsf{X}_0.a, t' \textit{ variablenfrei}\}
\end{aligned}$$

7. *für alle $r \in \mathcal{M}_{+,R} = \mathcal{M}_{R,s,+} \cup \mathcal{M}_{+,R,i}$, ist $R_+ \cup \{r\}$ lokal azyklisch*

mit $\Sigma_+ = T_+ \uplus N_+$ und $\Sigma_- = T_- \uplus N_-$ und $\mathcal{M}_{-,R} = \mathcal{M}_{-,R,s} \cup \mathcal{M}_{-,R,i}$,

Durch Verwendung von Definition 19 lassen sich aus wenigen Prädikaten und wenigen erweiterten Attributwerttermen große Mengen von Attributierungsregeln herleiten.

Die Herstellung der für Satz 1 notwendigen Eigenschaften lässt sich in der Praxis leicht erreichen. Für diese Arbeit wichtig ist dieser Prozess nicht, lediglich die damit einhergende Definition:

Definition 21. *Sei $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition mit $Pr = Pr_+ \cup Pr_-$ und $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$, sowie der prädikatabhängigen Teil-Attributgrammatik (AG_+, AG_-) mit $AG_+ \triangleq (G_+, A_+, R_+, B)$ und $AG_- \triangleq (G_-, A_-, R_-, B)$, wobei $G_+ \triangleq (N_+, T_+, P_+, Z_+)$ und $G_- \triangleq (N_-, T_-, P_-, Z_-)$. Sei $Subs \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ eine Menge passender Substitutionen zu AG und $\mathcal{M}_u$. Das Tupel $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ ist. Halten die Mengen nach $\mathcal{M}_u$ die Eigenschaften von Satz 1 ein, wobei $\mathsf{A}_- = \mathcal{M}_{-,A}$, $\mathsf{A}_+ = \mathcal{M}_{+,A}$, $\mathsf{R}_- = \mathcal{M}_{-,R}$ und $\mathsf{R}_+ = \mathcal{M}_{+,R}$ so heißt $\mathcal{M}_u$ mit den Substitutionen $Subs$ **schlichte Musteranwendung**.*

Interessant hingegen ist, wenn die Mengen vorhanden sind, welche Eigenschaften diese einhalten müssen, sodass eine geordnete Attributgrammatik in eine geordnete Attributgrammatik überführt wird. Die Beweise folgender Lemma sind [3] zu entnehmen.

Lemma 1. *Sei $\mathcal{AG}_G$ die Menge aller Attributgrammatiken mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik, $\mathcal{M}_G\colon \mathcal{AG} \to \mathcal{AG}$ ein zerlegungserhaltendes Muster und $AG' \triangleq \mathcal{M}_G(AG)$ die resultierende Attributgrammatik. Ist $\mathcal{M}_G$ ordnungserhaltend, so ist $\mathcal{M}_G$ auch zerlegungserhaltend.*

Lemma 2. *Sei $AG \in \mathcal{AG}_G$ eine geordnete Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und ein Muster durch die Änderungsmengen A_-, A_+, R_- und R_+ gegeben mit $\mathsf{A}_+ = \mathsf{R}_+ = \emptyset$, sodass $AG' \triangleq (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-)$ und $R' = (R \setminus \mathsf{R}_-)$ ist, wobei die Änderungsmengen die Eigenschaften von Satz 1 einhalten.*

Für jede Produktion $p \in P$ und jedes Symbol $\mathsf{X} \in \Sigma$ sind dann die durch IDP'_p und IDS'_{X} definierten Graphen azyklisch, wobei IDP'_p die induzierten Abhängigkeiten zwischen Attributvorkommen der resultierenden Attributgrammatik AG' sind, analog IDS'_{X} die induzierten Abhängigkeiten zwischen Symbolattributen der resultierenden Attributgrammatik sind. Der erweiterte Abhängigkeitsgraph $EDP'_p \triangleq IDP'_p \cup \{\mathsf{X}.a \to \mathsf{X}.b : \exists h, k \in \mathbb{N}$ mit $\mathsf{X}.a \in A'_{\mathsf{X}}(h) \wedge \mathsf{X}.b \in A'_{\mathsf{X}}(k) \wedge h < k\}$ ist ebenfalls azyklisch.

Lemma 3. *Sei $AG \in \mathcal{AG}_G$ eine geordnete Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und ein Muster durch die Änderungsmengen A_-, A_+, R_- und R_+ gegeben, mit $\mathsf{R}_- = \mathsf{A}_- = \emptyset$, sodass $AG' \triangleq (G, A', R', B)$ mit $A' = A \cup \mathsf{A}_+$ und $R' = R \cup \mathsf{R}_+$ ist, wobei die Änderungsmengen die Eigenschaften von Satz 1 einhalten und für alle $r_i, r_j \in \mathsf{R}_{+,a}$ mit $\mathsf{R}_{+,a} = \{r \colon r \in \mathsf{R}_+ \text{ und } r = a \leftarrow f(b_1, \cdots, b_n)\}$ dann ist $r_i = a \leftarrow f(b_{i_1}, \cdots, b_{i_n})$ und $r_j = a \leftarrow f(b_{j_1}, \cdots, b_{j_m})$ und $\{b_{i_1}, \cdots, b_{i_n}\} = \{b_{j_1}, \cdots, b_{j_m}\}$ mit $\left|\{b_{i_1}, \cdots, b_{i_n}\}\right| = \left|\{b_{j_1}, \cdots, b_{j_m}\}\right| = \left|\{b_1, \cdots, b_n\}\right|$*[2]

Für jede Produktion $p \in P$ und jedes Symbol $\mathsf{X} \in \Sigma$ sind dann die durch IDP'_p und IDS'_X definierten Graphen azyklisch, wobei IDP'_p die induzierten Abhängigkeiten zwischen Attributvorkommen der resultierenden Attributgrammatik AG' sind, analog IDS'_X die induzierten Abhängigkeiten zwischen Symbolattributen der resultierenden Attributgrammatik sind. Der erweiterte Abhängigkeitsgraph $EDP'_p \triangleq IDP'_p \cup \{\mathsf{X}.a \to \mathsf{X}.b : \exists h, k \in \mathbb{N} \text{ mit } \mathsf{X}.a \in A'_\mathsf{X}(h) \wedge \mathsf{X}.b \in A'_\mathsf{X}(k) \wedge h < k\}$ ist ebenfalls azyklisch.

Obwohl Lemma 3 Hinzufügen das Hinzufügen von Attributierungsregeln erlaubt ist eine Änderung bei bestehenden Attributierungsregeln gesondert zu behandeln, da Hinzufügen von Attributen als Abhängigkeit bei bestehenden Attributierungsregeln schnell zu Zyklen in den induzierten Abhängigkeiten führen kann.

Lemma 4. *Sei $AG \in \mathcal{AG}_G$ eine geordnete Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und ein Muster durch die Änderungsmengen A_-, A_+, R_- und R_+ gegeben, sodass $AG' \triangleq (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$ und $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ ist, wobei die Änderungsmengen die Eigenschaften von Satz 1 einhalten und für alle $r_i, r_j \in \mathsf{R}_{+,a}$ mit $\mathsf{R}_{+,a} = \{r \colon r \in \mathsf{R}_+ \text{ und } r = a \leftarrow f(b_1, \cdots, b_n)\}$ dann ist $r_i = a \leftarrow f(b_{i_1}, \cdots, b_{i_n})$ und $r_j = a \leftarrow f(b_{j_1}, \cdots, b_{j_n})$ und $\{b_{i_1}, \cdots, b_{i_n}\} = \{b_{j_1}, \cdots, b_{j_n}\}$. Für $r_i \in \mathsf{R}_-$ mit $r_i = a \leftarrow f(b_{i_1}, \cdots, b_{i'_n})$ ist dann $\{b_{i_1}, \cdots, b_{i'_n}\} \subseteq \{b_{i_1}, \cdots, b_{i_n}\}$ wobei $n' \leq n$, $n', n \in \mathbb{N}$. Dabei gilt $b_{i_j} \in A$ für alle $i \in \mathbb{N}$ und $i_1, \cdots, i_n, i'_n \in \mathbb{N}$ und für alle $r \in (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ ist $r \neq b \leftarrow f(\cdots, a, \cdots)$.*

Für jede Produktion $p \in P$ und jedes Symbol $\mathsf{X} \in \Sigma$ sind dann die durch IDP'_p und IDS'_X definierten Graphen azyklisch, wobei IDP'_p die induzierten Abhängigkeiten zwischen Attributvorkommen der resultierenden Attributgrammatik AG' sind, analog IDS'_X die induzierten Abhängigkeiten zwischen Symbolattributen der resultierenden Attributgrammatik sind. Der erweiterte Abhängigkeitsgraph $EDP'_p \triangleq IDP'_p \cup \{\mathsf{X}.a \to \mathsf{X}.b : \exists h, k \in \mathbb{N} \text{ mit } \mathsf{X}.a \in A'_\mathsf{X}(h) \wedge \mathsf{X}.b \in A'_\mathsf{X}(k) \wedge h < k\}$ ist ebenfalls azyklisch.

Somit sind die Eigenschaften und wesentlichen Aussagen gezeigt, um ausgehend von 21 einge schlichte geordnete Musteranwendung zu definieren.

Definition 22. *Sei $\mathcal{M}_u \triangleq (V, Pr, At)$ eine Attributgrammatik-unabhängige Musterdefinition mit $Pr = Pr_+ \cup Pr_-$ und $AG \triangleq (G, A, R, B)$ eine geordnet Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$, sowie der prädikatabhängigen Teil-Attributgrammatik (AG_+, AG_-) mit $AG_+ \triangleq (G_+, A_+, R_+, B)$ und $AG_- \triangleq (G_-, A_-, R_-, B)$, wobei $G_+ \triangleq (N, T, P_+, Z)$ und $G_- \triangleq (N, T, P_-, Z)$. Sei $S \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ eine Menge passender Substitutionen zu AG und $\mathcal{M}_u$. Das Tupel $\Delta_\mathcal{M} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ ist. Halten die Mengen nach $\mathcal{M}_u$ die Eigenschaften von Satz 1, die Vorraussetzungen von Lemma 3, Lemma 2 und Lemma 4 ein, wobei $\mathsf{A}_- = \mathcal{M}_{-,A}$, $\mathsf{A}_+ = \mathcal{M}_{+,A}$, $\mathsf{R}_- = \mathcal{M}_{-,R}$ und $\mathsf{R}_+ = \mathcal{M}_{+,R}$ so heißt $\mathcal{M}_u$ mit den Substitutionen Subs* ***schlichte geordnete Musteranwendung****.*

Zur Anwendung von Mustern ist dann nur folgender Satz zu zeigen:

Theorem 2. *Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G und $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition sowie einer Menge passender Substitutionen Subs. Für jede Musteranwendung $\Delta_\mathcal{M} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$ bzgl. AG und $\mathcal{M}_u$ und Subs mit $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ existiert eine schlichte Musteranwendung, wobei $\mathsf{A}_+ = \mathcal{M}_{+,A}$, $\mathsf{R}_+ = \mathcal{M}_{+,R}$, $\mathsf{A}_- = \mathcal{M}_{-,A}$ und $\mathsf{R}_- = \mathcal{M}_{-,R}$ sind.*

Für jede Eigenschaft von Satz 1 wird in [3] gezeigt, dass für jede Musteranwendung eine schlichte Musteranwendung existiert, für die die Mengen die Eigenschaften einhalten. Analog beschreiben die in Definition 22 genannten Lemmata die Eigenschaften, die darüber hinaus gelten müssen, sodass ein Muster auch ordnungserhaltend ist.

[2] In den neu hinzugefügten Regeln können die Attribute beliebig permutiert und beliebig oft vorkommen.

4 Evaluation und Anwendungen aus den Beweisen

In [3] wurde berichtet, wie für die Sprache aus [10] eine Implementierung mittels Paradigmen auf geordneten Attributgrammatiken erstellt wird. Die Lösung unter Verwendung von Paradigmen benötigt 40 Zeilen an Spezifikation, die Lösung ohne Verwendung von Mustern 1149. Weniger extrem sind die Werte für andere Beispiele, wenngleich der Faktor durchschnittlich ca. 11 ist.

Eine Anwendung, die sich aus den Beweisen ergibt, ist die iterative Konstruktion von geordneten Attributgrammatiken ohne Muster, bspw. wenn nur ein Teil der Muster zur Verfügung steht. Hinzufügen einfacher Attributierungsregeln, die keine bestehenden Attribute „überschreiben“ und nicht absichtlich so konstruiert sind, dass EDP_p zyklisch wird – siehe dazu bspw. die Arbeiten [7] und [9]. Dies wird durch weitere Einschränkungen (u. a. gleiche Abhängigkeiten in unterschiedlichen Kontexten) verhindert. Auch damit ist ein weiterer, natürlicher Hinweis zur iterativen Erweiterung bestehender geordneter Attributgrammatiken gefunden. Darüber hinaus gibt [3] eine Übersicht über die Implementierung von Mustern in geordneten Attributgrammatiken, sodass diese auch ohne ein dediziertes Werkzeug genutzt werden können, wenngleich dann der wesentliche Vorteil – kompaktere Spezifikation – nicht nutzbar ist.

Weitere Informationen, die den Umfang dieser Arbeit weit überschreiten, können [3] entnommen werden.

5 Zusammenfassung und Auswertung

Im Rahmen dieser Arbeit wurden die Eigenschaften, die Änderungsmengen auf Basis von Mustern oder Paradigmen einhalten müssen, vorgestellt. Neben den vorgestellten Eigenschaften, wurde angegeben, dass bei Einhalt dieser Eigenschaften, die resultierende Attributgrammatik auch wieder geordnet ist.

Ähnlich der Arbeit [27] wurde eine Methode zur Formalisierung des Aufbaus von Attributgrammatiken vorgestellt. Im Gegensatz zu der Arbeit von Lämmel und Riedewald waren die geprüften Kriterien in der vorliegenden Arbeit die Zerlegbarkeit und die Eigenschaft der Ordnung (im Sinne von OAG) für die Paradigmen. Neben den bereits auch in [29] vorgestellten Mustern können in dieser Arbeit die bereits in [5] und [6] vorgestellten Muster in dieses Rahmenwerk eingebettet werden, wenngleich die Vorstellung selbst nicht im Rahmen dieser Arbeit erfolgt.

Weitere Arbeiten, die sich mit der Reduktion des Spezifikationsumfangs von Attributgrammatiken beschäftigen sind Arbeiten auf Basis von Modulsystemen, wie [17,13,14], wobei keine Garantien bzgl. der zusammengesetzten Attributgrammatik formuliert werden. Darüber hinaus gibt es Referenzattributgrammatiken, wie [15,8], für die allerdings unentscheidbar ist, ob eine berechenbare Reihenfolge für jeden abstrakten Syntaxbaum existiert [8]. Ähnliche Argumente können für Attributgrammatiken mit Termersetzung, wie [12,23], gefunden werden, wenngleich diese geeignet sind Muster zu implementieren. Weitere Ansätze verwenden Bibliotheken oder dedizierte DSLs zur Verringerung des Spezifikationsumfangs, wie bspw. [21,22,25]. Letzerer Arbeiten sind insofern mit der vorliegenden Arbeit nicht vergleichbar, da eine Untersuchung in Bezug auf die Eigenschaften der Attributgrammatik nicht möglich sind.

Offen geblieben ist die Vorstellung der Muster im Rahmen der Definitionen dieser Arbeit sowie die Beschreibung der Komposition von (geordneten) Attributgrammatiken. Jedoch wurden eine Reihe der Erkenntnisse aus [3] in Abschnitt 4 wiedergegeben. Ebenfalls in dieser Arbeit offen geblieben ist die Verwendung anderer Arten von Attributgrammatiken – können Referenzattributgrammatiken oder Attributgrammatiken höherer Ordnung von Mustern profitieren und wie schlagen sich diese im Vergleich zu geordneten Attributgrammatiken mit Mustern in Bezug auf Performanz sowie Spezifikationsumfang.

Ein ebenso wichtiger und umfangreicher offener Bereich sind die Fragestellungen bzgl. der Parallelisierung von (geordneten) Attributgrammatiken oder die Implementierung parallelisierbarer Algorithmen mittels dieser parallelen Attributgrammatiken, die in Ansätzen bereits in [30] vorgestellt wurde.

Bisher nicht vorgestellte Muster sind gerade im Bereich bidirektionaler Transformationen und des angrenzenden projezierenden Editierens zu erwarten. Darüber hinaus existieren Muster bzw. Kombinationen von Mustern, die nicht in einer geordneten Attributgrammatik münden. Hier beschränkt sich die existierende Literatur auf einfache, künstliche Beispiele obwohl praktisch relevante Variationen der Namensanalyse mit Baumtransformationen unmittelbar außerhalb bestehender Musterdefinition sind.

Literatur

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
2. Albreshne, A., Pasquier, J.: A domain specific language for high-level process control programming in smart buildings. Procedia Computer Science **63**, 65 – 73 (2015), https://doi.org/http://dx.doi.org/10.1016/j.procs.2015.08.313http://dx.doi.org/10.1016/j.procs.2015.08.313, http://www.sciencedirect.com/science/article/pii/S1877050915024412
3. Berg, C.: Typische Muster bei der DSL-Implementierung unter Verwendung geordneter Attributgrammatiken. Dissertation, Martin-Luther-Universität Halle-Wittenberg (2019), http://nbn-resolving.de/urn:nbn:de:gbv:3:4-24518
4. Berg, C., Zimmermann, W.: Evaluierung von Möglichkeiten zur Implementierung von Semantischen Analysen für Domänenspezifische Sprachen (2014)
5. Berg, C., Zimmermann, W.: Typische Muster bei der Entwicklung Dom nen-spezifischer Sprachen mit Attributgrammatiken. In: Knoop, J., Ertl, M.A. (eds.) 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung, pp. 28 – 43. No. Bericht 2015-IX-1 in Schriftenreihe des Instituts für Computersprachen (2015)
6. Berg, C., Zimmermann, W.: Eigenschaften typischer Muster auf geordneten Attributgrammatiken. In: Amme, W., Heinze, T.S. (eds.) Programmiersprachen und Grundlagen der Programmierung 19. Kolloquium, KPS 2017, pp. 29 – 44. No. Bericht Math/Inf/02/2017 in Jenaer Schriften zur Mathematik und Informatik (2017)
7. van Binsbergen, L.T., Bransen, J., Dijkstra, A.: Linearly ordered attribute grammars: With automatic augmenting dependency selection. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. pp. 49–60. PEPM '15, ACM, New York, NY, USA (2015), https://doi.org/10.1145/2678015.268254310.1145/2678015.2682543, http://doi.acm.org/10.1145/2678015.2682543
8. Boyland, J.T.: Remote attribute grammars. Journal of the ACM **52**(4), 627–687 (Jul 2005), https://doi.org/10.1145/1082036.108204210.1145/1082036.1082042, http://doi.acm.org/10.1145/1082036.1082042
9. Bransen, J., Middelkoop, A., Dijkstra, A., Swierstra, S.D.: The kennedy-warren algorithm revisited: Ordering attribute grammars. In: Russo, C., Zhou, N.F. (eds.) Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings, pp. 183–197. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-27694-1_1410.1007/978-3-642-27694-1_14, http://dx.doi.org/10.1007/978-3-642-27694-1_14
10. Campusano, M., Fabry, J.: Live robot programming: The language, its implementation, and robot {API} independence. Science of Computer Programming **133, Part 1**, 1 – 19 (2017), https://doi.org/http://dx.doi.org/10.1016/j.scico.2016.06.002http://dx.doi.org/10.1016/j.scico.2016.06.002, http://www.sciencedirect.com/science/article/pii/S0167642316300697
11. Ekman, T., Hedin, G.: The jastadd system — modular extensible compiler construction. Science of Computer Programming **69**(1–3), 14 – 26 (2007), https://doi.org/http://dx.doi.org/10.1016/j.scico.2007.02.003http://dx.doi.org/10.1016/j.scico.2007.02.003, http://www.sciencedirect.com/science/article/pii/S0167642307001591, special issue on Experimental Software and Toolkits
12. Ekman, T., Hedin, G.: Rewritable Reference Attributed Grammars. In: Odersky, M. (ed.) ECOOP 2004 – Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3086, pp. 147–171. Springer Berlin Heidelberg (2004), https://doi.org/10.1007/978-3-540-24851-4_710.1007/978-3-540-24851-4_7, http://dx.doi.org/10.1007/978-3-540-24851-4_7
13. Farrow, R., Marlowe, T.J., Yellin, D.M.: Composable attribute grammars: Support for modularity in translator design and implementation. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 223–234. POPL '92, ACM, New York, NY, USA (1992), https://doi.org/10.1145/143165.14321010.1145/143165.143210, http://doi.acm.org/10.1145/143165.143210
14. Ganzinger, H., Giegerich, R.: Attribute coupled grammars. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction. pp. 157–170. SIGPLAN '84, ACM, New York, NY, USA (1984), https://doi.org/10.1145/502874.50289010.1145/502874.502890, http://doi.acm.org/10.1145/502874.502890
15. Hedin, G.: Reference attributed grammars. Informatica (Slovenia) **24**(3), 301–317 (2000)
16. Jazayeri, M., Walter, K.G.: Alternating semantic evaluator. In: Proceedings of the 1975 Annual Conference. pp. 230–234. ACM '75, ACM, New York, NY, USA (1975), https://doi.org/10.1145/800181.81032810.1145/800181.810328, http://doi.acm.org/10.1145/800181.810328
17. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Informatica **31**(7), 601–627 (1994), https://doi.org/10.1007/BF0117754810.1007/BF01177548, http://dx.doi.org/10.1007/BF01177548
18. Kastens, U.: Ordered attributed grammars. Acta Informatica **13**(3), 229–256 (1980), https://doi.org/10.1007/BF0028864410.1007/BF00288644

19. Kastens, U.: Attribute Grammars as a specification method. In: Alblas, H., Melichar, B. (eds.) Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science, vol. 545, pp. 16–47. Springer Berlin Heidelberg (1991), https://doi.org/10.1007/3-540-54572-7_210.1007/3-540-54572-7_2, http://dx.doi.org/10.1007/3-540-54572-7_2
20. Kastens, U., Hutt, B., Zimmermann, E.: GAG, a practical compiler generator, Lecture Notes in Computer Science, vol. 141. Springer-Verlag (1982), https://doi.org/10.1007/BFb003429710.1007/BFb0034297
21. Kastens, U., Waite, W.: Name analysis for modern languages: a general solution. Software: Practice and Experience **47**(11), 1597–1631 (2017), https://doi.org/10.1002/spe.248910.1002/spe.2489, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2489
22. Kastens, U., Waite, W.M.: Reusable specification modules for type analysis. Software: Practice and Experience **39**(9), 833–864 (2009), https://doi.org/10.1002/spe.91710.1002/spe.917, http://dx.doi.org/10.1002/spe.917
23. Kats, L.C.L., Sloane, A.M., Visser, E.: Decorated attribute grammars: Attribute evaluation meets strategic programming. In: de Moor, O., Schwartzbach, M. (eds.) Compiler Construction, Lecture Notes in Computer Science, vol. 5501, pp. 142–157. Springer Berlin Heidelberg (2009), https://doi.org/10.1007/978-3-642-00722-4_1110.1007/978-3-642-00722-4_11, http://dx.doi.org/10.1007/978-3-642-00722-4_11
24. Knuth, D.E.: Semantics of context-free languages. Mathematical systems theory **2**(2), 127–145 (1968), https://doi.org/10.1007/BF0169251110.1007/BF01692511, http://dx.doi.org/10.1007/BF01692511
25. Konat, G., Kats, L., Wachsmuth, G., Visser, E.: Software Language Engineering: 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers, chap. Declarative Name Binding and Scope Rules, pp. 311–331. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), https://doi.org/10.1007/978-3-642-36089-3_1810.1007/978-3-642-36089-3_18, http://dx.doi.org/10.1007/978-3-642-36089-3_18
26. Koskimies, K.: Object-orientation in attribute grammars. In: Alblas, H., Melichar, B. (eds.) Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings, pp. 297–329. Springer Berlin Heidelberg, Berlin, Heidelberg (1991), https://doi.org/10.1007/3-540-54572-7_1110.1007/3-540-54572-7_11, http://dx.doi.org/10.1007/3-540-54572-7_11
27. Lämmel, R., Riedewald, G.: Reconstruction of paradigm shifts. In: Parigot, D., Mernik, M. (eds.) Second Workshop on Attribute Grammars and their Applications WAGA'99. pp. 37–56. INRIA rocquencourt, Amsterdam, The Netherlands (March 1999), satellite event of ETAPS'99 http://www-sop.inria.fr/members/Didier.Parigot/www/fnc2/WAGA99/waga99.html
28. Lewis, P., Rosenkrantz, D., Stearns, R.: Attributed translations. Journal of Computer and System Sciences **9**(3), 279 – 307 (1974), https://doi.org/http://dx.doi.org/10.1016/S0022-0000(74)80045-0http://dx.doi.org/10.1016/S0022-0000(74)80045-0, http://www.sciencedirect.com/science/article/pii/S0022000074800450
29. Paakki, J.: Attribute grammar paradigms – a high-level methodology in language implementation. ACM Comput. Surv. **27**(2), 196–255 (Jun 1995), https://doi.org/10.1145/210376.19740910.1145/210376.197409, http://doi.acm.org/10.1145/210376.197409
30. Simon, E.: A new programming methodology using attribute grammars. Acta Cybernetica **7**(4), 425–436 (Aug 1986), http://dl.acm.org/citation.cfm?id=17075.17082
31. Tienari, M.: On the definition of an attribute grammar. In: Jones, N.D. (ed.) Semantics-Directed Compiler Generation: Proceedings of a Workshop Aarhus, Denmark, January 1980, pp. 408–414. Springer Berlin Heidelberg, Berlin, Heidelberg (1980), https://doi.org/10.1007/3-540-10250-7_3110.1007/3-540-10250-7_31, http://dx.doi.org/10.1007/3-540-10250-7_31
32. Vacchi, E., Cazzola, W.: Neverlang: A framework for feature-oriented language development. Computer Languages, Systems & Structures **43**, 1 – 40 (2015), https://doi.org/https://doi.org/10.1016/j.cl.2015.02.001https://doi.org/10.1016/j.cl.2015.02.001, http://www.sciencedirect.com/science/article/pii/S1477842415000056
33. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Sci. Comput. Program. **75**(1-2), 39–54 (Jan 2010), https://doi.org/10.1016/j.scico.2009.07.00410.1016/j.scico.2009.07.004, http://dx.doi.org/10.1016/j.scico.2009.07.004
34. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation. pp. 131–145. PLDI '89, ACM, New York, NY, USA (1989), https://doi.org/10.1145/73141.748310.1145/73141.74830, http://doi.acm.org/10.1145/73141.74830
35. Waite, W.M., Goos, G.: Compiler construction. Springer-Verlag, New York (1984)

Zustandsbasiertes Debugging mit abstrakten Zustandsmaschinen

Sebastian Beyer und Wolf Zimmermann
{sebastian.beyer|wolf.zimmermann}@informatik.uni-halle.de

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg

Zusammenfassung. In der vorliegenden Arbeit wird ein Ansatz für das Debugging von sequentiellen Programmen mittels abstrakter Zustandsmaschinen (ASMs) beschrieben. Dazu wird einleitend eine kurze Erläuterung der Grundlagen und der Modellierung von Programmiersprachkonzepten durch ASMs gegeben. Danach führen wir zwei weitere Update-Formen in der ASM (`break` und `watch`) ein, um sowohl das Verhalten der klassischen Debugging-Mechanismen (Breakpoint und Watchpoint) nachzubilden als auch das Debugging auf Programmpunkten auf das Debugging von beliebigen Programmzuständen verallgemeinern zu können.

1 Einleitung

Abstrakten Zustandsmaschinen (engl. Abstract State Machines, kurz: ASM) können verwendet werden, um eine operationale Semantik von Programmiersprachen zu definieren [12,2,13,7,9,11,4,1,17]. ASMs definieren ein Zustandsübergangssystem, wobei Zustände Σ-Algebren über einer Signatur Σ sind. Zustandsübergänge sind durch sogenannte Aktualisierungsmengen definiert, die die Änderung der Interpretation von Funktionssymbolen von Σ spezifieren [10]. Die Aktualisierungsmengen werden regelbasiert spezifiziert. Verschiedene Werkzeuge führen ASMs aus, so dass eine interpretative Ausführung von Programmen einer Programmiersprache möglich ist und damit auch klassisches Debugging durch Einzelschrittausführung ermöglicht [6]. In klassischen Debuggern sind vorrangig zwei Mechanismen integriert, um Defekte in Programmen aufspüren zu können. Das ist zum einen der Haltepunkt (engl: *Breakpoint*), mit dessen Hilfe die Ausführung eines Programms durch den Debugger an einem vorher festgelegten Programmpunkt unterbrochen wird. Der Nutzer des Debuggers hat dann die Möglichkeit, den Programmzustand einzusehen und die Ursache des Defekts nachzuvollziehen. Zum anderen werden Beobachtungspunkte (engl: *Watchpoint*) genutzt, um herauszufinden, an welchen Programmpunkten eine beobachtete Variable einen bestimmten Wert hat. Hier kann geprüft werden, wann eine Variable einen möglicherweise falschen Wert annimmt.
In dieser Arbeit werden ASMs um zwei spezielle Formen von Aktualisierungen (hier genannt: *Pseudo-Aktualisierungen*) erweitert: `break` und `watch`. Falls ein

`break` in der Aktualisierungsmenge eines neuen Zustands vorkommt, so wird der Lauf der ASM gestoppt und der gesamte ASM-Zustand *vor* dem Anwenden der Aktualisierungsmenge wird angezeigt. Nach einer Bestätigung des Nutzers wird der Lauf fortgesetzt. Wenn die Pseudo-Aktualisierung `watch` Element der Aktualisierungsmenge ist, so wird der an `watch` übergebene Term ausgewertet, das Ergebnis protokolliert und dann die übrige Aktualisierungsmenge auf den aktuellen Zustand angewandt. Der Lauf der ASM wird hier also nicht unterbrochen.
In Kombination mit ASM-Regeln (hier: `IF`-Regeln) ist es möglich, mit `break` und `watch` das Verhalten von Breakpoints bzw. Watchpoints aus dem klassischen Debugging zu definieren. Ferner kann im Unterschied zu den klassischen, programmpunktbasierten Mechanismen, die die meisten Debugger nutzen, in ASM-Regeln auf den gesamten ASM-Zustand zurückgegriffen werden. Dies erlaubt zusammen mit den `IF`-Regeln ein Debugging auf Basis beliebiger Eigenschaften von Programmzuständen.

2 Grundlagen

Dieser Abschnitt fasst die notwendigen Grundlagen für die Modellierung der Semantik einer Programmiersprache durch abstrakte Zustandsmaschinen wie folgt zusammen: Im ersten Abschnitt wird ein Zustandsbegriff für die ASM formuliert (Abschnitt 2.1). Der zweite Abschnitt führt abstrakte Zustandsmaschinen als Zustandsübergangssystem ein (Abschnitt 2.2). Einige der Definitionen sind an denen aus einer Lehrveranstaltung zu Semantik von Programmiersprachen angelehnt.

2.1 Algebraische Grundlagen

Dieser Abschnitt umfasst die algebraischen Grundlagen, die für die Definition des Zustandsbegriffs der ASM benötigt werden. Die Basis für alles Folgende bildet die Signatur [18]. Um die Modellierung verschiedener Konzepte von Programmiersprachen zu vereinfachen, werden die Begriffe partielle Signatur und ordnungssortierte Signatur aus [18] zusammengeführt.

Definition 1. *Partielle, ordnungssortierte Signatur (POS)*
Eine partielle, ordnungssortierte Signatur Σ ist ein Tupel $\langle S, \sqsubseteq, F, F' \rangle$ wobei

- *S eine endliche Menge von Sorten,*
- *$\sqsubseteq \subseteq S \times S$ eine Halbordnung auf Sorten (*Untersortenbeziehung*),*
- *F eine endliche Menge von* totalen *Operationssymbolen mit der Eigenschaft, dass falls $f : s_1 \times ... \times s_n \to s \in F$, $s'_1 \sqsubseteq s_1, ..., s'_n \sqsubseteq s_n$ und $s' \sqsubseteq s$, dann ist $f : s'_1 \times ... \times s'_n \to s' \in F$*
- *und F' eine endliche Menge von* partiellen *Operationssymbolen mit der Eigenschaft, dass falls $f : s_1 \times ... \times s_n \to ? \ s \in F'$, $s'_1 \sqsubseteq s_1, ..., s'_n \sqsubseteq s_n$ und $s' \sqsubseteq s$, dann ist $f : s'_1 \times ... \times s'_n \to ? \ s' \in F'$*

ist.

Mit partiellen, ordnungssortierten Signaturen lassen sich Schnittstellen für partielle Funktionen und eine Hierarchie auf Sorten modellieren. Zu beachten ist, dass bei einer POS, die ein Operationssymbol f enthält, deren Parametersorten und/oder deren Ergebnissorte eine Obersorte einer anderen Sorte ist, so wird f mit allen möglichen Untersortenkonstellationen überladen.
Das folgende Beispiel zeigt auf, wie Signaturen definiert werden können.

Beispiel 1. Signatur: INT

```
spec INT is
    sorts
        Integer
    end
    ops
        null:                          → Integer;
        succ:       Integer            → Integer;
        intAdd:     Integer, Integer   → Integer;
        intSub:     Integer, Integer   → Integer;
        intMul:     Integer, Integer   → Integer;
        intDiv:     Integer, Integer   →? Integer;
    end
end
```

Das Beispiel enthält die Schnittstelle zur Spezifikation eines ganzzahligen Zahlentyps mit den typischen arithmetischen Operationen. Um die Modellierung von Signaturen besser strukturieren zu können, wird im Folgenden die Signaturerweiterung definiert.

Definition 2. *Signaturerweiterung [18]*
Sei $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$ eine partielle, ordnungssortierte Signatur. Eine POS $\overline{\Sigma} = \langle \overline{S}, \overline{\sqsubseteq}, \overline{F}, \overline{F'} \rangle$ mit $S \subseteq \overline{S}$, $\sqsubseteq \subseteq \overline{\sqsubseteq}$, $F \subseteq \overline{F}$ und $F' \subseteq \overline{F'}$ heißt Erweiterung *von Σ, notiert durch $\overline{\Sigma}$ extends Σ*

Um Ausdrücke über Signaturen zu bilden, werden an dieser Stelle Terme definiert.

Definition 3. *Σ-Term [18]*
Gegeben sei eine POS $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$:
Sei X eine S-sortierte Menge von freien Variablen *(mit $X \cap (F \cup F') = \emptyset$). Für jede Sorte $s \in S$ ist die Menge $T(\Sigma, X)_s$ der Terme der Sorte s die kleinste Menge, die*

1. *jedes $x \in X_s$ und jedes nullstellige Operationssymbol $f : \rightarrow s \in F \cup F'$*
2. *jedes $f(t_1, ..., t_n)$, wobei $f : s_1 \times ..., \times s_n \rightarrow s \in F \cup F'$ und jedes $t_i \in T(\Sigma, X)_{s_i}$ $(i = 1, ..., n)$*
3. *alle Terme $T(\Sigma, X)_{s'}$, mit $s' \sqsubseteq s$*

enthält.

Terme ohne Elemente aus X werden *Grundterme* genannt und $T(\Sigma, \emptyset)_s$ als $T(\Sigma)_s$ notiert. Mit partiellen, ordnungssortierten Signaturen und Termen ist nun das Handwerkszeug vorhanden, um eine Schnittstelle für die Semantik einer Programmiersprache zu beschreiben.
Über Σ-Algebren [18] lassen sich Signaturen implementieren. Σ-Algebren ordnen Sorten Trägermengen und Operationssymbolen mathematische Funktionen zu. Informell *implementiert* eine Σ-Algebra eine Signatur $\Sigma = \langle S, F \rangle$, wenn sie jeder Sorte $s \in S$ eine Trägermenge und jedem Operationssymbol $f \in F$ eine Funktion zuordnet.
Im Folgenden wird die Basis-Definition von Σ-Algebren auf partielle, ordnungssortierte Σ-Algebren spezialisiert, um partiellen, ordnungssortierten Signaturen gerecht zu werden. Auch hier werden partielle Σ-Algebren und ordnungssortierte Σ-Algebren aus [18] zusammengeführt.

Definition 4. *Partielle, ordnungssortierte Σ-Algebra*
Gegeben sei eine partielle, ordnungssortierte Signatur $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$:
Eine partielle, ordnungssortierte Σ-Algebra ist ein Paar $\mathfrak{A} = \langle A, \Phi \rangle$ mit

- *$A = \langle A_s \rangle_{s \in S}$ ist eine Familie von Trägermengen ist, so dass $A_s \subseteq A_{s'}$ falls $s \sqsubseteq s'$ ist.*
- *Für alle Funktionssymbole $f : s_1^{(1)} \times \cdots \times s_n^{(1)} \to s^{(1)}, \ldots,\ f : s_1^{(m)} \times \cdots \times s_n^{(m)} \to s^{(m)} \in F$ ist*

$$f_A : \bigcup_{j=1}^{m} A_{s_1}^{(j)} \times \cdots \times \bigcup_{j=1}^{m} A_{s_n}^{(j)} \to \bigcup_{j=1}^{m} A_s^{(j)}$$

  eine totale Funktion (Interpretation des Operationssymbols f), die folgende Bedingung erfüllt:
  Falls $f : s_1 \times \cdots \times s_n \to s \in F$ und $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$ dann ist auch $f_A(a_1, \ldots, a_n) \in A_s$.
- *Für alle Funktionssymbole $f : s_1^{(1)} \times \cdots \times s_n^{(1)} \to ?s^{(1)}, \ldots,\ f : s_1^{(m)} \times \cdots \times s_n^{(m)} \to ?s^{(m)} \in F'$ ist*

$$f_A : \bigcup_{j=1}^{m} A_{s_1}^{(j)} \times \cdots \times \bigcup_{j=1}^{m} A_{s_n}^{(j)} \to \bigcup_{j=1}^{m} A_s^{(j)}$$

  eine partielle Funktion (Interpretation des Operationssymbols f), die folgende Bedingung erfüllt:
  Falls $f : s_1 \times \cdots \times s_n \to ?s \in F'$ und $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$ dann ist auch $f_A(a_1, \ldots, a_n) \in A_s \cup \{\bot\}$.

Partielle, ordnungssortierte Σ-Algebren ordnen Sorten in Untersortenbeziehung Trägermengen in einer Teilmengenbeziehung zu. Außerdem werden überladene Funktionssymbole f jeweils durch eine Funktion f_A implementiert, die jeweils die *größtmöglichen* Sorten in der Halbordnung der Sorten ($\sqsubseteq$) als Parameter/Ergebnis nutzt. Partielle Funktionen erweitern totale Funktionen, indem sie Definitionslücken im Bildbereich der Funktion durch ein Bottom-Element ($\bot$) auffüllen. Die Interpretation von Termen in der Σ-Algebra wird wie folgt definiert.

Definition 5. *Interpretation von Termen [18]*
Gegeben sei eine POS $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$ *und eine POA* $\mathfrak{A} = \langle A, \Phi \rangle$*, die* Σ *implementiert:*
Sei $\beta : X \to A$ *eine* Belegung *der Variablen* $x \in X$*. Die Interpretation eines Terms* t *in* $\mathfrak{A}$ *ist eine Abbildung* $v : T(\Sigma, X) \to A$ *mit:*

1. $v(x) := \beta(x)$ *für alle* $x \in X$
2. $v(f(t_1, ..., t_n)) := f_A(v(t_1), ..., v(t_n))$ *für alle Operationssymbole* $f \in F \cup F'$ *und*
   $t_1 \in T(\Sigma, X)_{s_1}$, ..., $t_n \in T(\Sigma, X)_{s_n}$.

2.2 Abstrakte Zustandsmaschinen

Dieser Abschnitt führt in die Grundlagen zu abstrakten Zustandsmaschinen ein, die in den nachfolgenden Abschnitten genutzt werden, um die Semantik einer Programmiersprache zu modellieren. Im letzten Teil dieser Arbeit wird das hier definierte ASM-Modell um die zusätzlichen Aktualisierungsformen erweitert, um so das Debugging von Programmen mit dem ASM-Modell zu ermöglichen.
Doch zuvor ist es notwendig eine abstrakte Zustandsmaschine als Transitionssystem herzuleiten. Dafür werden Aktualisierungen benötigt.

Definition 6. *Abstrakte Aktualisierung*
Gegeben sei eine partielle, ordnungssortierte Signatur $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$.
Eine abstrakte Aktualisierung *ist ein Paar von* Σ*-Grundtermen, notiert durch* $t_1 := t_2$.

Mit abstrakten Aktualisierungen werden analog zu den Definitionen des vergangenen Abschnitts erst die Schnittstellen für Aktualisierungen beschrieben. Dies ist notwendig, da die in dieser Arbeit genutze ASM-Beschreibungssprache ausschließlich Aktualisierungen auf Basis von Σ-Termen zulässt.

Definition 7. *(Konkrete) Aktualisierung [17]*
Gegeben sei eine abstrakte Aktualisierung $f(t_1, ..., t_n) := t$ *und die Interpretationsfunktion der* Σ*-Terme* v*. Eine* (konkrete) Aktualisierung *(engl.* Update*) für* Σ*-Algebra* $\mathfrak{A}$ *ist ein Tripel* $(f, (v(t_1), ..., v(t_n)), v(t))$*, wobei* f *ein Operationssymbol und* $v(t_1) \in A_1, ..., v(t_n) \in A_n, v(t) \in A$*, mit* $A_1, ..., A_n, A$ *Trägermengen von* $\mathfrak{A}$*, sind. Eine Menge von Aktualisierungen heißt* Aktualisierungsmenge.

Über Aktualisierungen lässt sich die Interpretation von Operationssymbolen einer partiellen, ordnungssortierten Signatur ändern. Dabei wird eine (konkrete) Aktualisierung stehts über die entsprechende Schnittstelle bzw. abstrakte Aktualisierung ermittelt. In einer Aktualisierungsmenge werden (konkrete) Aktualisierungen zusammengefasst. Hierbei muss eine Aktualisierungsmenge als wichtiges Kriterium *konsistent* sein.

Definition 8. *Konsistente Aktualisierungsmenge [17]*
Eine Aktualisierungsmenge U *ist* konsistent*, wenn sie die folgende Bedingung erfüllt:*
wenn $(f, (a_1, ..., a_n), b) \in U$ *und* $(f, (a_1, ..., a_n), c) \in U$ *ist, dann gilt* $b = c$.

An diesem Punkt ist es möglich, mittels Aktualisierungen die Interpretation von Operationssymbolen zu ändern. Dies wird im Folgenden genutzt, um einen Übergang von einer partiellen, ordnungssortierten Σ-Algebra in eine andere POA zu beschreiben.

Definition 9. *Feuern von Aktualisierungen [17]*
Das Ergebnis des Feuerns einer konsistenten Aktualisierungsmenge U in einer partielle, ordnungssortierte Σ-Algebra $\mathfrak{A} = \langle A, \Phi_A \rangle$ ist eine neue POA $\mathfrak{B} = \langle B, \Phi_B \rangle$, in dem die Interpretation der Funktionssymbole f aus Σ wie folgt geändert wird:

$$f_B(a_1, ..., a_n) = \begin{cases} b & (f, (a_1, ..., a_n), b) \in U \\ f_A(a_1, ..., a_n) & sonst \end{cases}$$

Das Feuern von Aktualisierungen zeigt, wie eine konsistente Aktualisierungsmenge auf eine partielle, ordnungssortierte Σ-Algebra angewandt wird und so eine neue POA entsteht. Auf diesem Weg ist es möglich, wenn POAs als Zustände betrachtet werden, einen Zustandsübergang zu definieren, der für ein Zustandsübergangssystem benötigt wird. Dabei kommt hier die vorher definierte Eigenschaft der *Konsistenz* von Aktualisierungsmengen zum Tragen, da es sonst nicht möglich ist, die Nachfolge-Algebra eindeutig zu bestimmen bzw. könnten dann zwei unterschiedliche Algebren aus der Ausgangs-Algebra berechnet werden.
Um zu verdeutlichen, wie der Übergang zwischen Σ-Algebren funktioniert, wird an dieser Stelle die Modellierung des ganzzahligen Datentyps INT um eine kleine Sortier-Komponente erweitert.

Beispiel 2. Schnittstelle für das Sortieren von Zahlen

```
spec SORTING extends INT is
    sorts end
    ops
                  a:        → Integer;
                  b:        → Integer;
                  c:        → Integer;
        partial   swap:     → Integer;
    end
end
```

Hierbei sind drei konstante Operationssymbole definiert, die nachfolgend drei zu sortierende Zahlen enthalten. Das Funktionssymbol swap fungiert als Tauschregister, soll also Werte beim Austausch zwischenspeichern. Das folgende Beispiel zeigt zwei Varianten, wie ausgehend von einer *Start*-Algebra I durch das Feuern von Aktualisierungen ein aufsteigendes Sortieren von a nach c möglich ist.

Beispiel 3. Sortieren von Zahlen

1. Variante:

$$\left\{\begin{array}{r} \mathsf{a}_I = 3 \\ \mathsf{b}_I = 1 \\ \mathsf{c}_I = 0 \\ \underset{I}{\mathsf{swap}_I} = \bot \end{array}\right\} \underset{U_1}{\rightarrow} \left\{\begin{array}{r} \mathsf{a}_{\mathfrak{N}_1} = 0 \\ \mathsf{b}_{\mathfrak{N}_1} = 1 \\ \mathsf{c}_{\mathfrak{N}_1} = 0 \\ \underset{\mathfrak{N}_1}{\mathsf{swap}_{\mathfrak{N}_1}} = 3 \end{array}\right\} \underset{U_2}{\rightarrow} \left\{\begin{array}{r} \mathsf{a}_{\mathfrak{N}_2} = 3 \\ \mathsf{b}_{\mathfrak{N}_2} = 1 \\ \mathsf{c}_{\mathfrak{N}_2} = 0 \\ \underset{\mathfrak{N}_2}{\mathsf{swap}_{\mathfrak{N}_2}} = \bot \end{array}\right\}$$

$$U_1 := \{(\mathsf{swap}, (), \mathsf{a}_I), (\mathsf{a}, (), \mathsf{c}_I)\},\ U_2 := \{(\mathsf{c}, (), \mathsf{swap}_{\mathfrak{N}_1}), (\mathsf{swap}, (), \bot)\}$$

2. Variante

$$\left\{\begin{array}{r} \mathsf{a}_I = 3 \\ \mathsf{b}_I = 1 \\ \mathsf{c}_I = 0 \\ \underset{I}{\mathsf{swap}_I} = \bot \end{array}\right\} \underset{U_3}{\rightarrow} \left\{\begin{array}{r} \mathsf{a}_{\mathfrak{N}_3} = 3 \\ \mathsf{b}_{\mathfrak{N}_3} = 1 \\ \mathsf{c}_{\mathfrak{N}_3} = 0 \\ \underset{\mathfrak{N}_3}{\mathsf{swap}_{\mathfrak{N}_3}} = \bot \end{array}\right\}$$

$$U_3 := \{(\mathsf{a}, (), \mathsf{c}_I), (\mathsf{c}, (), \mathsf{a}_I)\}$$

Innerhalb jeder geschweiften Klammer wird eine POA mit den entsprechenden Werten für die einzelnen Funktionen beschrieben. Die Pfeile ($\rightarrow$) zeigen den Übergang zwischen zwei POAs an und sind mit der Aktualisierungsmenge annotiert, die zur Ergebnis-Algebra geführt hat.
Die erste Variante des Beispiels nutzt den swap-Speicher, um schrittweise die Interpretation von a und c so zu ändern, dass die eine aufsteigende Sortierung entsteht. Variante zwei macht keinen Gebrauch von swap. Dafür werden hier innerhalb eines Übergangs die Werte für a und c direkt ausgetauscht. Dies zeichnet den Algebra-Übergang aus. Nun kann auf Basis aller bisherigen Definitionen die abstrakte Zustandsmaschine selbst definiert werden.

Definition 10. *Abstrakte Zustandsmaschine*
Sei Σ eine partielle, ordnungssortierte Signatur. Eine abstrakte Zustandsmaschine *(engl.* abstract state machine *[ASM]) ist ein Tupel $M = \langle \Sigma, \mathbb{A}, \mathbb{I}, R \rangle$, wobei*

i. $\mathbb{A}$ *eine Menge von partiellen, ordnungssortierten Σ-Algebren (*Zustände*)*
ii. $\mathbb{I} \subseteq \mathbb{A}$ *(*Anfangszustände*)*
iii. und R eine endliche Menge von Regeln der Form **if** φ **then** $\mathbb{U}$*, φ eine prädikatenlogische Formel (vgl. [3]) über Σ und $\mathbb{U}$ eine endliche Menge von Aktualisierungen*

ist. Die Menge $\text{Update}_M(\mathfrak{A}) := \{u \in U : \exists$ **if** φ **then** $U \in R \mid \mathfrak{A} \models \varphi\}$ *heißt* Aktualisierungsmenge im Zustand $\mathfrak{A}$.

Die Definition ist abgelehnt an [17]. Neben dem Unterschied, dass hier mehrere Startzustände zugelassen werden (vgl. [3]), besteht im Besonderen in der Definition der Menge der Regeln R Unterschiede, da in dieser Arbeit ausschließlich Regeln der Form **if** φ **then** U (`IF`-Regel) verwendet werden. Die Aktualisierungsmenge für den Übergang von einem Zustand $\mathfrak{A}$ in einen Zustand $\mathfrak{B}$ setzt sich

aus den Aktualisierungsmengen der `IF`-Regeln zusammen, deren Bedingung (φ) im Zustand $\mathfrak{A}$ zu `true` auswertet ($\mathfrak{A} \models \varphi$).
Um eine sprachliche Unterscheidung zwischen Operationssymbolen, die ihre Interpretation ändern können und solchen, die es nicht können, zu treffen, werden dynamische Operationssymbole definiert.

Definition 11. *Dynamische Operationssymbole*
Gegeben sei ein partielle, ordnungssortierte Signatur $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$.
Ein Operationssymbol $f \in F \cup F'$ heißt dynamisch*, wenn sich deren Interpretation bei einem Zustandsübergang (Feuern von Aktualisierungen) ändern kann.*

Es ist nun möglich, eine Σ-Algebra ausschließlich durch ihren *dynamischen Anteil*, also ihre dynamischen Funktionen zu beschreiben, die sich nur diese bei einem Übergang zu einer anderen Σ-Algebra ändern können.
Ausgehend von einem der in einer ASM definierten Startzustände, ist ein Lauf eine Folge von Zustandsübergangen. Ein Lauf endet, wenn die im aktuellen Zustand berechnete Aktualisierungsmenge leer oder inkonsistent ist [17].

3 Modellierung einer Programmiersprachsemantik mit Signaturen und ASMs

Mithilfe der oben beschriebenen Konzepte lassen sich Programmiersprachkonzepte und somit die Semantik einer Programmiersprache effizient beschreiben. Diese Programmiersprachsemantik-Modelle werden benötigt, um Programme der betreffenden Sprache ausführen und auch debuggen zu können.
Dafür wird zuerst der einzige Datentyp der Beispielsprache definiert.

Beispiel 4. Ganzzahligen Datentyp: INTEGER

```
spec INTEGER extends INT is
    sorts
        Bool
    end
    ops
        intNE:      Integer, Integer    → Integer;
        intGT:      Integer, Integer    → Integer;
        isTrue:     Integer             → Bool;
        isFalse:    Integer             → Bool;
    end
end
```

Im vorliegenden Beispiel wurde die Schnittstelle für die Implementierung eines ganzzahligen Zahlentyps INT um Funktionssymbole zum Vergleich von ganzen Zahlen erweitert. Diese kann durch eine Σ-Algebra mit der Menge der natürlichen Zahlen ($\mathbb{N}$) als Trägermenge und entsprechenden Operationen über natürliche Zahlen implementiert werden.
Die Sorte Bool kann durch `true` und `false` implementiert werden. Desweiteren

kann der abstrakte Syntaxbaum zu einem Programm der gegebenen Programmiersprache direkt als Termalgebra für die ASM definiert werden.

Beispiel 5. Modellierung der abstrakten Syntax

```
spec AST extends INTEGER, ID, OP is
    sorts
        Node, Prog ⊑ Node, Decls ⊑ Node, Decl ⊑ Node,
        Stats ⊑ Node, Stat ⊑ Node, Expr ⊑ Node, Intconst ⊑ Node
    end
    ops
            progN:        Decls, Stats     → Prog;
            declsN:       Decls, Decl      → Decls;
            declN:        Id               → Decl;
            statsN:       Stats, Stat      → Stats;
            assgnStatN:   Id, Expr         → Stat;
            brnchStatN:   Expr, Stats      → Stat;
            loopStatN:    Expr, Stats      → Stat;
            binExprN:     Expr, Op, Expr   → Expr;
            iconstExprN:  Intconst         → Expr;
            idExprN:      Id               → Expr;
            eval:         Expr             →? Integer;
    end
end
```

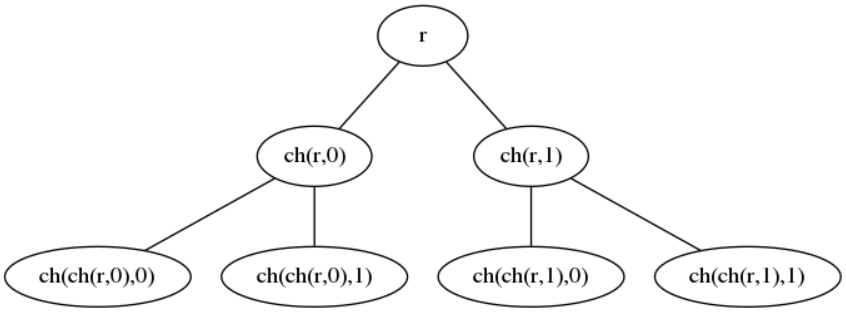
Dieses Beispiel beinhaltet die Modellierung der abtrakten Syntax einer einfachen Programmiersprache. Da bis auf das Erzeugen von abstrakten Syntaxbäumen zur Repräsentation von Programmen keine weitere Semantik für diese Schnittstelle erforderlich ist, kann diese einfach durch eine Termalgebra implementiert werden. Im Beispiel sind alle Baumknoten Untersorte der Sorte `node`.
Um innerhalb von abstrakten Syntaxbäumen eine Adressierung vornehmen zu können, werden Occurences definiert.

Beispiel 6. Adressen im AST (Occurences)

```
spec OCC extends INTEGER is
    sorts
        Occ
    end
    ops
        r:                      → Occ;
        ch:      Occ, Integer → Occ;
    end
end
```

Diese beschreiben eine Schnittstelle für eine Liste von ganzen Zahlen, wobei die Operation zu r die Wurzel des ASTs repräsentiert und alle Kindknoten durch die Operation zu ch, wie im folgenden Beispiel zu sehen ist.

Beispiel 7. Baumadressierung mit Occurences



Das Beispiel zeigt die Adressierung eines Baums. Dabei werden allgemein alle Kindknoten eines Baumknotens von links nach rechts (s. Bsp.) aufsteigend durchnummeriert. So ist eine eindeutige Adressierung im Baum möglich.
Auf Basis der Baumadressierung (Occurences) ist es möglich den Kontrollfluss eines Programms zu spezifizieren. Dies geschieht im folgenden Beispiel.

Beispiel 8. Kontrollfluss

```
spec CONTROLFLOW extends OCC, AST, INTEGER is
    sorts

    end
    ops
        first:                          →? Occ;
        next:       Occ                 →? Occ;
        yes:        Occ                 →? Occ;
        no:         Occ                 →? Occ;
        getNode:    Prog, Occ           →? Node;
        getChild:   Node, Integer       →? Node;
    end
end
```

Die Funktion zu next liefert für eine gegebene Baumadresse die Adresse des nächsten auszuführenden Baumknotens. Falls der aktuelle Baumknoten eine Verzweigung oder eine Schleife beschreibt, so liefern die Operationen zu yes und no die Nachfolgeadresse bei Erfüllung der Verzweigungs-/Schleifenbedingung (yes) bzw. bei Nichterfüllen (no). Die Operation zu first gibt die Adresse des ersten auszuführenden Baumknotens aus. Mit getNode wird eine Zuordnung von Adressen zu Baumknoten definiert. Die Funktion zu getChild liefert für einen gegebenen Baumknoten des Kindknoten analog der Nummerierung in den Baumadressen.
Im Programmzustand werden alle relevanten Schnittstellen für das Sprachsemantik-Modell zusammengefasst.

Beispiel 9. Modellierung des Programmzustands

```
spec PROGSTATE extends INTEGER, ID, AST, CONTROLFLOW is
    sorts end
```

```
    ops
        mem:        Id          → Integer;
        p:                      → Prog;
        pc:                     → Occ;
    end
end
```

Im PROGSTATE werden Schnittstellen für dynamische Operationen, also Funktionen, deren Implementierung sich während eines Zustandsübergangs ändern kann, definiert. Hierei beschreibt mem den Variablenspeicher, p das auszuführende Programm (als AST repräsentiert) und pc den aktuellen Befehlszähler.
Am Ende dieses Kapitels soll noch eine kleine Auswahl an ASM-Regeln aufzeigen, wie die dynamische Semantik der Programmiersprache definiert und so Programme ausgeführt werden können.

Beispiel 10. ASM-Regeln (Auswahl)

```
// (1)
if is(getNode(p, pc), assgnStatN) then
    mem(getChild(getNode(p, pc), 1)) := eval(getChild(getNode(p, pc), 2));
    pc := next(p, pc);
end
// (2)
if is(getNode(p, pc), brnchStatN) ∧ isTrue(eval(getChild(getNode(p, pc), 1))) then
    pc := yes(p, pc);
end
// (3)
if is(getNode(p, pc), brnchStatN) ∧ isFalse(eval(getChild(getNode(p, pc), 1))) then
    pc := no(p, pc);
end
...
```

In der oben gegebenen Auswahl an IF-Regeln bildet die (1) die Behandlung der Auswertung von Ausdrücken. Dabei wird zur Vereinfachung auf die Operation eval zurückgegriffen. next liefert in dieser Regel die Adresse des nächsten zu bearbeitenden Baumknotens. In (2) wird die Verzweigung behandelt, falls die Bedingung der Verzweigung zu true auswertet. Die Implementierung von yes liefert dann die entsprechende Adresse. Analog dazu wird in (3) der Fall abgehandelt, dass die Bedingung der Verzweigung zu false auswertet. Das vordefinierte Prädikat is prüft, ob für einen gegebenen Term in der Termalgebra (hier der abstrakte Syntaxbaum) ein gegebenes Operationssymbol an der Wurzel des Terms (also als äußerstes Symbol) zu finden ist.

4 Debugging-Konzepte von abtrakten Zustandsmaschinen

In diesem Abschnitt wird die vorher definierte ASM um zwei weitere Aktualisierungsformen erweitert (sogenannte *Pseudo-Aktualisierungen*). Die erste Form ist das *break*, dass, falls in der Aktualisierungsmenge enthalten, den Lauf der ASM *vor* dem Feuern der Aktualisierungsmenge im aktuellen Zustand anhält und den Zustand der ASM offenlegt, also den Inhalt aller dynamischen Operationen ausgibt. Falls hingegen die zweite Aktualisierungsform *watch* in der Aktualisierungsmenge enthalten ist, so wird ein an *watch* übergebener Term ausgewertet und das Ergebnis ausgegeben. Der Lauf der ASM wird dabei nicht unterbrochen. Diese beiden Formen können auch gleichzeitig in der Aktualisierungsmenge vorkommen.
Über die `IF`-Regel kann gesteuert werden, wann *break* und/oder *watch* in die Aktualisierungsmenge aufgenommen werden sollen. In diesem Kontext bedeutet *Programm-Debugging* also, dass neue `IF`-Regeln zur ASM hinzugefügt werden, die die Aktualisierungsmenge um *break* und *watch* erweitern können, ohne dabei das Modell der Semantik der Programmiersprache zu verändern.
Klassische Debugging-Methoden basiern zumeist auf Programmpunkten in Form von Programmzeilen. In dieser Arbeit werden die Programmpunkte als Baumadressen (`OCC` - Occurences) betrachtet. Dies ist aber gleichwertig, da für alle Adressen der Blätter des abstrakten Syntaxbaums eine surjektive Abbildung auf die Zeilen im Quellprogramm definiert werden kann.
Das folgende Muster für eine ASM-Regel zeigt verallgemeinert, wie mittels Baumadressen die zu debuggenden Programmpunkte gefunden werden können.

Beispiel 11. Klassisches Debugging mit abstrakten Zustandsmaschinen

if getNode(p, pc) = α **then**
    break;
end

Der Wert α ist dabei die Baumadresse eines AST-Knotens vor dessen Ausführung die ASM anhält. Dieses Muster lässt sich analog für *watch* anwenden.
Am folgenden Programm soll das ASM-Debugging mit Baumadressen verdeutlicht werden:

```
1  DECLS
2      (ch(ch(r,0),1))      VAR x;
3  STATS
4      (ch(ch(ch(r,1),1),0))          x := 0;
5      (ch(ch(ch(ch(r,1),0),1),0))    x := 6 div x;
6  END
```

Das oben gegebene Programm wurde um die relevanten Baumadressen des abstrakten Syntaxbaums werweitert, sodass im Folgenden ersichtlich ist, welcher Programmpunkt für das Debugging genutzt wird. Im Deklarationsteil des Programms wird über die Anweisung `Var x;` eine Variable von einem ganzzahligen Typen definiert. Diese Variable wird im Folgenden mit dem Wert 0 belegt und in

der darauffolgenden Anweisung eine 6 ganzzahlig durch den Wert von `x` dividiert und wieder der Variable zugewiesen. Das dieses Programm zu einem Fehler führt, da hier eine Division durch 0 ausgeführt wird, ist auf Grund der überschaubaren Länge des Programms ziemlich offensichtlich. Trotzdem sollen anhand dieses Beispiels die zwei Aktualisierungsformen der ASMs vorgestellt werden.
Im ersten Schritt soll vor dem Programmpunkt, an dem der Defekt auftritt gehalten werden.

Beispiel 12. break vor Division

```
if getNode(p, pc) = ch(ch(ch(ch(r,1),0),1),0) then
    break;
end
```

Die im Beispiel gegebene Occurence ist die Adresse des Baumknotens für die Zuweisung in Zeile 5. Gemäß dem Muster wird die Ausführung (der Lauf) der abstrakten Zustandsmaschine also vor Ausführung der Zuweisung bzw. vor Auswertung des Ausdrucks auf der rechten Seite der Zuweisung gestoppt. An diesem Punkt ist sind die Inhalte der dynamischen Operationen einsehbar, also auch der Wert der Variable `x`. Die Funktion `getNode` liefert hierzu den Baumknoten des abstrakten Syntaxbaums, der gerade ausgeführt wird. Mithilfe dieser Regel könnte in diesem Fall die unmittelbare Ursache des Fehlverhaltens des Beispielprogramms ermittelt werden, nämlich das die Variable `x` den Wert 0 enthält.
Die Nutzung der `IF`-Regel bietet an dieser Stelle noch viel weitreichendere Möglichkeiten. Diese werden im nächsten Beispiel genauer betrachtet.
Im zweiten Schritt kann mit der Pseudo-Aktualisierung *watch* nachgeprüt werden, wann die Variable `x` den Wert 0 annimmt. Um dies zu erreichen, wird folgende Regel definiert:

Beispiel 13. watch wenn `x` gleich 0

```
if mem("x") == null then
    watch(getNode(p, pc));
end
```

In diesem Beispiel wird wieder über eine `IF`-Regel mittels `mem("x") == null` geprüft, ob der Wert der Variable `x` gleich 0 ist. In diesem Fall liefert das Pseudo-Update `watch` die Baumadresse. Dies liefert nicht nur die AST-Knoten-Adresse zurück, an dem `x` den Wert 0 annimmt, sondern auch alle Adressen der Folgezustände, bei denen die Variable den Wert 0 hat. Dadurch entsteht eine Art Backtrace-Funktion, die beim genauen Nachvollziehen der Fehlersituation hilfreich sein kann.
Es konnte also gezeigt werden, dass mit den Pseudo-Aktualisierungen *break* und *watch* die klassischen Debugging-Methoden nachgebildet werden können. Es ist auch möglich, ein beliebiges Programm vollständig im bereits angesprochenen Einzelschrittverfahren zu durchlaufen. Dafür kann die folgende Regel genutzt werden.

Beispiel 14. Debugging im Einzelschrittverfahren

if *true* **then**
    break;
end

Hier ist *break* in jeder Aktualisierungsmenge enthalten, sodass vor jeder Zustandsänderung der Lauf der ASM angehalten wird. Diese Regel kann unabhängig von einem Programm und auch unabhängig von einem Programmiersprachsemantik-Modell genutzt werden. In Kombination mit den Baumadressen kann das Einzelschrittverfahren natürlich auch auf bestimmte Funktionen oder Programmblöcke beschränkt werden.
Im Folgenden Programm wird unabhängig von Programmpunkten die Schleifeninvariante, der dort vorkommenden `while` Schleife debuggt.

Beispiel 15. Summer der ersten n Zahlen

DECLS
    VAR i; **VAR** x;
STATS
    i := 0; x := 0;
    WHILE i ¡ n **DO**
        i := i + 1;
        x := x + i;
    END
END

Das Beispielprogramm berechnet die Summe über der ersten n Zahlen. Die Invariante der Schleife ist die *gaußsche Summenformel*:

$$\mathrm{Inv}_{\mathrm{while}} \mathrel{\hat{=}} \mathsf{x} = \frac{\mathsf{i} \cdot (\mathsf{i}+1)}{2}$$

Um $\mathrm{Inv}_{\mathrm{while}}$ während der Ausführung des Programms zu überprüfen, kann folgende `IF`-Regel mit *break* definiert werden:

Beispiel 16. Debugging auf Basis der Schleifeninvariante $\mathrm{Inv}_{\mathrm{while}}$

if mem("x") = intDiv(intMul(mem("i"), intAdd(mem("i"), 1), 2) **then**
    break;
end

Das *break* ist in diesem Beispiel im genau dann in der Aktualisierungsmenge, wenn die Schleifeninvariante im Programm erfüllt ist. Der Term in der Bedingung der Regel drückt exakt die vorher bereits genannte Schleifeninvariante $\mathrm{Inv}_{\mathrm{while}}$ aus, wobei `mem` den Variablenspeicher und intDiv, intMul und intAdd die ganzzahlige Division, Multiplikation bzw. Addition abbilden. Auch bei diesem Debugging-Ansatz ist es möglich, die Schleifeninvariante nur in einem bestimmten Programmteil, z. B. bei Eintritt in die Schleife, überprüfen zu lassen. Das programmpunktunabhängige Arbeiten mit *watch* funktioniert analog.
Neben Schleifeninvarianten ist es natürlich auch möglich, Vor- und Nachbedingungen von Funktionen bzw. in objektorientierten Programmiersprachen Klasseninvarianten unabhängig von Programmpunkten zu überwachen.

5 Verwandte Arbeiten

In [16] und [19] werden verschiedene Debugging-Methoden neben den klassischen Break- und Watchpoints beschrieben, die das Finden von Defekten vereinfachen sollen. Dabei ergibt sich, dass das Setzen von programmpunktbasierten Haltepunkten und Beobachtungspunkten oft nur unzureichend für das Finden von Fehlern in Programmen ist. Die ASM Workbench [6] ist Framework, mit dem ASM-Spezifikationen erstellt und auch debuggt werden können. Hierbei kann allerdings nur auf programmpunktbasiertes Debugging aber auch auf Einzelschrittverfahren zurückgegriffen werden. Die CoreASM [8] ist eine durch Plug-Ins erweiterbare ASM-Beschreibung. Über ein solches Plug-In [5] ist das klassische Debugging möglich. Dabei wird allerdings vor allem die ASM-Spezifiaktion selbst debuggt. Andere ASM-Tools, wie z. B. [15] und [14] können die ihnen zu Grunde liegenden Technologien nutzen, um ein (klassisches) Debugging zu ermöglichen. In [14] wird bespielsweise eine ASM-Spezifiaktion nach C++ übersetzt und eröffnet so die Möglichkeit mit den für C++ vorhandenen Debugging-Technologien zu arbeiten.
In keiner der genannten Arbeiten ist aber die Möglichkeit gegeben, Break- und Watchpoints auf Basis beliebiger Programmzustände zu setzen.

6 Schlussfolgerung

In der Arbeit wurde gezeigt, wie unter Verwendung von ASMs das klassische Debugging von Programmpunkten auf allgemeine Eigenschaften von Zuständen erweitert werden kann. Die Erweiterung der Aktualisierungen um die Pseudo-Aktualisierungen `break` und `watch` zeigt, dass damit sowohl das klassische Debugging funktioniert, als auch solches, das über programmpunktbasiertes Debugging hinaus geht, möglich ist.
Zukünftig soll dieser Ansatz auch Anwendung im Debugging von Serviceorientierten Architekturen finden, da die Möglichkeit besteht, mit ASM-Modellen SOAs auf verschiedenen Granularitätsstufen zu betrachten. So kann das Modell auf Protokollebene zur Überwachung und zum Debugging der Kommunikation zwischen Services genutzt werden. Hierbei wird der Service als Black-Box betrachtet. Auch die Modellierung des Services als Komponente in einem Komponentensystem ist möglich und somit die Betrachtung des Gesamtsystems.

Literatur

1. Egon Börger, Nicu G Fruja, Vincenzo Gervasi, and Robert F Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2-3):235–284, 2005.
2. Egon Börger, Uwe Glässer, and Wolfgang Muller. A formal definition of an abstract vhdl'93 simulator by ea-machines. In *Formal Semantics for VHDL*, pages 107–139. Springer, 1995.
3. E. Börger and Robert F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin, Heidelberg, 2003.

4. Steven C Cater and James K Huggins. An asm dynamic semantics for standard ml. In *International Workshop on Abstract State Machines*, pages 203–222. Springer, 2000.
5. Marcel Dausend, Michael Stegmaier, and Alexander Raschke. Debugging abstract state machine specifications: an extension of coreasm. In *Proceedings of iFM*, pages 21–25, 2012.
6. Giuseppe Del Castillo. The asm workbench: a tool environment for computer-aided analysis and validation of abstract state machine models. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 578–581. Springer, 2001.
7. Dirk Fahland and Wolfgang Reisig. Asm-based semantics for bpel: The negative control flow. In *Abstract State Machines*, pages 131–152, 2005.
8. Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. Coreasm: An extensible asm execution engine. *Fundamenta Informaticae*, 77(1-2):71–103, 2007.
9. Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. The formal semantics of sdl-2000: status and perspectives. *Computer Networks*, 42(3):343–358, 2003.
10. Yuri Gurevich et al. Evolving algebras 1993: Lipari guide., 1993.
11. James K Huggins and Wuwei Shen. The static and dynamic semantics of c. In *Local Proc. Int. Workshop on Abstract State Machines*, number 87, 2000.
12. Philipp W Kutter and Alfonso Pierantonio. Montages specifications of realistic programming languages. *Journal of universal computer science*, 3(5):416–442, 1997.
13. Wolfgang Mueller, Juergen Ruf, Dirk Hoffmann, Joachim Gerlach, Thomas Kropf, and Wolfgang Rosenstiehl. The simulation semantics of systemc. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 64–70. IEEE, 2001.
14. Joachim Schmid. Compiling abstract state machines to c++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001.
15. Joachim Schmid. Introduction to asmgofer. *Archived web pages at: http://web. archive. org/web/20070701135506/www. tydo. de/AsmGofer*, 39, 2001.
16. Diomidis Spinellis. Modern debugging: the art of finding a needle in a haystack. *Communications of the ACM*, 61(11):124–134, 2018.
17. Robert F Stärk, Joachim Schmid, and Egon Börger. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media, 2012.
18. Martin Wirsing. Algebraic specification. In *Formal models and semantics*, pages 675–788. Elsevier, 1990.
19. Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

Shape Inference from Memory Graphs (Extended Abstract)*

Jan H. Boockmann[0000−0001−6816−8393] and Gerald Lüttgen

Software Technologies Research Group, University of Bamberg, Germany
{jan.boockmann, gerald.luettgen}@swt-bamberg.de

Abstract. This paper presents ongoing research on an approach to automatically infer recursive shape predicates from memory graphs, so as to formally describe the dynamic data structures contained in a program. These predicates are expressed in separation logic and can be used for program verification and comprehension.

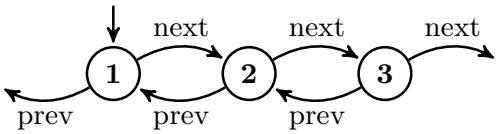
Introduction. The inference system proposed in this paper aims to synthesize a *shape predicate* phrased in *separation logic* [4] that accurately captures the list-based dynamic data structures exhibited by some memory graph in a program. Such a graph consists of a set of nodes, each having the same type and connected with other nodes via labeled pointers, and a single unlabeled pointer to the graph's entry node. A *shape predicate* encodes the shape, i.e., the structure, of a dynamic data structure, e.g., an acyclic doubly-linked list (DLL), thus matching all concrete memory graphs corresponding to that structure. To do so, recursive shape predicates reason about (in)equalities of pointer values and employ separation logic's *separating conjunction*, i.e., conjuncts that must hold on separate heap parts. For a DLL example, Figs. 1a to 1c depict a memory graph with type information and the corresponding shape predicate to be inferred. The memory graph originates from a run of a program constructing such a list, while the shape predicate is phrased using two Prolog predicates `entry` and `p`.

Technically, our approach to infer a shape predicate from a given memory graph first generates candidate predicates of increasing complexity and expressed as sets of rules in Prolog, using a heuristics for pruning the search space. Within this rule set and under separation logic semantics, a meta-interpreter then performs a backtracking search for a deterministic subset of the rules, and hence a shape predicate that accurately captures the graph. Finally, the inferred rules are translated to a shape predicate in separation logic.

Memory graph modeling. Memory graphs are modeled by shape predicates encoded by Prolog rules of the form $\texttt{p(This,}\ddagger^{p-1}\texttt{) :- field}_f\texttt{(This,}\ddagger\texttt{)}^{|F|}\texttt{,}\ \texttt{p(}\dagger^{a}\texttt{)}^{r}\texttt{.}$, which denotes a predicate `p` with arity p, where the rule head contains p parameters, including the current node `This`, and the rule body consists of

* This research is partially supported by the German Research Foundation (DFG) under project *DSI2* (grant no. LU 1748/4-2).

(a) Observed memory graph.



(b) Available type information.

```
1 struct dll {
2   struct dll *prev;
3   struct dll *next; };
```

(c) Inferred rules in Prolog syntax that encode the data structure's shape.

```
1 entry(This)   :- next(This, Next), prev(This, null), p(Next, This).
2 p(This, Prev) :- next(This, Next), prev(This, Prev), p(Next, This).
3 p(This, Prev) :- next(This, null), prev(This, Prev).
```

(d) Translated predicates in VeriFast's separation logic syntax.

```
1  predicate entry(struct dll* This;) =
2    malloc_block_dll(This) &*&
3    This->next |-> ?Next &*& This->prev |-> ?Prev &*&
4    Prev == NULL && This != NULL && This != Next &&
5    Next != NULL ? p(Next, This) : false;
6
7  predicate p(struct dll* This, struct dll* Par1;) =
8    malloc_block_dll(This) &*&
9    This->next |-> ?Next &*& This->prev |-> ?Prev &*&
10   Prev != NULL && Prev != This && Prev == Par1 &&
11   Next != This && Next != Par1 && Next != Prev &*&
12   Next != NULL ? p(Next, This) : Next == NULL ? emp : false;
```

Fig. 1. Example involving a DLL shape.

two clause types: field clauses $\texttt{field}_f$ that dereference the fields of `This`, and r recursive calls with a arguments each. The symbol † denotes atom `null` or an already bound variable; ‡ may additionally denote a fresh variable. (In)equalities of pointer values are encoded implicitly by convention: the value of each variable within a rule is distinct and not equal to `null`.

Rule complexity. While the $\texttt{field}_f$ clauses are given by type information, parameters p, a, r are adjustable to derive rules of varying complexity. Small parameter values are sufficient to describe simple shapes, e.g., a singly-linked lists (SLL). More complex shapes require larger values, e.g., our DLL uses $p \in \{1, 2\}$, $a \in \{2\}$, and $r \in \{0, 1\}$. Note that $p = 1$ ensures the existence of an entry predicate, and $r = 0$ permits 'termination' of the recursive predicate.

Pruning. The potentially large search space for finding a shape predicate matching a memory graph is pruned as follows: rules are removed whose field clauses cannot be satisfied by any node, e.g., `next(This, null), prev(This, null)`. Note that the inference system uses Prolog syntax to encode candidate rules, but employs a resource-like evaluation in the style of separation logic, i.e., the current node `This` is 'consumed' during evaluation. Hence, an analysis using separation logic semantics can identify rules that always cause a failure, e.g., rules containing a recursive call with `null`/`This` as first argument or multiple recursive calls

with the same first argument. Optionally, *singleton rules* are removed that may lead to overly simple predicates, e.g., an SLL predicate for a DLL shape.

Backtracking search. A Prolog *meta-interpreter* that simulates separation logic semantics by failing upon consuming a node twice, conducts a backtracking search to find a candidate rule subset matching the memory graph. If unsuccessful, the approach is restarted with candidate rules of increased complexity. The interpreter ensures that the inferred rule subset is *deterministic*, i.e., no two rules solely differ wrt. their recursive calls. Hence, the search space is reduced significantly, at the cost of excluding non-deterministic predicates. Notably, separation logic dialects such as VeriFast [6] require deterministic predicates.

Translation to separation logic. The inferred rule set (Fig. 1c) is translated to predicates in VeriFast (Fig. 1d), where `malloc_block_dll` consumes a memory chunk of type `dll`, operator `|->` introduces a fresh symbolic variable, and `&*&` denotes separating conjunction. Observe that explicit pointer (in)equalities serve as guards for the associated recursive calls within a nested ternary condition. Keyword `emp` denotes no recursive call, and `false` causes failure.

Conclusions & related work. We sketched an approach to infer shape predicates in separation logic from a single memory graph, which is related to systems such as Aleph [5] in the field of *Inductive Logic Programming (ILP)*, where a theory in the form of a predicate is inferred from positive and negative examples, e.g., the provided memory graph. However, ILP systems typically do not employ a resource-sensitive evaluation and may yield non-deterministic rules as output. *Program comprehension* tools such as DSI [7] employ pre-defined (shape) templates to aggregate memory graphs. *Shape analysis* tools such as Predator [3] use information about data structures to reason about properties, such as memory safety, but is often restricted to pre-defined shapes. Its successor Forester [2] learns and encodes common patterns in memory graphs as tree automata, whereas [1] employs a theorem prover in a refinement loop to obtain shape predicates for particular program points.

References

1. Brockschmidt, M., Chen, Y., Cook, B., Kohli, P., Krishna, S., Tarlow, D., Zhu, H.: Learning to verify the heap. Tech. Rep. MSR-TR-2016-17, Microsoft Research
2. Holík, L., Hruska, M., Lengál, O., Rogalewicz, A., Simácek, J., Vojnar, T.: Forester: From heap shapes to automata predicates. In: TACAS. LNCS, vol. 10206, pp. 365–369 (2017)
3. Holík, L., Kotoun, M., Peringer, P., Soková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: HVC. LNCS, vol. 10028, pp. 202–209 (2016)
4. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE (2002)
5. Srinivasan, A.: The ALEPH manual. Comput. Lab. Oxford Univ. https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html (2001)
6. Vogels, F., Jacobs, B., Piessens, F.: Featherweight VeriFast. LMCS **11**(3) (2015)
7. White, D.H., Rupprecht, T., Lüttgen, G.: DSI: An evidence-based approach to identify dynamic data structures in C programs. In: ISSTA. pp. 259–269. ACM (2016)

Muli: Constraint-Logic Object-Oriented Programming mit freien Objekten

Jan C. Dageförde und Herbert Kuchen

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

Zusammenfassung. Constraint-Logic Object-Oriented Programming (CLOOP) kombiniert die Flexibilität objektorientierter (imperativer) Programmierung mit nützlichen Features aus Constraint- und Logik-Programmierung [3]. Daher ist CLOOP insbesondere für diejenigen Anwendungen interessant, deren Geschäftslogik überwiegend objektorientiert programmiert ist, die aber gelegentlich Lösungen für Suchprobleme erfordern.
In CLOOP werden freie Variablen, d. h. logische Variablen, in symbolischen Ausdrücken verwendet. Obwohl bisher nur freie Variablen mit primitiven Typen verwendbar sind [2], ist es in einer Programmiersprache mit Features der Objektorientierung erstrebenswert, Variablen mit Referenztypen – also „freie Objekte“ – ebenfalls zu unterstützen [1]. Es werden Besonderheiten freier Objekte vorgestellt und diskutiert, wie zur Laufzeit mit ihnen umgegangen werden kann. Darüber hinaus wird die Implementierung eines Prototyps beschrieben, welcher freie Objekte zur Ausführungsumgebung der Münster Logic-Imperative Language (Muli) hinzufügt.

Schlüsselwörter: Constraint-logic Object-oriented Programming · freie Variablen · Referenztypen.

Literatur

1. Dageförde, J.C.: Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming. In: Silva, J. (ed.) WFLP 2018: Functional and Constraint Logic Programming, pp. 131–144. Springer (2019). https://doi.org/10.1007/978-3-030-16202-3_8
2. Dageförde, J.C., Kuchen, H.: An Operational Semantics for Constraint-Logic Imperative Programming. In: Seipel, D., Hanus, M., Abreu, S. (eds.) Declarative Programming and Knowledge Management. pp. 64–80. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00801-7_5
3. Dageförde, J.C., Kuchen, H.: A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli. Journal of Computer Languages **53**, 63–78 (2019). https://doi.org/10.1016/j.cola.2019.05.001

Integer Division by Multiplying with the Double-Width Reciprocal

M. Anton Ertl*

TU Wien

Abstract. Earlier work on integer division by multiplying with the reciprocal has focused on multiplying with a single-width reciprocal, combined with a correction and followed by a shift. The present work explores using a double-width reciprocal to allow getting rid of the correction and shift.

1 Introduction

Integer division is expensive on many processors; e.g. on the Skylake microarchitecture[1] an unsigned division o a 64-bit number by a 64-bit number takes 35 cycles, and signed division 42 cycles. By contrast, on the same architecture a 64-bit-by-64-bit division with a 128-bit result takes 3 cycles, and one multiplication can be started per cycle [Fog19]. So replacing division by constants with multiplication by the reciprocal improves performance.

Therefore a number of compilers perform this optimization. However, in general this approach requires a $w+1$-bit reciprocal and a shift-right for dividing a w-bit number; instead, the compilers use a w-bit reciprocal and some divisor-dependent fixup code. In contrast, this paper explores using a $2w$-bit reciprocal without shift, and, for unsigned division, without fixup. This approach is actually fastest in some circumstances. It is also amenable to optimizing loop-invariant divisors, because the code for computing the parameters of the code in the loop body is relatively simple, especially for unsigned division. By contrast, neither gcc-8 nor clang-6.0 optimize loop-invariant divisors.

Still, in many cases it is probably better to implement other published work (see Section 5), and the main contribution of this paper is to show that using a $2w$-bit reciprocal can be beneficial in some cases, and to give an idea of where it works well.

Section 2 provides background on division by multiplication with the reciprocal. Section 3 discusses using a $2w$-bit reciprocal in unsigned division, while Section 4 discusses signed division (both using unsigned multiplication and signed multiplication).

* anton@mips.complang.tuwien.ac.at

[1] The microarchitecture of the mainstream client, server and mobile CPUs sold by Intel since late 2015, e.g., Core ix-6xxx...9xxx.

1.1 Symbols

In formulas in the rest of the paper we use the same letters for the same concepts:

n	dividend (numerator)
d	divisor (denominator)
q	quotient
r	remainder
w	word width in bits
C	scaled approximate reciprocal
k	2^k is the scale factor for C
C_l	least significant word of C
C_h	most significant word of C
D	correction term (possibly scaled)

2 Background

We first look at unsigned division. The basic idea of integer division by multiplication with the reciprocal is to replace integer division with a cheaper computation:

$$q = \lfloor \frac{n}{d} \rfloor = \lfloor \frac{nC}{2^k} \rfloor$$

$\lfloor x/2^k \rfloor$ can be implemented with shifts, or, in our approach, by just selecting the right word of a multi-word result. Multiplication is also much cheaper than division on many CPUs. The question is how to determine C and k.

A simple way to select C is to compute

$$C = \lceil \frac{2^k}{d} \rceil = \frac{2^k + e}{d}$$

where e ($0 \leq e < d$) indicates the size of the error we get from rounding up. So if we put that into our formula above, we get:

$$\lfloor \frac{n}{d} \rfloor = \lfloor n \frac{2^k + e}{d2^k} \rfloor = \lfloor \frac{n}{d} + \frac{ne}{d2^k} \rfloor$$

By increasing k, we can reduce the error, but this also increases C, requiring longer multiplication. Earlier work tried to go for the smallest C and the smallest k that produces the correct result. It turns out that, in general, for dealing with w-bit numbers, a $w+1$-bit C is needed. In about 70% of the cases [Fis11], a w-bit C is good enough (e.g., see Fig. 1, first column). For the other 30%, earlier work has devised various ways to make do with a w-bit by w-bit multiplication, by applying some form of correction or an alternative formula. E.g., one of these cases is $n/7$. Fig. 1 shows the code produced by gcc (second column) and by Robison [Rob05] and Fish [Fis11] (third column). The latter variant corresponds to the following formula (which works in those cases where the formula above for C would require more than w bits for C):

gcc $n/10$ 6 cycles latency	gcc $n/7$ 8 cycles latency	Fish $n/7$ 6.25 cycles latency	this paper $n/7$ 6 cycles latency

`movabs $C,%rdx`
`mov %rdi,%rax`
`mul %rdx`
`mov %rdx,%rax`
`shr $0x3,%rax`
	`movabs $C,%rdx`
	`mov %rdi,%rax`
	`mul %rdx`
	`sub %rdx,%rdi`
	`shr %rdi`
	`lea (%rdx,%rdi),%rax`
	`shr $0x2,%rax`
		`movabs $C,%rax`
		`mov %rax,%rcx`
		`mul %rdi`
		`add %rcx,%rax`
		`adc $0x0,%rdx`
		`shr $0x2,%rdx`
		`mov %rdx,%rax`
			`movabs $Cl,%rax`
			`mul %rdi`
			`mov %rdx,%rcx`
			`movabs $Ch,%rax`
			`mul %rdi`
			`add %rcx,%rax`
			`adc $0x0,%rdx`
			`mov %rdx,%rax`

Fig. 1. One way to divide by 10 and three ways to divide by 7; n is in `%rdi`, result in `%rax`; latency numbers refer to latency from `%rdi` becoming ready until `%rax` is ready; they are measured on a Skylake (Core i5-6600K).

$$\lfloor \frac{n}{d} \rfloor = \lfloor \lfloor \frac{2^k}{d} \rfloor \frac{n+1}{2^k} \rfloor$$

3 Using wider multiplication for unsigned division

The reason for trying to use a multiplier $C < 2^w$ is the presumption that multiplication with bigger C is expensive. While w-bit by $2w$-bit multiplication is more expensive than w-bit by w-bit multiplication, it is not that much more expensive, and using it can actually be cheaper than the workarounds for making do with w-bit by w-bit multiplication. E.g., the AMD64 code for unsigned 64-bit division by 7 is shown in the right column of Fig. 1; on Skylake[2] it has the same latency as the $n/10$ code produced by gcc, but it also works for the $n/7$ case.

This code works with $C = \lceil 2^{2w}/d \rceil$. As a result, we need w-bit by $2w$-bit multiplication, but we do not need a shift, because the result is in the most significant word of the multiplication result. We split C into two words $C = C_l + 2^w C_h$, so the computation is

$$q = \lfloor \frac{nC_l + 2^w nC_h}{2^{2w}} \rfloor = \lfloor \frac{nC_l}{2^{2w}} + \frac{nC_h}{2^w} \rfloor$$

The least significant word of the first multiplication is eliminated by the flooring, and can be ignored; we add the most significant word of the first multiplication to the result of the second multiplication; the most significant word of this sum is our quotient.

The nice thing about this computation is that the two multiplications can theoretically be performed in parallel. In practice, current CPUs have only one multiplier, but this multiplier is pipelined, so the second multiplication can be

[2] Skylake is the microarchitecture of mainstream Intel CPUs since the 6th generation of Core i processors in 2015 and is also used in Intel CPUs of the 7th, 8th, and 9th generation, and some CPUs of the upcoming 10th generation.

started one cycle after the first. By using $k = 2w$, we eliminate the final shift-right used by the other variants, and its impact on latency (2 cycles on Skylake).

3.1 Computing C

Computing $C = \lceil 2^{2w}/d \rceil$ requires division of a $2w + 1$-bit number by a w-bit number, giving a $2w$-bit result. For just-in-time compilation and for dealing with loop-invariant divisors, we do not want to call a (slow) general multi-precision library. Fortunately, if we have a $2w/w \to w$ division (as present in AMD64), this computation can be performed relatively cheaply:

$$C = \lceil \frac{2^{2w}}{d} \rceil = \lfloor \frac{2^{2w} + d - 1}{d} \rfloor$$

$$C_h = \lfloor \frac{2^w}{d} \rfloor, r_h = 2^w \bmod d$$

$$C_l = \lfloor \frac{2^w r_h + d - 1}{d} \rfloor$$

In AMD64 assembly language:

```
#in:  d = %rdi
mov    $1,%rdx
mov    $0,%rax
div    %rdi
mov    %rax,%rsi
lea    -1(%rdi),%rax
div    %rdi
mov    %rsi,%rdx
#out: ch=%rdx cl=%rax
```

3.2 Special cases

Our approach works for $d > 1$.

For $d = 1$, $C = 2^w$, which does not fit in the two words that we reserve for reciprocals. If d is known at compile-time, treating this as a special case is easy. If we use this approach for loop-invariant divisors, we can fork between $d = 1$ and $d > 1$ before the loop, and have a copy of the loop optimized for $d = 1$ (or maybe for powers of 2).

The other special case is $d = 0$. If d is known at compile-time, the compiler can just produce the same code as in the unoptimized case, resulting in the same behaviour. For loop-invariant divisors, we want to minimize the number of special cases. What the compiler should do depends on the programming language, and, for some programming languages, on the compiler.

If the programming language specifies a specific behaviour for division-by-zero, we have to implement that. If the programming language does not define

the behaviour, some compiler writers think that they can do anything, and if you want to follow that path, you can simply ignore that case.

But I argue [Ert17] that even for these cases, the optimized code should exhibit the same behaviour as the unoptimized case. One way to deal with this is to define a specific behaviour (e.g., an exception) at the compiler level; then you have to implement that. Another way is to just use the hardware divide instruction in the unoptimized case and also for the "optimized" case; then a way to reduce special cases is to combine this case with $d = 1$ and use a loop that uses the divide instruction (i.e., without optimizing the division) for $d < 2$.

Most architectures define the behaviour on divide-by-zero, and in these cases we may be able to do better:

E.g., the AMD64 architecture specifies that division-by-zero produces an exception. For the loop-invariant case, this exception will occur when computing C, and we do not need to worry about $d = 0$ in the rest of the loop. However, this means that for $d \neq 1$ we have to peel the first iteration of the loop, and compute C at the place of that first-iteration division (in order to preserve the order of exceptions).

Aarch64 specifies that division-by-zero produces 0. Then we can just use $C = 0$, and using that in $\lfloor nC/2^{2w} \rfloor$ gives 0, just like the original div instruction. However, at least Cortex-A53, -A72 and -A73 have very fast dividers, so a compiler probably should forego replacing division by multiplication with the reciprocal for these CPUs. Still, if there are CPUs with slow division instructions where division-by-zero produces 0, you can apply these considerations.

3.3 Possible usage

As a compiler writer, the simplest way to make use of this paper's approach is to use it for all $d > 1$. If you want to invest more development resources into this topic, you can detect the 70% of divisors where a $C < 2^w$ is sufficient, and use a multiply followed by a shift-right for that. You can also optimize dividing by powers-of-two into shift-right.

Finally, if you have enough development resources, you can also detect and optimize loop-invariant divisors. For this use, every special case needs a separate copy of the loop. The benefit of this paper's approach is that it reduces the number of cases that need separate loops; in particular, you can handle $d > 1$ with one loop.

One usage case for loop-invariant divisors is the conversion of integers to strings with arbitrary base (e.g., Java's `Integer.toString(int i, int radix)`). This usage has relatively low trip counts, but also few different radixes (and often the radix is the same as in the last invocation). Given the relatively high cost to compute C, the low trip counts would be a problem. This can be mitigated by memoizing the computation of C.

3.4 Remainder

The remainder of the division can be computed from the quotient in the obvious way.

$$r = n \bmod d = n - qd$$

This is used with every way to compute q (except that on some architectures (e.g., AMD64), the division instruction produces both q and r), including the various ways to performed signed division, so there is no need to discuss it in the rest of this paper.

4 Signed division

4.1 2s-complement numbers

In this section we work with the 2s-complement representation of negative integers. This representation is used in all significant architectures introduced since 1970. In 2s-complement representation, a negative number x is represented by $x' = x + 2^w$. As a result signed multiplication of $a < 0$ with b becomes

$$ab = (a' - 2^w)b = a'b - 2^w b$$

and likewise for $b < 0$. Widening signed multiplication[3] instructions perform these corrections internally. If we use unsigned multiplication instructions, we have to generate code to perform them.

4.2 Signed integer division

In symmetric (aka truncated) division n/d, the quotient is rounded towards 0 (truncated), and consequently a non-zero remainder is negative iff the signs of n and d differ.

In floored division, the quotient is rounded towards $-\infty$ (floored), and consequently, a non-zero remainder has the same sign as d.

There is also Euclidean division, where $0 \le r < |d|$ [Bou92].

Different programming languages specify different forms of division and, in particular, remainder operations; some of them support several (using different operators, e.g., `mod` and `rem` in Ada).[4]

[3] $2^w b \equiv 0 \pmod{2^w}$, so in non-widening multiplication the difference between signed and unsigned multiplication vanishes.

[4] https://en.wikipedia.org/wiki/Modulo_operation

4.3 Signed division by unsigned reciprocal multiplication

We first look at using unsigned multiplication.

For $d > 1$, we can use the same C as in the unsigned-division case. However, if $n < 0$, we have to compute

$$nC = (n' - 2^w)C = n'C - 2^w C$$

But there is more: C is a little bigger than the ideal multiplier $2^{2w}/d$. For $n < 0$, this means that the result is a little smaller than n/d, and rounding it towards $-\infty$ produces a result that is one less than the result of symmetric division.

So, for symmetric division, if $n < 0$, the total correction term is

$$D_s = 2^{2w} - 2^w C$$

This is used in

$$q = \lfloor \frac{nC_l + 2^w nC_h + D}{2^{2w}} \rfloor \text{ if } n < 0$$

For floored and Euclidean division (they are the same for $d > 0$), rounding towards $-\infty$ is the right thing in principle, but we have to correct for that fact that C is a little too big, which, combined with rounding towards $-\infty$, produces a division result that is 1 too low when n is divisible by d. One way to correct for that is to add no more than $1/d$ to the end result before flooring. In terms of the scaled correction term this turns out to be $C - 1$ (C itself is a bit too large in general). So we could use the following total correction term:

$$D_{f'} = C - 1 - 2^w C$$

This would require a $3w$-bit addition. In practice we can make do with a $2w$-bit addition by using

$$D_f = 2^w(\lfloor \frac{C-1}{2^w} \rfloor - C)$$

If you also want to optimize negative divisors (they are very rare), one way to deal with them for symmetric and floored division is to perform division by $-d$, with appropriate corrections elsewhere:

For symmetric division, you negate either the quotient q (cheaper), or the dividend n.

For floored division, you negate the dividend n.

For Euclidean division, you negate the quotient q.

Special cases: In addition to $d = 0$ and $d = 1$, signed division also has the special case $d = -1$; this special case has an additional problem compared to $d = 1$: The result of $-2^{w-1}/-1 = 2^{w-1}$ is not representable as a w-bit 2s-complement number. The simplest way to deal with $d = -1$ is not to optimize it.

4.4 Using signed multiplication

It seems obvious to use signed multiplication for signed division, and approaches that use single-width reciprocals often do so. However, even these approaches perform corrections, either for $n < 0$ or independent of the sign, so at least for $d > 0$ these approaches do not offer an advantage over using unsigned multiplication (and $d < 0$ is very rare). Still, this section reports on my explorations in this direction.

Let's consider $d > 2$ first. For $n \geq 0$, this coincides with unsigned division. For $n < 0$:

$$\lfloor \frac{nC}{2^{2w}} \rfloor = \lfloor \frac{n'C - 2^w C}{2^{2w}} \rfloor = \lfloor \frac{n'C_l - 2^w C_l}{2^{2w}} + \frac{n'C_h - 2^w C_h}{2^w} \rfloor$$

So signed multiplication can be used for multiplying both parts of C. There is one complication, though: The result of the low-order multiplication can be negative and needs to be sign-extended to the full width before performing the addition.

For symmetric division, we still need to perform the correction depending on the sign of n.

For floored and Euclidean division, one can perform a correction that is constant across the whole range of n, resulting in the computation

$$q = \lfloor \frac{n'C_l - 2^w C_l}{2^{2w}} + \frac{n'C_h - 2^w C_h}{2^w} + \frac{D_g}{2^{2w}} \rfloor$$

and we use

$$D_{g'} = 2^{w-1}(\frac{C}{2^{2w}} - \frac{1}{d})$$

$$D_g = \lceil 2^{2w} D_g \rceil$$

D'_g is the maximum amount by which our approximation $C/2^{2n}$ deviates from the exact $1/d$ in the negative range of n. D_g is the next integer scaled by 2^{2w} (as used in the formula for q).

The advantage of this approach is that there is no need for a branch or other kind of conditional code. However, in the frequent case that $n \geq 0$ for almost all invocations, it is probably better to use unsigned multiplication with conditional correction for $n < 0$.

For $d < -1$, signed multiplication works nicely (with $C = -\lceil 2^{2w}/-d \rceil = \lfloor 2^{2w}/d \rfloor$), and avoids the need for a separate negation step and is therefore probably better than the variant using unsigned multiplication above. The correction term has to be adjusted appropriately; in particular, the correction factor for Euclidean division is now different than for floored division.

Using signed multiplication turns $d = 2$ into a special case, because $C = 2^{2w-1}$ is not representable as signed $2w$-bit number. It can be handled by treating all $d = 2^i$ as separate case that uses arithmetic shift right; this would also cover $d = 1$.

5 Related work

If you read only one paper about the topic, my recommendation is Robison's [Rob05]. The main benefit of that work is that division is replaced by the computation

$$\lfloor \frac{n}{d} \rfloor = \lfloor \frac{an+b}{2^k} \rfloor$$

where a and b are w-bit numbers and not much harder to compute than C in the present paper (and $b = 0$ or $b = a$). So this approach can be used for loop-invariant divisors. Depending on the circumstances, Robison's computation can be faster or slower than the present paper's computation.

Fish [Fis11] arrives more or less at Robison's computation through different reasoning and suggests an alternative way of coding it.

Early work on division by constants by Artzy et al. [AHS76] did not frame it as using an approximation to the reciprocal.

Alverson [Alv91] presents unsigned and signed division by multiplying with a $w + 1$-bit reciprocal and shifting. Alverson performs signed division on absolute values using unsigned multiplication, with a correction term (bias) for floored division, and correcting the sign in the end.

Granlund and Montgomery [GM94] use signed multiplication (with correction) for the symmetric signed division case, and but use unsigned multiplication and sign manipulation for the signed division case. They also discuss using floating-point multiplication, division of a double-word dividend, the case where it is known that the remainder is 0, implementation in gcc, and results.

Möller and Grandlund [MG11] perform double-word by single-word division using single-word multiplication plus corrections; in a way, the opposite of the present paper, which uses a wider multiplier to eliminate corrections, and only produces a single-word result.

Muller et al. [MTdDM05] discuss implementing 32-bit by 32-bit division using 16-bit by 16-bit multiplication, also in the opposite direction of the present work.

Other works on division by reciprocal multiplication are by Warren [War03, Chapter 10], Cavignino and Werbrouck [CW08,CW11], and Drane et al. [DCC12].

6 Conclusion

In division by multiplying with the reciprocal, using a double-wide reciprocal eliminates the final shift, which can reduce the latency of the whole operation. The benefit is most pronounced for unsigned division, while for signed division conditional or unconditional corrections are needed, which may make using a double-wide reciprocal less attractive.

References

AHS76. E. Artzy, J. A. Hinds, and H. J. Saal. A fast division technique for constant divisors. *Communications of the ACM*, 19(2):98–101, February 1976. 5

Alv91. Robert Alverson. Integer division using reciprocals. In *Proceedings 10th IEEE Symposium on Computer Arithmetic*, pages 186–190, Grenoble, France, June 1991. 5

Bou92. Raymond T. Boute. The Euclidian definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, April 1992. 4.2

CW08. D. Cavagnino and A. E. Werbrouck. Efficient algorithms for integer division by constants using multiplication. *The Computer Journal*, 51(4):470–480, July 2008. 5

CW11. D. Cavagnino and A. E. Werbrouck. An analysis of associated dividends in the DBM algorithm for division by constants using multiplication. *The Computer Journal*, 54(1):148–156, January 2011. 5

DCC12. Theo Drane, Wai-Chuen Cheung, and George A. Constantinides. Correctly rounded constant integer division via multiply-add. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1243–1246. IEEE, 2012. 5

Ert17. M. Anton Ertl. The intended meaning of *Undefined Behaviour* in C programs. In Wolfram Amme and Thomas Heinze, editors, *19. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'17)*, pages 20–28, 2017. 3.2

Fis11. Fish. Labor of division (Episode III): Faster unsigned division by constants. Blog Entry `https://ridiculousfish.com/blog/posts/labor-of-division-episode-iii.html`, October 2011. 2, 5

Fog19. Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. `https://www.agner.org/optimize/instruction_tables.pdf`, 2019. 1

GM94. Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72, Orlando, Fla., June 1994. 5

MG11. Niels Möller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, February 2011. 5

MTdDM05. Jean-Michel Muller, Arnaud Tisserand, Benoit de Dinechin, and Christophe Monat. Division by constant for the ST100 DSP microprocessor. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, 2005. 5

Rob05. Arch D. Robison. N-bit unsigned division via N-bit multiply-add. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, 2005. 2, 5

War03. Henry S. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2003. 5

Proving Termination of Pointer Programs on Top of Graph-based Symbolic Execution

Ira Justus Fesefeldt

Chair for Software Modeling and Verification
RWTH Aachen, Ahornstraße 55, 52074 Aachen, Germany
justus.fesefeldt@rwth-aachen.de
https://moves.rwth-aachen.de

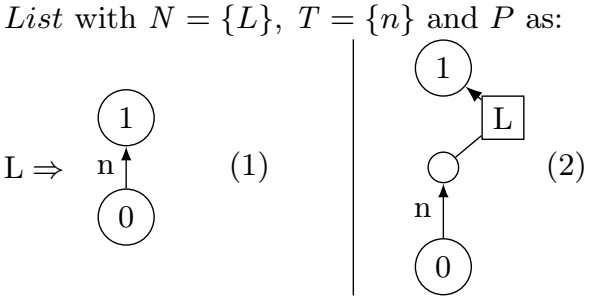
1 Grammars and Transitions

Many programming languages, such as C or Java, use pointers to express data structures. Shape analysis attempts to discover data structure invariants which support the verification of correctness properties, e.g. memory safety. We use a particular shape analysis in which dynamically allocated memory is modeled as languages of graphs which are represented by an abstract graph and a graph grammar[4,3]. Such languages are only capable of describing the shape of data structures. To prove termination of a program, we need to use variants, which we synthesize by further annotating the edges of the graph by an index with a feasible meaning. For demonstrate this for a grammar that generates list data structures. In this case the annotated indices represent the size of the list.

The List Grammar We use edge-replacement grammars[2] in order to abstract graphs. Figure 1a shows an edge-replacement grammar for list data structures. We have a terminal label n and a nonterminal label L. An edge with label L represents a list of arbitrary length. It can be derived by the rule (1) to a graph that only consists of one edge with label n and that represents a list from vertex 0 to vertex 1 of size 1. We use the rule (2) to derive a list from vertex 0 to vertex 1 that has at least 2 selectors: the edge with label n and at least one selector in the nonterminal edge with label L. In a derivation, we replace an edge $e = (v, v')$ with a graph H, where the vertex in H with label 0 is merged with v and the vertex in H with label 1 is merged with v'. We write $H \Rightarrow_{(1)} H'$ for derivation of the rule (1) and similiar for (2). For the language of all derivable graphs starting from H we write $L_{List}(H)$.

Transition Systems The semantics of programs is often described as a transition system[1]. We use a transition system where the states are tuples of a location and a graph, to describe the meaning of a program. The graph in the states represents the allocation in memory in this particular state. However, we will not mention locations in the following example for simplicity. The program P in Figure 1b iterates over all next selectors c.n until it reaches null and removes every object t along the way. The transition system TS_c in Figure 1c is a concrete transition system of the program in Figure 1b. Every transition represents one

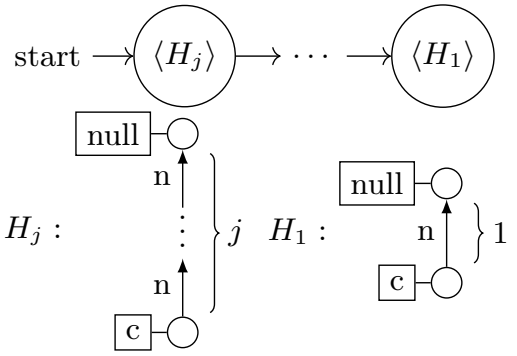
iteration and thus after every transition the pointer c is moved to the next object and the last vertex is removed. To analyse the arbitrarily large TS_c, we abstract



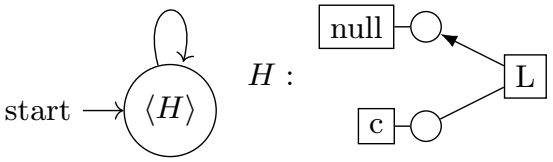
(a) Grammar for Lists

```
while(c != null){
        t := c;
        c := c.n;
        remove(t);
}
```

(b) The Program P to analyse



(c) The concrete transition system for P: $TS_c = (C, \to_c)$



(d) The abstract transition system for TS_c: $TS_a = (A, \to_a)$

Fig. 1: The Setting of the Example

it using a graph-based symbolic execution[4] to the non-arbitrarily, finitly large transition system TS_a in Figure 1d. TS_a only consists of the one state $\langle H \rangle$ that abstracts every concrete state $\langle H_i \rangle$. In particular, the graph H abstracts every graph H_i via the edge with label L.

2 Termination Analysis

Although the transition system TS_a from Figure 1d describes every possible TS_c as in Figure 1c, we lose the size of the list. The size of the list is a feasible variant to prove termination of P. We regain this information by annotating the edges of every graph with indices composed of integers and variables that represent the size. To prove the termination of a concrete transition system, we need to find a variant that is decreasing after every iteration. As already discussed, we suspect the size of the list to be such a variant. Our abstract transition system TS_a only consists of the state $\langle H \rangle$. The goal is now to prove that every transition to a concrete state that is abstracted by $\langle H \rangle$, the size of the list is decreased.

The transition $\langle H \rangle \to_a \langle H \rangle$ is decomposable into three operations, which we can use to calculate the indices after the transition using the index before the

transition. The three operations are: forward derivations (written as $\Rightarrow$), graph-manipulations of the edges with terminal labels (written as $\xrightarrow{\mu}$) and backward derivations (written as $\Leftarrow$). The decomposition of this particular transition is of the form $H \Rightarrow_{(2)} H_m \xrightarrow{\mu} H'_m = H$. Here we do not use any backwards derivations. This decomposition expresses every concrete transition $\langle H_i \rangle \rightarrow_c \langle H_{i-1} \rangle$ in the sense that $H_i \in L_{List}(H_m)$ and $H_{i-1} \in L_{List}(H'_m)$.

We assume that the edge in H with label L is L_1 and we annotate this edge with a variable v. We now assume that this list has size v (where this value is arbitrary, but fixed). After a forward derivation $H \Rightarrow_{(2)} H_m$ the graph H_m consists of an edge n_2 with the label n and an edge L_2 with the label L. The edge n_2 induces a list of size 1. Thus the list induced by L_1 is of size $v-1$. Next, the edge n_2 is removed by μ. The remaining list is thus of size $v-1$. The variant is decreasing after every transition (i.e. it infers a ranking function). In this case, we can assign v the value i for the graph H_i. The index after reaching $\langle H_{i-1} \rangle$ is changed from v to $v-1$ according to our analysis. By inductive application of this reasoning, we can deduce that any infinite path will eventually yield a list of size smaller than one. However, a list can not have a size smaller than one, hence no path in the concrete transition system $\langle H_j \rangle \rightarrow_c \ldots$ can be infinite.

3 Conclusion

The presented technique uses a hyperedge-replacement-based symbolic execution in order to analyse invariants of pointer programs. Hyperedge-replacment grammars (HRG) are a generalised version of edge-replacement grammars[2]. To prove termination of pointer programs, we do however need variants which we synthesize by annotating the hyperedges in the abstract transition system. The annotations gets their meaning by a feasible predicate we have to define (here: the size of the list). For a state c_q we analyse the change of the index after one iteration relative to a fixed but arbitrary value. If the change yields a decreasing index, we can conclude that there exists no path in the concrete transition with infinitely many states that are abstracted by c_q. We can generalize this technique by using predicates that are compatible with the techniques shown here, especially that have a relative index set and rules to handle forward and backward derivations, and analyse the indices by using a data flow analysis[5].

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
2. Habel, A.: Hyperedge Replacement: Grammars and Languages, LNCS, vol. 643. Springer (1992)
3. Heinen, J., Jansen, C., Katoen, J.P., Noll, T.: Juggrnaut: using graph grammars for abstracting unbounded heap structures. FMSD **47**(2), 159–203 (2015)
4. Jansen, C.: Static Analysis of Pointer Programs - Linking Graph Grammars and Separation Logic. Ph.D. thesis, RWTH Aachen University, Germany (2017)
5. Nielson, F., Riis Nielson, H., Hankin, C.: Principles of Program Analysis. Springer (1999)

Studiengang Informatik – die Anfänge

Gerhard Goos
Fakultät für Informatik
Karlsruher Institut für Technologie

© Gerhard Goos 2019

1 Die Vorgeschichte

Die Geschichte der Informatik im deutschsprachigen Raum des zwanzigsten Jahrhunderts beginnt natürlich mit den Unentscheidbarkeitssätzen von Kurt Gödel in Wien, (Gödel, 1931), und dem Wirken von Konrad Zuse, 1910-1995, ab 1936, dem Bau seines ersten, noch mechanischen Rechners Z1 mit binärer Gleitpunktarithmetik 1937, dem Entwurf der ersten *höheren* Programmiersprache, des Plankalküls 1944/45 und den weiteren Rechnern der Zuse KG. Dann folgt die Arbeit von Heinz Billing, 1914-2017, der 1948 den Trommelspeicher erfand und Anfang der 50er Jahre die Rechner G1, G2, G1a, G3 für die Astronomen der Max-Planck-Gesellschaft baute. Die Erfindung des Prinzips des virtuellen Speichers wird gemeinhin englischen Ingenieuren zugeschrieben, die es in Manchester im Atlas Rechner realisierten, (Kilburn et al., 1962). Unabhängig und früher veröffentlichte jedoch Fritz-Rudolf Güntsch die Idee in seiner Dissertation (Güntsch, 1957) an der TU Berlin; in Berlin wurde das aber nie in einen Rechner eingebaut.

Auch die weitere Entwicklung an den Universitäten und dann in der Industrie war vor allem auf die Rechnerkonstruktion gerichtet: OPREMA (Wilhelm Kämmerer, Jena, Carl Zeiss Werke, 1954), Mailüfterl (Heinz Zemanek, Wien, 1958), PERM (Robert Sauer und Hans Piloty, TH München, 1956), ERMETH (Eduard Stiefel, Heinz Rutishauser und Ambros Speiser, ETH Zürich, 1956), DERA (Alwin Walter, TH Darmstadt, 1957), sowie Rechner von SEL, Telefunken und Siemens.

Getrieben von den Mathematikern etablierte sich dann in den 50er Jahren die Idee der Abstraktion: Programme sollte man in einer an mathematische Formeln angelehnten Schreibweise formulieren statt im Maschinencode des jeweiligen Rechners. Das würde den Transfer eines Programms von einem Rechner zum anderen ermöglichen. Der erste, der

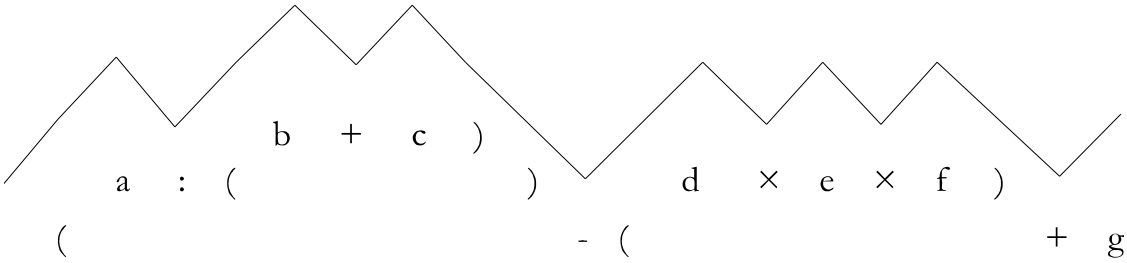


Abbildung 1.1: Rutishausers Klammergebirge

diese Idee praktisch formulierte, war Heinz Rutishauser mit dem *Klammergebirge* in seiner Habilitationsschrift 1952, Abb. 1.1.

Abstraktion ist auch die Grundlage des Blockschachtelungsprinzips, das Klaus Samelson 1954/55 erfand, (Samelson, 1957), und das sich heute in allen neueren Programmiersprachen findet. Daran schließt sich die Erfindung des Kellerprinzips durch Klaus Samelson und F.L. Bauer an, das 1959 zur Veröffentlichung der Arbeit *Sequentielle Formelübersetzung* führte, die in Amerika sofort übersetzt und auf Englisch nochmals veröffentlicht wurde, (Samelson und Bauer, 1959, 1960).

Ein besonders eindrucksvolles Beispiel von Abstraktion war die Erfindung der Netztheorie, heute als Petrinetze bekannt, durch C.-A. Petri in seiner Dissertation 1962. Dies war das weltweit erste Modellierungswerkzeug für verteilte und parallele Systeme, (Petri, 1962).

Auch die 60er Jahre sahen vor allem Fortschritte im Rechnerbau und bei Programmiersprachen und ihrer Üersetzung. Bahnbrechend war die Konstruktion des D4A, des ersten Arbeitsplatzrechners, 1959-64 durch Nikolaus J. Lehmann an der TH Dresden. Er wurde als Cellatron C 8205 ab 1965 in ca. 3000 Exemplare vom VEB Rechenelektronik in Zella-Mehlis produziert, blieb aber im Westen völlig unbekannt. Natürlich hatte er noch keinen Bildschirm und statt des noch nicht bekannten Plattenspeichers war ein Trommelspeicher eingebaut, Abb. 1.2, (Hantzschmann, 2011).

2 Die Vorbereitungen

Die Geschichte der Informatik-Studiengänge beginnt interessanterweise in Freiburg an der Saane, also in der Schweiz. Dort gründete Prof. Billetter im Jahr 1958 ein Institut für Automation und Unternehmensforschung (IAUF) und führte 1959 einen zweijährigen *postgraduate* Studiengang Automation ein, der bis 1999 angeboten wurde,(Kohlas, 2018). Das Institut hatte bei seiner Gründung viele Widerstände zu überwinden. Es war sogar vorgeschlagen worden, einen deutschen Theologen um ein Gutachten über die Vor- und Nachteile einer solchen Institutsgründung zu bitten. Das Institut war erfolgreich und bildete die Blaupause für die Errichtung des Betriebswirtschaftlichen Instituts für Organisation und Automation (BIFOA) 1963 durch Prof. Grochla an der Universität zu Köln.

In Deutschland begannen zahlreiche Universitäten in der zweiten Hälfte der 50er und Anfang der 60er Jahre Programmierkurse anzubieten, zunächst in Maschinensprache der

Abbildung 1.2: der erste PC, Quelle: Kustodie der TU Dresden

jeweils verfügbaren Rechner und dann mit den Programmiersprachen Algol 58 und vor allem Algol 60. Die Initiative zur Definition von Algol ging von F.L. Bauer, K. Samelson, damals in Mainz, H. Rutishauser in Zürich und H. Bottenbruch in Darmstadt aus. Sie gewannen internationale Unterstützung insbesondere von J. McCarthy, J.W. Backus und A.J. Perlis in den USA. Die Sprache war inhaltlich ein voller Erfolg. Sir Tony Hoare charakterisierte Algol 60 im Jahr 1973 mit dem Satz *Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.* Bei den Anwendungsprogrammierern namentlich in der Physik konnte sich Algol allerdings gegen Fortran nicht durchsetzen.

Ich selbst gab meinen ersten Algol-Kurs im WS 1962/63 an der Universität Erlangen, ein halbes Jahr nach meinem Diplom. Die Frage eines Lehrauftrags für diese Veranstaltung war in der alten Ordinarienuniversität uninteressant. So etwas wurde im Gespräch auf dem Flur entschieden. Das in Karlsruhe zur gleichen Zeit benutzte Buch Algol-*Praktikum* von Karl Nickel, (Nickel, 1964), lehrte keine Prozeduren, ließ also die wesentliche Grundlage der operationalen Abstraktion aus; das sei für die Studenten zu schwierig!

1967 beginnt Herr Prof. F.L. Bauer an der TH München eine zweisemestrige An-

fängervorlesung *Einführung in die Informationsverarbeitung*. Ich war damals der zuständige Assistent und mein Inhaltsverzeichnis von 1967 zeigt bereits alle wesentlichen Überschriften des späteren zweibändigen Werks (Bauer und Goos, 1971), das vielen Lesern bekannt sein dürfte. Es diente an vielen Universitäten als Grundlage für die Anfängervorlesungen in Informatik. Zur gleichen Zeit 1967, begann Herr Prof. N.J. Lehmann an der TU Dresden eine Vorlesung *maschinelle Rechentechnik* als Wahlfach.

Die größte internationale Wirkung erzielten die Münchner 1968 mit der Veranstaltung der *Garmisch Conference on Software Engineering*, (Naur und Randell, 1969), mit der dieser Begriff aus der Taufe gehoben wurde. Das eigentlich angestrebte *software engineering* Institut kam erst viele Jahre später an der Carnegie-Mellon Universität in Pittsburgh zustande. Die Münchner erzielten aber weiterhin große Erfolge mit ihren Sommerschulen in Marktoberdorf im Allgäu, die in den 70er Jahren der Platz waren, von dem aus sich Strukturiertes Programmieren weltweit durchsetzte. Obwohl das im ÜRF nicht vorgesehen war, wurde *software engineering* unter den Bezeichnungen *Programmiertechnik* oder *Softwaretechnik* rasch ein zentrales Studienfach der Informatik.

In der Politik hatte der Ministerrat der DDR bereits am 3. 7. 1964 ein Programm zur Entwicklung und Einführung und Durchsetzung der maschinellen Datenverarbeitung in der DDR in den Jahren 1964 bis 1970 (DDR-Ministerrat, 1964) beschlossen.

In Westdeutschland gab es beim BMwF, dem späteren BMFT, einen *Fachbeirat für Datenverarbeitung* (FDV), der im November 1967 begann, sich mit dem Bedarf und Ausbau von Lehrstühlen für die Datenverarbeitung zu befassen.[1]. Daraus resultierten im Juli 1968 *Empfehlungen zur Ausbildung auf dem Gebiet der Datenverarbeitung*, die auch der KMK, dem Wissenschaftsrat und der WRK zugeleitet wurden.

Zugrunde lag auch eine Erhebung der Firma Diebold im Auftrag des BMwF zum Bedarf an Informationsverarbeitern (akademisch und nicht-akademisch) in der deutschen Wirtschaft. Später gab es noch zwei weitere solche Erhebungen. Ihnen war gemeinsam, daß sie einen erheblichen Bedarf anzeigten, der einerseits bei der nächsten Erhebung jeweils noch erheblich übertroffen wurde und andererseits im Ministerium ungläubiges und entsetztes Staunen über die Höhe des Bedarfs auslöste. Das Ministerium tat alles, um die Zahlen herunter zu spielen. Daß sich da ein Erdbeben außerhalb physikalischer Anwendungen anbahnte, wollte man nicht wahr haben.

Das Wort *Informatik* für den neuen Studiengang taucht erstmalig im Protokoll eines ad hoc-Ausschusses „DV-Lehrstühle und -Ausbildung“ des FDV vom 19.1. 1968 unter

1. Mitglieder: Prof. Dr. Klaus Samelson (Direktor des Rechenzentrums der TH München), Prof. Dr.-Ing. Heinz Unger (Direktor des Instituts für angewandte Mathematik der Universität Bonn), Dr. Christian Fritzsche (Leiter der Abteilung Halbleiterphysik am Institut für Elektrowerkstoffe der Fraunhofer-Gesellschaft in Freiburg), Dr. Hans Joachim Stuckenberg (Leiter der experimentellen Elektronikgruppe beim Deutschen Elektronensynchroton DESY in Hamburg), Prof. Dr.-Ing. Robert Piloty (Direktor des Instituts für Nachrichtenverarbeitung der TH Darmstadt), Prof. Dr.-Ing. Wolfgang Giloi (Direktor des Instituts für Informationsverarbeitung der TU Berlin), Dipl.Ing. Meisel (Fernmeldetechnisches Zentralamt in Darmstadt), Dr. Horst Springer (Direktor des Forschungsinstituts für Funk und Mathematik in Werthoven), Prof. Dr. Karl Heinrich Weise (ordentlicher Professor für Mathematik an der Universität Kiel), Prof. Dr. Wilhelm Krelle (Direktor des Instituts für Gesellschafts- und Wirtschaftswissenschaften der Universität Bonn), Klaus Schneider (Direktor der Zentralstelle für maschinelle Dokumentation in Frankfurt), Prof. Dr. Helmar G. Frank (Direktor des Instituts für Kybernetik an der Pädagogischen Hochschule Berlin), Dr. Theo Ankel (BASF in Ludwigshafen), Karl-Heinz Loske (Mannesmann AG, Hüttenwerk Duisburg-Huckingen), Engelbert Weber (Leiter der Datenverarbeitung der Deutschen Lufthansa in Frankfurt). Als Gäste u.a.: Dr. Bretschneider (Siemens), Prof. Dr. Fränz und Dr. Radius (AEG-Telefunken), Prof. Dr. Ganzhorn (IBM)

Leitung von Prof. Piloty auf.[2]

Öffentlich bekannt wurde *Informatik* durch die Reden des damaligen Ministers Gerhard Stoltenberg, der am 28. 6. 1968 vor einer Versammlung der Max-Planck-Gesellschaft und dann nochmals am 22.7. 1968 auf einem Kongreß in Berlin ein *Überregionale Forschungsprogramm Informatk* (ÜRF) ankündigte. Die detaillierten Empfehlungen des FDV dazu wurden am 3.7. 1968 veröffentlicht, vgl. dazu (Reuse und Vollmar, 2008). Sie umfaßten die Themengebiete in Tab. 2.1.

Tabelle 2.1: Themengebiete des ÜRF

Automatentheorie und formale Sprachen
Programm- und Dialogsprachen sowie ihre Übersetzer
Rechnerorganisation und Schaltwerke
Betriebssysteme
Systeme zur Informationsverwaltung
Verfahren zur digitalen Verarbeitung kontinuierlicher Signale
Rechnertechnologie
Automatisierung technischer Prozesse mit Digitalrechnern
Rechnerunterstütztes Planen, Entwerfen und Konstruieren
Methoden zur Anwendung der Datenverarbeitung in der Medizin
Methoden zur Anwendung der Datenverarbeitung im pädagogischen Bereich
Betriebswirtschaftliche Anwendungen der Datenverarbeitung
Methoden zur Anwendung der Datenverarbeitung in Recht und öffentlicher Verwaltung

Diese Liste lag einem aus Vertretern der GAMM und der NTG bestehenden Ausschuß[3] vor, der dann im Herbst 1969 das endgültige *überregionale Forschungsprogramm Informatik* vorschlug, das von Bund und Ländern angenommen wurde. Der Begriff *überregional* war dem Artikel 91b des Grundgesetzes geschuldet, der nur bei Aufgaben überregionaler Bedeutung eine finanzielle Zusammenarbeit zwischen Bund und Ländern erlaubte. Beteiligt waren zunächst Universitäten in Berlin (TH), Bonn, Darmstadt, Erlangen, Hannover, Karlsruhe, München (TH), Stuttgart, wozu später auch Aachen, Kiel, Saarbrücken und noch später Dortmund stießen.

Nebenbei wurde am Rande der Sitzung am 15./16.9 1969 die *Gesellschaft für Informatik* (GI) gegründet.

Gleichzeitig hatte die KMK eine ebenfalls aus Mitgliedern von GAMM und NTG bestehende Kommission berufen[4], die auf der Grundlage der Arbeit des vorgenannten GAMM-NTG-Ausschusses eine Rahmenprüfungsordnung für Informatik erarbeitete. Von der KMK wurde diese allerdings erst am 1.2. 1973 in Kraft gesetzt.

3 Der Anfang

2. Informatik war damals noch ein Warenzeichen der Firma Standard Elektrik Lorenz (SEL, heute Nokia) in Stuttgart für das dort 1956/57 von Karl Steinbuch entwickelte *Informatik-System Quelle* für die Firma Quelle. Es mußte erst für den neuen Zweck freigegeben werden. Die Académie Française hatte 1966 definiert: *L'informatique est la science du traitement rationnel, notamment par machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications dans les domaines technique, économiques et sociaux.*

3. Mitglieder: die Professoren Bauer, Dörr, Händler, Kulisch, Samelson, Schlender, Unger und Weise für die GAMM und für die NTG Einsele, Giloi, Steinbuch.

4. Mitglieder: Bauer, Hotz, Piloty, Zemanek

Nach diesen Vorarbeiten ging es dann sehr schnell weiter. Der vorgenannte Ausschuß beim BMwF wurde in *Sachverständigenausschuß Überregionales Forschungsprogramm Informatik* umbenannt und begleitete den Aufbau der Informatik zunächst unter Vorsitz von Prof Piloty und dann bis 1978 unter Vorsitz von Prof. Goos.

An der TH München hatte der Informatik-Studiengang als Teil der Mathematik-Ausbildung bereits 1967 begonnen. Allerdings wurde der Studiengang Informatik erst 1973 selbständig, vgl. (Bauer, 2007).

In Dresden wurde zum 1. 1. 1969 eine Sektion Informationsverarbeitung ins Leben gerufen und Herrn Lehmanns Vorlesung von 1967 mündete im September 1969 in einen eigenständigen Diplomstudiengang Informationsverarbeitung der TH Dresden.

An der Ingenieurschule Furtwangen begann die Ausbildung 1968/69. Dort hieß das Fach noch *Informatorik*.

An der Universität Karlsruhe gründete Herr Prof. Nickel zum 1. Januar 1969 ein Institut für Informatik, das zunächst nur aus seinem Lehrstuhl bestand. Die Informatik-Vorlesungen begannen im WS 1968/69 mit vielen Lehrbeauftragten auch aus der Industrie. Im Sommer 1969 wurde eine Prüfungsordnung für Informatik beschlossen und genehmigt, so daß im WS 1969/70 von mehr als 200 Studenten das Vollstudium aufgenommen werden konnte. Die meisten Studenten kamen aus anderen Fachrichtungen (vor allem Mathematik und Elektrotechnik), deren Vordiplom für das neue Fach Informatik anerkannt wurde.

Auch in Saarbrücken und an der TH Darmstadt wurde 1969 der Studiengang Informatik begonnen. Diese Hochschulen hatten aber zunächst Schwierigkeiten bei der Personalgewinnung. 1970/71 folgten weitere Hochschulen. In der DDR richtete die Universität Leipzig einen Schwerpunkt *Mathematische Kybernetik und Rechentechnik* im Studiengang Mathematik ein.

In Österreich wurde die gesetzliche Grundlage für das Informatikstudium durch das Bundesgesetz vom 10.7. 1969 geschaffen. Noch im Jahr 1969 wurden an der Johannes Kepler Universität Linz mit Vorlesungen des Diplomstudiengangs Informatik begonnen. Die TH Wien folgte 1971.

Von den Gründervätern der Informatik sind besonders die Herren F.L. Bauer, R. Piloty und K. Samelson hervorzuheben. Die Herren Bauer und Samelson waren ursprünglich Physiker, die dann zur numerischen Mathematik wechselten. Herr Piloty war Nachrichtentechniker. Dementsprechend war der Informatikstudiengang vor allem auf die rechentechnischen Grundlagen mit den Themen Rechnerkonstruktion, Betriebssysteme, Programmiersprachen und ihre Übersetzung sowie auf physikalisch-technische Anwendungen ausgerichtet, wie das auch die Vorgeschichte nahelegt. Auch das aus dem Atomministerium hervorgegangene und von Physikern dominierte BMFT betonte diese Richtung.

Das Brot- und Buttergeschäft der kommerziellen Datenverarbeitung wurde dann zwar von Nixdorf, Siemens, Dietz und anderen Herstellern der mittleren Datentechnik aufgenommen, aber zu spät, IBM hatte das Geschäft bereits fest in der Hand. An den Informatikfakultäten in Deutschland hatten Darmstadt (Prof. Wedekind) und Erlangen-Nürnberg (Prof. Mertens) das Thema im Auge. In Karlsruhe gab es im Lehrplan eine Vorlesung *kommerzielle Datenverarbeitung*, die aber nie gehalten wurde. Anfangs gab es nur in Linz Kurse für Cobol. Sonst wurde diese Programmiersprache ignoriert.

Gravierend war auch das Fehlen der Gebiete *software engineering* (Programmier- oder Softwaretechnik), *Algorithmen und Datenstrukturen* sowie „Komplexitätstheorie“ in der

ÜRF-Liste. Natürlich wurden viele neue Algorithmen erfunden, aber meist im Zusammenhang mit der Übersetzung von Programmiersprachen oder mit numerischem Rechnen. Das Gebiet „Künstliche Intelligenz“, die Vision, daß man mit Informatik-Hilfsmitteln menschliche Intelligenz simulieren könnte, wurde von Karl Steinbuch 1961/62 in Karlsruhe begonnen, (Steinbuch, 1961a,b), geriet aber durch übertriebene Hoffnungen und Aussagen sehr schnell in Verruf. Das Thema „Datenbanken“ war in der Industrie und der Verwaltung, sowie in der 1968 errichteten Großforschungseinrichtungen GMD wichtig, aber nicht an den Univeritäten.

4 Die Karlsruher Informatik ab 1970

Nach den großen organisatorischen Leistungen von 1968/69 waren die Herren Nickel und Kulisch zu einem Forschungsjahr in die USA entschwunden. Als ich mich im Sommer 1970 über den Stand der Informatik in Karlsruhe informieren wollte, wurde ich von den anwesenden Assistenten belehrt, daß das Wichtigste sei, die Grundordnung der Universität und das Landeshochschulgesetz zu beachten. Das half mir wenig und ich mußte nach Madison, Wisconsin, fliegen, um von Herrn Nickel Genaueres zu erfahren.

Nach meiner Zusage wurde ich am 16ten September 1970 zum Vorsitzenden einer Berufungskommission für 6 Forschungsgruppenleiter bzw. Ordinarien und zum Mitglied einer weiteren Berufungskommission für die beiden weiteren Forschungsgruppenleiter gewählt, die das ÜRF den Karlsruhern zugesprochen hatte. Man stelle sich das heute vor: Da wird ein Angestellter der TU München, der noch dazu keinerlei Erfahrung in Berufungskommissionen hat, an einer anderen Universität zum Vorsitzenden gewählt. Ministerium und Rektor wären heute sicher einem Herzinfarkt nahe. Vor diesem Hintergrund lautete mein damaliger Leitspruch

Wir haben ein Ziel, und wenn dem Gesetze oder Verordnungen entgegen stehen, und das Ziel wichtig ist, so müssen sich die Gesetze und Verordnungen ändern, aber nicht das Ziel.

Mein Arbeitsbeginn am 1. Oktober 1970 in Karlsruhe war gekennzeichnet von einer Fülle sehr heterogener Aufgaben.

Der Dekan der Fakultät Mathematik, Herr Kollege Heuser, übereignete mir als Leiter des Instituts für Informatik 78 Stellen, darunter 8 Forschungsgruppenleiterstellen, und einen laufenden Etat von mehr als einer halben Million DM, die die Universität Karlsruhe dank der Bemühungen von Herrn Kollegen Nickel im Rahmen des überregionalen Forschungsprogramms Informatik (ÜRF) zugeteilt bekommen hatte. Die Universität Stuttgart hatte übrigens Stellen und laufende Mittel in gleicher Größenordnung, und im Unterschied zu zahlreichen anderen Bundesländern waren diese Zuteilungen auch Bestandteil des regulären Haushalts des Kultusministeriums.

Die Hauptaufgabe war die Leitung einer Berufungskommission zur Besetzung von sechs Forschungsgruppenleiterstellen (und die Mitgliedschaft in einer weiteren Kommission zur Besetzung der beiden restlichen Stellen). Von diesen Stellen waren im Landeshaushalt 1971 bereits 3 als Ordinariate ausgebracht; die restlichen 5 wurden 1972 in ordentliche Professuren umgewandelt.

Herr Kollege F.L. Bauer in München hatte vorhergesagt, daß es eigentlich nur ungefähr 30 berufungsfähige junge Informatiker in Deutschland gäbe; das ÜRF hatte aber insgesamt 119 zu besetzenden Forschungsgruppenleiterstellen in Westdeutschland ausgewiesen. Nun erwies sich, daß die Zahl 30 von Herrn Bauer doch wohl etwas untertrieben war, aber

trotzdem war klar, daß nur die Universitäten gewinnen konnten, die die Berufungsfrage entschlossen und mit großer Geschwindigkeit angingen. Im Unterschied zu Stuttgart zählte Karlsruhe zu dieser Gewinnergruppe.

Wir hatten bereits im Wintersemester 1970/71 alle unsere Berufungslisten fertig und es waren bereits 2 Rufe an die Herren Dr. G. Krüger und Dr. H. Wettstein auf ordentliche Professuren ausgesprochen, die zum 1.6. 1971 angenommen wurden. Zwar konnten wir mit den Berufungslisten keinen Schönheitspreis gewinnen; es handelte sich um Einer- oder Zweierlisten, aber diese Aussage bezieht sich nicht auf die berufenen Kollegen. Die Kollegen aus der Mathematik erwogen allen Ernstes, ob man in Informatik nicht auch Leute ohne Promotion berufen könne, was ich vehement bestritt. Dahinter standen Meinungen aus dem Jahr 1968, in dem generell die Notwendigkeit akademischer Grade und Titel bestritten wurde; auch unter den Karlsruher Informatikassistenten gab es Leute, die zwar Forschungsgruppenleiter werden wollten, aber zunächst den Titel eines ordentlichen Professors ablehnten.

In Summe schafften wir es, 1971 sechs der acht Gruppenleiterstellen zu besetzen, die beiden anderen folgten im Jahr 1972. Aus der Sicht der Lehre war das auch bitter nötig. Wir hatten im Frühjahr 1971 bereits 394 Studenten aber nur eine besetzte Professur. Im Wintersemester 1971/72 kamen dann 225 Anfänger dazu und ich hatte neben den noch abzuwickelnden Berufungsverfahren, dem Vorsitz des ÜRF-Ausschusses in Bonn, anderen organisatorischen Aufgaben und einer Vorlesung im Hauptstudium auch noch die vierstündige Anfängervorlesung zu halten.

Zum Gebrauch in den Anfängervorlesungen bestellten wir uns 1971 beim norwegischen Rechenzentrum in Oslo für die neu installierte Univac 1108 des Rechenzentrums einen Übersetzer für die Programmiersprache Simula 67. Das kostete uns zwar 10% unseres laufenden Etats 1971, ermöglichte uns aber eine Ausbildung in objektorientierter Programmierung, lange bevor andere Leute in Europa oder den USA auf diesen Zug aufsprangen. Als uns 1978 ein Kollege aus Xerox Parc in einem Vortrag die Vorzüge von Simula anpries, konnten wir nur grinsen.

Die Raumnot der Karlsruher Informatik war zunächst außerordentlich. Die Forschungsgruppen waren über den ganzen Campus verstreut einschließlich der Westhochschule. Außer zu Sitzungen traf man sich kaum und an eine vernünftige Zusammenarbeit war zunächst nicht zu denken. Das änderte sich erst im Frühjahr 1972 mit dem Bezug des damals neuen Informatikgebäudes, das zusammen mit dem Rechenzentrum im Schnellverfahren errichtet worden war. Damit kamen erstmalig alle Informatiklehrstühle unter ein Dach und es konnte mit dem Ausbau der technischen Infrastruktur für die Lehrstühle begonnen werden.

Dazu zählte die Anmietung eines Rechners Burroughs B6700 als zentralen Informatikrechner. Er wurde im Sommer 1972 nach einem intensiv geführten Auswahlverfahren installiert. Nominal hatte er nur die halbe Leistung eines Telefunken TR440. Letzteren konnten wir uns im Unterschied zur Universität Stuttgart nicht leisten, da unsere Rechnermietmittel eine Million DM niedriger waren als die der Stuttgarter. Die Beschaffung erwies sich jedoch als Glückstreffer. Dank der Flexibilität des Systems und mit zahlreichen Eigenleistungen konnten wir das System so ausbauen, daß 1976, als wir das System durch einen Siemens-Rechner ersetzten, die Leistung als doppelt so hoch wie die eines TR440 eingeschätzt wurde. Zu den Eigenleistungen zählten Programmierarbeiten für den Satellitenrechner, über den Bildschirmgeräte und Prozeßrechner angeschlossen waren, sowie die eigenhändige Verkabelung der Räume des Gebäudes durch die wissenschaftlichen Mitar-

beiter zum Anschluß zahlreicher Bildschirmgeräte über serielle Leitungen. Die Flexibilität zeigte sich in den Anschlußprotokollen. Wir hatten keine Probleme mit dem Anschluß von Rechnern, die nicht von Burroughs stammten, und konnten *dumme Terminals* kaufen, die weniger als die Hälfte der Sichtgeräte kosteten, die wir bei den großen Herstellern, auch den deutschen, mit ihren proprietären Anschlußprotokollen hätten aufwenden müssen. Die Preise waren im Vergleich zu heute trotzdem riesig. Ein Sichtgerät mit Tastatur kostete 23.000 DM.

Zum 1.10 1972 spaltete sich das Institut für Informatik von der Fakultät für Mathematik ab und wurde eine selbständige Fakultät mit zunächst 9 Professoren. Im Vergleich war dies die größte Informatikgruppe in Deutschland; andernorts waren entweder weniger Forschungsgruppen beantragt und genehmigt worden, oder die Berufungsverfahren waren noch nicht so weit fortgeschritten.

Organisatorisch war die Fakultätsgründung ein gewaltiger Schritt bezüglich der Einordnung des Fachs in der Universität und im öffentlichen Ansehen. Intern änderte sich zunächst nur wenig. Wir hatten zwar jetzt Sitz und Stimme im Senat der Universität und brauchten auch eine eigene Promotionsordnung. Aber die Fakultät wurde weiterhin zu 70% vom Bund durch das ÜRF finanziert und im Unterschied zu allen anderen Fakultäten wurden die Stellen und Mittel den Lehrstühlen bzw. Instituten nicht vom Kanzler zugeteilt, sondern wie im vorigen Institut für Informatik zentral durch die Fakultät verwaltet und von dort zugeteilt. Dies erleichterte die Beantragung und Abrechnung der Mittel gegenüber dem BMFT. Das Verfahren hatte auch viele andere Vorteile und wurde sehr lange beibehalten.

Nachdem wir bereits 1971 dem ersten Diplominformatiker in Deutschland sein Abschlußzeugnis überreichen konnten, gab es 1973 die erste Promotion und die erste Habilitation in der Fakultät.

Ab 1974 erhielt die Universität Karlsruhe 5 weitere Forschungsgruppen für Gebiete der angewandten Informatik. Davon wurde eine, später zwei Gruppen in die Fakultät aufgenommen; die anderen wurden in anderen Fakultäten angesiedelt, was sich sehr bewährte.

Die Zusammenarbeit insbesondere mit dem Maschinenbau war in vielerlei Hinsicht sehr fruchtbar. Wenn man uns fragte, wohin unsere Absolventen nach dem Diplom gingen, war unsere Standardantwort „in den Schwarzwald“. Viele landeten in der Leitung der Informationsverarbeitung bei kleinen und mittleren Elektro- und Maschinenbauunternehmen. Diese Zusammenarbeit mit dem Maschinenbau konnten wir 1990 nach der Wende auch in Sachsen sehen. Sachsen hatte damals an die 50.000 Diplomingenieure mit Zusatzkenntnissen in Informatik, die sie umgehend verwerten konnten, da solche Leute überall in Deutschland fehlten.

5 Und Heute ?

Zu den Grundgebieten der Informatik, der damaligen *Kerninformatik*, traten sehr bald die Telematik und graphische Datenverarbeitung sowie Anwendungsgebiete wie Wirtschaftsinformatik, Robotik, CAD/CAM, medizinische Informatik usw. Das Thema Telematik trieb in Karlsruhe Herr Kollege Krüger voran. Robotik, CAD/CAM und andere Maschinenbauanwendungen waren bei den Professoren Rembold und Grabowski angesiedelt. Herr Kollege Encarnação in Darmstadt gewann mit dem *graphischen Kernsystem* (GKS) Weltruhm.

Heute haben wir 4 grundlegende Rechenparadigmen, nämlich die von Neumann-Architektur mit Parallelverarbeitung und Rechnernetzen, zelluläre Automaten, evolutionäre oder genetische Programmierung und (künstliche) neuronale Netze. Sie werden heute alle auf von Neumann-Rechnern implementiert. Ob das in Zukunft so bleibt und ob es auch Quantenrechner mit Anwendungen außerhalb der Kryptographie gibt, werden wir sehen.

Bei Algorithmen können wir heute zwei Stoßrichtungen unterscheiden: einerseits Algorithmen, die konstruktiv ein Ergebnis berechnen (und dementsprechend mathematisch korrekt sein sollten); andererseits Algorithmen, die Muster in vorgelegten Daten erkennen, und daraus Schlüsse ziehen. Ob diese Schlüsse richtig sind, hängt von den zugrunde liegenden Daten ab, und kann nur statistisch beurteilt werden. Die Datenbasis ist im allgemeinen nicht vollständig; wenn sie zu klein oder einseitig ist, sinkt die Vertrauenswürdigkeit der Schlußfolgerungen.

Im *software engineering* haben sich die Systemgrößen vervielfacht: Vor 50 Jahren war ein Softwaresystem mit 100.000 Quellzeilen riesig. Heute ist das ein kleines System. Ansonsten kämpfen wir mit den gleichen Problemen wie damals: Die Ergebnisse der Anforderungsanalyse sind nicht ausreichend, um eine Konstruktion zu ermöglichen, die den Kunden befriedigt; die Korrektheit und Sicherheit der Abläufe ist nicht gewährleistet, weil die Implementierung der am häufigsten benutzten Programmiersprachen wie C, C++, Java, usw. erlaubt, aus Effizienzgründen die notwendigen Prüfungen zu unterdrücken.

6 Zum Schluß

Meinen Studenten von 1962 sagte ich

> Ein Rechner ist ein Vollidiot mit Spezialbegabung: Er hat ein großes, präzises Gedächtnis und kann schneller rechnen als ein Mensch

Dieser Satz gilt auch heute noch.

Zu diesem historischen Rückblick haben viele Kollegen beigetragen. Besonders möchte ich mich bei den Herren K. Hantzschmann, G. Hotz, S. Katzenbeisser, J. Kohlas, G. Merkel, H. Reichel, G. Schmidt, R. Wilhelm und C.A. Zehnder bedanken. Insbesondere die Aufklärung von Ereignissen und Nicht-Ereignissen in der DDR war nicht einfach.

Literaturverzeichnis

Bauer (2007): 40 Jahre Informatik in München: 1967 – 2007, Festschrift. Fakultät für Informatik, TU München.

Bauer, F. L. und Goos, G. (1971): *Informatik — Eine einführende Übersicht*. Springer. 2 Bde., 1. Aufl. 1971, 4. Aufl. 1991/92.

DDR-Ministerrat (1964): Programm zur Entwicklung, Einführung und Durchsetzung der maschinellen Datenverarbeitung in der DDR in den Jahren 1964 bis 1970. Bundesarchiv. (SAPMO-BArch, DY 30/J IV 2/2/936, Bl. 37-160).

Gödel, K. (1931): Über Formal Unentscheidbare Sätze der Principia Mathematica Und Verwandter Systeme I. *Mh. Math. Phys.*, 38: 173–198.

Güntsch, F.-R. (1957): *Logischer Entwurf eines digitalen Rechnergerätes mit mehreren asynchron laufenden Trommeln und automatischem Schnellspeicherbetrieb*. Dissertation, Technische Universität Berlin. D83.

Hantzschmann, K. (2011): N.J. Lehmann – der Sächsische Computerpionier. Preprints aus dem Institut für Informatik, Universität Rostock. CS-07-11.

Kilburn, T., Edwards, D., Lanigan, M. und Sumner, F. (1962): One-Level Storage System. *IRE Trans. Electronic Computers*.

Kohlas, J. (2018): Eine kleine Geschichte der Informatik an der Universität Freiburg. Universität Freiburg. www3.unifr.ch/inf/de/assets/public/misc/Geschichte der Informatik an der Universität Freiburg.pdf.

Naur, P. und Randell, B. (Hrsg.) (1969): *Software Engineering*. NATO Science Committee. Report on a Conference, Garmisch, 7.–11. Oct. 1968.

Nickel, K. (1964): *ALGOL-Praktikum; eine Einführung in das Programmieren*. G. Braun Verlag Karlsruhe.

Petri, C.-A. (1962): Kommunikation mit Automaten. Techn. Ber., Schriften des Instituts für Instrumentelle Mathematik, Bonn.

Reuse, B. und Vollmar, R. (Hrsg.) (2008): *Informatikforschung in Deutschland*. Springer Verlag.

Samelson, K. (1957): Probleme der Programmierungstechnik. In *Internationales Kolloquium über Probleme der Rechentechnik 1955*, S. 61–68, Berlin.

Samelson, K. und Bauer, F. L. (1959): Sequentielle Formelübersetzung. *Elektron. Rechenanlagen*, 1: 176–182.

Samelson, K. und Bauer, F. L. (1960): Sequential Formula Translation. *Commun. ACM*, 3(2): 76–83.

Schmidt, G. (2018): Rückblick auf die Anfänge der Münchner Informatik. Manuskript.

Steinbuch, K. (1961a): *Automat und Mensch. Über menschliche und maschinelle Intelligenz*. Springer-Verlag Wien.

Steinbuch, K. (1961b): Die Lernmatrix. *Kybernetik*, 1: 36–45.

Wilhelm, R. (2009): Ein Flaggschiff der neuen Zeit. *Saargeschichte n*, 2009(2).

Compile-time Verification of Declarative Programs with SMT Solvers

– Extended Abstract –

Michael Hanus

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

Abstract. Although declarative languages provide a higher programming level than imperative ones, declarative programs can still crash at run-time due to unintended uses of operations. For instance, operations might have contracts which cannot be expressed with standard type systems so that they have to be checked at run time, or partially defined operations fail when they are invoked with unintended arguments. In order to obtain reliable programs, it is desirable to verify at compile time that such failures cannot occur at run time. In this talk we present techniques to use SMT solvers for this task. Since SMT solvers are powerful automatic verification tools, we develop fully automatic methods to verify contracts and the absence of non-failure situations at compile time. Although these methods cannot verify all kinds of contracts, they are successful for many non-trivial verification tasks. We present our approach for programs written in the functional logic language Curry so that it is also applicable to purely functional or purely logic programs.

1 Contract Checking

Statically typed programming languages are useful to develop reliable software since they allow the detection of specific classes of run-time errors at compile time. Unfortunately, not all requirements on operations can be expressed by standard static type systems. Hence, one can either refine the type system, e.g., use a dependently typed programming language and a more sophisticated programming discipline [6], or add *contracts* with pre- and postconditions to operations. Here, we follow the latter approach since it provides a smooth integration into existing software development processes. For instance, consider the well-known factorial function (in Haskell syntax):

```
fac n = if n==0 then 1
                else n * fac (n-1)
```

Although `fac` is intended to work on non-negative natural numbers, standard static type systems cannot express this constraint so that one has to provide a more general type like

```
fac :: Int → Int
```

Although this type annotation avoids the application of `fac` on characters or strings, applications on negative numbers are allowed and result in an infinite loop.

Such unintended applications can be avoided by specifying a *precondition*, i.e., a Boolean expression to restrict the applicability of an operation. Following the notation proposed in [1], a precondition for an operation f is a Boolean operation with name f`'pre`. For instance, a precondition for `fac` is

```
fac'pre n = n >= 0
```

Such a precondition can be checked at run time before each invocation of `fac`. Then an application to `fac` to a negative number results in a run-time error (contract violation) instead of an infinite loop.

Unfortunately, run-time contract checking requires additional execution time so that it is often turned off, in particular, in production systems. To improve this situation for declarative languages, we propose to reduce the number of contract checks by (automatically) verifying them at compile time. Since we do not expect to verify all of them at compile time, our approach can be seen as a compromise between a full static verification, e.g., with proof assistants like Agda, Coq, or Isabelle, which is time-consuming and difficult, and a full dynamic checking, which might be inefficient. In the best case, all contracts are verified so that we have a high confidence in the quality of the entire software system.

In this work, we use SMT solvers [2] for the verification task. For instance, we can verify that the precondition for the recursive call to `fac` is always satisfied by the following reasoning:

1. If `fac` is applied to some argument n, we know that $n \geq 0$ since the precondition must hold for this invocation.
2. Since the `else`-branch is selected for a recursive call to `fac`, the `if`-condition does not hold so that $\neg(n = 0)$ holds.
3. Altogether, we know that $n \geq 0 \wedge \neg(n = 0)$ holds before the recursive call to `fac`. Since this implies that $n > 0$ and, thus, $(n - 1) \geq 0$ holds (in integer arithmetic), we know that the precondition of the recursive call to `fac` always holds.

Since we have shown that the precondition of the recursive call is always satisfied, we can omit the precondition checking for these calls. As a consequence, $n - 1$ precondition checks are avoided when we evaluate `fac` n.

As a proof-of-concept, we made this idea more precise for the functional logic language Curry [5], an extension of Haskell with non-determinism and logic programming features so that our results can also be applied to purely functional and purely logic languages. In particular, we defined an *assertion-collecting semantics* which can be interpreted as an abstraction of the concrete semantics with the following differences:

1. Instead concrete values, symbolic values are considered.
2. Properties that are known to be valid, also called *assertions*, are collected.
3. To ensure termination, recursive function calls are not evaluated but only their pre- and postconditions are collected.

The actual verification tool uses this semantics to collect assertions before function calls and at the end of a function's right-hand side and sends them to an SMT solver. The details of this verification approach are described in [3] where the approach is also practically evaluated. Since SMT solvers like Z3 [2] provide good reasoning on integer theories, many program contracts involving domain restrictions on integers, as shown with `fac`, or size restrictions on data (e.g., the length of the output of reversing a list is identical to the length of the input list) can be verified in a fully automatically manner. Since the verification results are used to reduce the number of dynamic contract checks, one can obtain speedups by orders of magnitudes.

Apart from the efficiency improvement, verified contracts increase the confidence in the software system. By verifying postconditions, one can actually show the correctness of functions. For instance, consider the recursive function that sums up all natural numbers:

```
sum :: Int  → Int
sum n = if n==0 then 0
                else n + sum (n-1)
```

The precondition requires that the argument must be non-negative, and the postcondition specifies the correctness of this function by Gauss' sum formula:

```
sum'pre  n   = n>=0
sum'post n f = f == n * (n+1) `div` 2
```

Our method allows a fully automatic verification of this postcondition by showing that it is implied by the assertions collected at the end of each branch of the function's definition.

2 Fail-Free Verification

The occurrence of failures during a program execution is an annoying but frequent problem when software systems are developed. External reasons for such failures, like missing files, access rights, or unexpected formats of external data, can be caught by exception handlers in order to avoid a crash of the entire software system. However, there are also failures due to programming errors, like calling a partially defined operation with unintended arguments. In imperative programs a typical error of this kind is dereferencing a pointer variable whose current value is the null pointer. Although such kinds of errors cannot occur in declarative programs, there are other errors typical for declarative programming, like failures due to incomplete pattern matching. For instance, consider the following operations (in Curry or Haskell syntax) which compute the first element and the tail of a list:

```
head :: [a]  → a
head (x:xs) = x
```

```
tail :: [a]  → a
tail (x:xs) = xs
```

Since the case for empty lists is not given, the operations are partially defined so that a computation runs into a failure when these operations are applied to an empty list, as in the expression "`head (tail [1])`". Hence, a careful programmer

must ensure that such calls do not occur at run time. For instance, if partially defined operations are applied to unknown data, e.g., given by the user at run time, it is good style to check the arguments before the actual application. This is done in the following code snipped which defines an operation to read a command together with some arguments from standard input (the operation `words` breaks a string into a list of words separated by white spaces):

```
readCommand = do
  putStr "Input a command:"
  s <- getLine
  let ws = words s
  if null ws then readCommand
             else processCommand (head ws) (tail ws)
```

Here, the calls to `head` and `tail` cannot lead to a failure since it is ensured that the variable `ws` is bound to a non-empty list when entering the `else` branch. Since programming with failures is a typical programming technique in functional logic programming, we do not want to ban all failures but only those that lead to a failure at the top level of execution. Programs without such top-level failures are called *fail free*. For instance, failing calls to `head` and `tail` are acceptable when one searches for list structures satisfying some constraints and this search is encapsulated in the main computation.

For this purpose, we need specific contracts: we annotate operations with *non-fail conditions*. For instance, a non-fail condition for `head` is

```
head'nonfail xs = not (null xs)
```

The intended meaning of a non-fail condition is that the evaluation of an operation applied to arguments satisfying the non-fail condition never fails. Non-fail conditions are different from *preconditions* since the latter must be satisfied for any call of an operation. Thus, if we take the above non-fail condition for `head` as a precondition, it is not allowed to use `head` in logic computations to search for non-empty lists. For instance, the integer division operation `div` is a partial operation in Curry, i.e., it fails (and does not crash) if the second argument is zero. Hence, it has the following non-fail condition:

```
div'nonfail :: Int → Int → Bool
div'nonfail x y = y/=0
```

In order to show that a program is fail-free, one has to verify that the trivial non-fail condition

```
main'nonfail = True
```

is correct for the `main` function.

Again, SMT solvers are an appropriate tool for this verification task. For instance, consider the operation to compute the sign of an integer:

```
sig :: Int → Int
sig x | x>0  = 1
      | x==0 = 0
      | x<0  = -1
```

Guarded rules are an abbreviation for nested if-then-else expressions, i.e., the rule will be desugared to

```
sig x = if x>0 then 1
               else if x==0 then 0
                            else if x<0 then -1
                                        else failed
```

In order to show that the trivial non-fail condition

```
sig'nonfail x = True
```

is correct, one has to show that `sig` does not fail on any integer. This amounts to showing that the last alternative with the call to `failed` is not reachable. This is easy to see: to reach the last alternative, all previous `if`-conditions are not satisfied. Since the formula

$$\neg(\texttt{x>0}) \wedge \neg(\texttt{x==0}) \wedge \neg(\texttt{x<0})$$

is unsatisfiable, which can be automatically proved by an SMT solver, the last alternative is not reachable so that the non-fail condition is correct.

To formalize this reasoning, a concrete semantics of Curry covering failures together with an appropriate assertion-collecting semantics is developed in [4]. The corresponding tool uses the assertion-collecting semantics to extract conditions from a program which ensure that all non-fail conditions are correct and sends the conditions to an SMT solver. The practical evaluation demonstrates that many modules, including the large prelude module of Curry, can be automatically verified by inserting a few non-trivial non-fail conditions. If the verification tool cannot prove the correctness of all non-fail conditions, it points to the potential failure situations. These can either be corrected (e.g., by adding missing branches or calling an operation only after additional checks) or, if it is intended, by strengthening the non-fail conditions.

References

1. S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012.
2. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer LNCS 4963, 2008.
3. M. Hanus. Combining static and dynamic contract checking for Curry. In *Proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017)*, pages 323–340. Spriner LNCS 10855, 2017. Long version to appear in Fundamenta Informaticae.
4. M. Hanus. Verifying fail-free declarative programs. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming(PPDP 2018)*, pages 12:1–12:13. ACM Press, 2018.
5. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-lang.org`, 2016.
6. A. Stump. *Verified Functional Programming in Agda.* ACM and Morgan & Claypool, 2016.

Parallele Programmierung mit MOSTflexiPL

Christian Heinlein

Hochschule Aalen – Technik und Wirtschaft
christian.heinlein@hs-aalen.de

Extended Abstract

MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language, siehe http:// flexipl.info) ist eine Programmiersprache, deren Syntax vom Anwender nahezu beliebig erweitert und angepasst werden kann. Aufbauend auf einer kleinen Menge vordefinierter Grundoperatoren, können nach Belieben weitere Operatoren für unterschiedlichste Zwecke definiert werden. Da Operatoren beliebig viele Namen besitzen und auf beliebig viele Operanden angewandt werden können, decken sie neben den üblichen Präfix-, Infix- und Postfix-Operatoren (z. B. `-x` für Vorzeichenwechsel, `x+y` für Addition oder `n!` für Fakultät; die Operatornamen sind jeweils fett gedruckt) auch Mixfix-Operatoren (z. B. `a[i]` für Array-Elementzugriff), Kontrollstrukturen (z. B. **`if`** `x` **`then`** `y` **`else`** `z` **`end`**), Typkonstruktoren (z. B. **`List`** `T`) und Deklarationsformen (z. B. `S` **`->`** `T`, um `S` als Untertyp von `T` zu deklarieren) ab.

Darüber hinaus gibt es einige wenige Grundoperatoren für parallele Programmierung:

- `x||y` führt die Teilausdrücke `x` und `y` parallel aus und wartet auf die Beendigung von beiden, während `||x` den Teilausdruck `x` asynchron ausführt, ohne auf seine Beendigung zu warten.
- `s : sem` deklariert ein Semaphor `s` mit Anfangswert 0.
  `s!+` bzw. `s!-` entsprechen den Operationen V und P von Dijkstra, d. h. sie erhöhen bzw. erniedrigen das Semaphor `s` um 1, wobei `s!-` anschließend ggf. wartet, bis der Wert des Semaphors wieder mindestens 0 ist.

Aufbauend auf diesen Grundoperatoren, kann jeder Programmierer ohne große Mühe „höherwertige“ Operatoren für parallele Programmierung realisieren, zum Beispiel:

- Eine `for`-Schleife, deren Schleifenrumpf für jeden Wert der Laufvariablen parallel ausgeführt wird
- Variablen, die immer abwechselnd geschrieben und gelesen werden müssen (sog. MVars in Haskell)
- Asynchron berechnete Werte (sog. futures), Monitore u. v. a. m.

Damit eignet sich MOSTflexiPL auch hervorragend zum Experimentieren mit neuen Sprachkonstrukten, weil sich diese häufig in wenigen Minuten implementieren lassen.

Eine weitere nützliche Grundoperation, die zunächst nichts mit paralleler Programmierung zu tun hat, ist `abort`, um Ausführungen vorzeitig abzubrechen. Im folgenden Beispiel wird ein Array `a` der Größe `N` nach einem Wert `x` durchsucht. Sobald der Wert gefunden wurde, wird die `for`-Schleife, die mit der Marke `search` gekennzeichnet ist, mittels `abort search` abgebrochen:

```
search ::
for i = 1 .. N do
    if a[i] = x then
        abort search
    end
end
```

Allerdings gibt es zwischen `abort` und der parallelen Ausführung `...||...` eine interessante Wechselwirkung, die im folgenden Beispiel illustriert wird:

```
search :: (
    for i = 1 .. N/2 do
        if a[i] = x then
            abort search
        end
    end
||
    for i = N/2 + 1 .. N do
        if a[i] = x then
            abort search
        end
    end
)
```

Um die Suche nach dem Wert `x` zu beschleunigen, werden die beiden Hälften des Arrays parallel durchlaufen. Sobald der Wert in einer Hälfte gefunden wurde, soll die gesamte Suche abgebrochen werden. Deshalb führt ein `abort` in einem Zweig einer parallelen Ausführung auch zum Abbruch des jeweils anderen Zweigs.

Das folgende Beispiel zeigt eine weitere nützliche Anwendung dieses Prinzips:

```
search :: ((
    for i = 1 .. N do
        if a[i] = x then
            abort search
        end
    end;
    abort search
) || (
    sleep 2000;
    abort search
))
```

Die Suche nach dem Wert `x` soll nach spätestens 2000 Millisekunden abgebrochen werden. Hierfür wird parallel zur Suche eine entsprechend lange `sleep`-Operation ausgeführt, nach deren Beendigung `abort search` ausgeführt wird, wodurch auch die Schleife im anderen Zweig der parallelen Ausführung abgebrochen wird. Wenn die Schleife bereits früher beendet ist, wird anschließend ebenfalls `abort search` ausgeführt, wodurch umgekehrt auch die `sleep`-Operation im anderen Zweig abgebrochen wird.

Mit dem gleichen Mechanismus ist es auch möglich, Operationen vorzeitig abzubrechen, die prinzipiell „endlos“ lange dauern können, z. B. das Warten auf eine Benutzereingabe oder das Warten auf die Freigabe eines Semaphors. Das folgende Beispiel zeigt einen allgemein verwendbaren Operator `try ... for ... ms`, der als Operanden eine beliebige (unausgewertete) Aktion `action` (mit beliebigem Typ `T`) und eine maximale Wartezeit `time` in Millisekunden erhält:

```
try [<T:type>] <{action:T}> for <time:int> ms : bool
{
    wait :: ((
        action;
        abort wait
    ) || (
        sleep time;
        abort wait
    ))
}
```

Eine Anwendung dieses Operators wie z. B. `try s!- for 2000 ms` führt dann die Aktion `action` (im Beispiel `s!-`) und die Operation `sleep time` (im Beispiel `sleep 2000`) parallel aus. Sobald eine der beiden Ausführungen beendet ist, wird der gesamte Block, der mit `wait` gekennzeichnet ist, abgebrochen, woraus folgt, dass die Ausführung von `action` nach spätestens `time` Millisekunden abgebrochen wird.

Dieses Beispiel zeigt erneut das zentrale Grundprinzip von MOSTflexiPL: Aufbauend auf einer kleinen Menge essentieller Grundoperationen, kann jeder Programmierer ohne große Mühe nach Belieben komplexere Operationen definieren. Wenn bestimmte solcher Operationen häufiger benötigt werden, kann man sie in einer Bibliothek aufbewahren, die dann in unterschiedlichen Programmen verwendet werden kann.

Implementierungstechnisch birgt das Zusammenspiel zwischen `abort` und parallelen sowie asynchronen Ausführungen einige Herausforderungen, die durch sorgfältige Analysen der möglichen Wechselwirkungen und semiformale Korrektheitsüberlegungen gemeistert wurden. Damit blockierende Operationen wie `sleep`, `read` und `...!-` durch `abort` abgebrochen werden können, ohne hierfür systemabhängige Spezialmechanismen zu verwenden, mussten sie stellenweise „unkonventionell“ implementiert werden. Auf diese Weise konnte das gesamte MOSTflexiPL-Laufzeitsystem vollkommen portabel in C++11 implementiert werden.

Mining von BPMN-Prozessartefakten auf GitHub

Thomas S. Heinze[1], Viktor Stefanko[2] und Wolfram Amme[2]

[1] Institute of Data Science
German Aerospace Center (DLR)
thomas.heinze@dlr.de
[2] Institute of Computer Science
Friedrich Schiller University Jena
[wolfram.amme,viktor.stefanko]@uni-jena.de

Zusammenfassung. Die Validierung von Programmanalysen und Analysewerkzeugen erfolgt häufig anhand einer Sammlung von Beispielprogrammen. Für eine aussagekräftige Bewertung sollte diese Sammlung repräsentativ sein. Gerade im Bereich der Analysen für domänenspezifische Sprachen wird zu diesem Zweck aber oft nur auf Fallstudien oder gar einzelne künstliche Programmbeispiele zurückgegriffen. In diesem Beitrag zeigen wir für die domänenspezifische Sprache BPMN, wie durch systematisches Mining auf GitHub.com ein Corpus von BPMN-Prozessen für die Validierung von Analysewerkzeugen generiert werden kann.

1 Einführung

Ansätze zum Repository-Mining, das heißt zur systematischen Gewinnung, Aufbereitung und Analyse von Daten aus Code-Repositorys, erfreuen sich mit der wachsenden Bedeutung von Plattformen wie etwa GitHub.com, GitLab.com, Bitbucket.org und SourceForge.net und den damit zur Verfügung stehenden Datenquellen steigender Beliebtheit. So lassen sich vielfältige Fragestellungen zu Softwareentwicklungsprozessen und -produkten auf Grundlage der in Code-Repositorys enthaltenen Informationen untersuchen und Hypothesen zur Softwareentwicklung empirisch testen. Steht in den meisten Arbeiten noch klassischer Quellcode im Vordergrund, wird das zugrundeliegende Prinzip immer mehr auch auf andere Produkte der Softwareentwicklung übertragen. Als exemplarisches Beispiel sei auf das Mining vom UML-Modellen auf GitHub.com verwiesen [6,10], welches auch die Inspiration für den in diesem Beitrag vorgestellten Ansatz gab.

Gerade im Bereich der domänenspezifischen Sprachen könnten, wie schon in [10] angedeutet, Ansätze zum Repository-Mining die empirische Forschung unterstützten. So fehlen oft umfassende und insbesondere repräsentative Sammlungen von Programmbeispielen zur Untersuchung von Fragen zur praktischen Sprachverwendung oder zur Sprachunterstützung durch Modellierungs- und Analysewerkzeuge. Zudem stehen oft Anforderungen an den Schutz geistigen Eigentums der Verwendung von realen Programmbeispielen aus einem industriellen oder wirtschaftlichen Kontext gegenüber. Stattdessen wird beispielsweise in

Forschungsarbeiten zu Modellierungs- und Analysewerkzeugen zu Validierungszwecken immer wieder auf künstliche Programmbeispiele oder einzelne Fallstudien zurückgegriffen [10], die natürlich nur eine eingeschränkte Aussagekraft besitzen. In diesem Beitrag greifen wir daher den Ansatz aus [6,10] auf und wenden ihn auf die domänenspezifische Sprache zur Geschäftsprozessmodellierung BPMN an. Dazu generieren wir durch systematisches Mining auf `GitHub.com` einen Corpus von BPMN-Prozessen, den wir anschließend zur Validierung von Aussagen zur praktischen Relevanz eines Analysewerkzeugs für BPMN nutzen.

Business Process Model and Notation (BPMN). Die Sprache BPMN definiert ein standardisiertes Format zur IT-Unterstützung von Geschäftsprozessen. Dabei spannt BPMN den Bogen von Domänenexperten, die sich für die Prozessmodellierung verantwortlich zeichnen, zur IT, die Prozesse bis hin zu deren vollautomatischen Ausführung implementiert. Der Sprachstandard in [1] spezifiziert zu diesem Zweck neben einer grafischen Notation zur Prozessmodellierung auch ein XML-Dialekt zur Serialisierung und eine Sprachsemantik. Neben einer Reihe von Entwicklungsumgebungen zur Modellierung stehen verschiedene Ausführungsumgebungen, wie etwa *Activiti*[3] oder *Camunda*[4], zur Verfügung. BPMN ist eine komplexe Sprache und die Prozessmodellierung ist aufgrund des Sprachentwurfs zum Teil recht fehleranfällig. Dementsprechend wichtig sind Analysewerkzeuge zur Unterstützung der Prozessmodellierung. Primär im Forschungsumfeld existieren mehrere Arbeiten zur Prozessanalyse. Die vorgeschlagenen Ansätze decken das Spektrum, beginnend bei *Linter*-Werkzeugen zur Aufdeckung von Verstößen gegen einfache Syntax- und Semantikregeln aus dem Sprachstandard, bis hin zu Techniken und Werkzeugen zur statischen Analyse und formalen Verifikation, wie in [7,8], ab. In diesem Beitrag betrachten wir beispielhaft das Analysewerkzeug *BPMNspector*[5] [4] aus der ersten Kategorie.

Die Schwierigkeiten der Generierung einer repräsentativen Sammlung von Prozessbeispielen für Forschungszwecke sind in der Fachdomäne der Geschäftsprozesse allgemein bekannt und nicht zuletzt durch deren Verwendung begründet: Der mit BPMN modellierte Prozess ist oft das Produkt und unterliegt einer entsprechenden Geheimhaltung [11]. In der Folge überwiegen auch die oben bereits erwähnten Fallstudien und künstlichen Beispiele bei der Validierung von Analysewerkzeugen und -techniken für BPMN [2,4,8,7,9]. Auf das den Autoren einzige bekannte Projekt *BenchFlow*[6] zu einer umfassenden Prozesssammlung kann nicht öffentlich zugegriffen werden (insgesamt 8.363 Modelle, Anteil BPMN: 64% [11]).

Der Beitrag gliedert sich wie folgt: In Abschnitt 2 wird der Mining-Prozess vorgestellt, mit dessen Hilfe Prozessartefakte in Repositorys auf `GitHub.com` gesucht und ein Corpus von BPMN-Prozessen generiert werden konnte. Die Ergebnisse des Mining-Prozesses und der nachfolgenden Analyse auf dem generierten Corpus bildet den Gegenstand von Abschnitt 3. Dabei wird sowohl auf

[3] `https://www.activiti.org` (21.8.2019)
[4] `https://camunda.com` (21.8.2019)
[5] `https://github.com/uniba-dsg/BPMNspector` (21.8.2019)
[6] `http://design.inf.usi.ch/research/projects/benchflow` (21.8.2019)

Charakteristika der identifizierten Prozessartefakte als auch auf die Resultate der Anwendung des Analysewerkzeugs *BPMNspector* eingegangen. Der Beitrag schließt mit einer Zusammenfassung und einer Diskussion zu Einschränkungen für die Interpretation der Ergebnisse des Beitrags in Abschnitt 4.

2 Mining-Prozess und Analyse

Das systematische Repository-Mining ermöglicht die Analyse der in heutigen Code-Repositorys vielfältig vorliegenden Daten zu Softwareentwicklungsprozessen und -produkten. Der Mining-Prozess gliedert sich dabei in mehrere Schritte: Zieldefinition, Datenauswahl, Datengewinnung und -extraktion, Datenvorverarbeitung und -bereinigung, sowie Datenanalyse und schließlich Interpretation der gewonnenen Ergebnisse. Als wichtige Quellen für die Datengewinnung dienen insbesondere öffentliche Repositorys auf Plattformen wie `GitLab.com`, `Bitbucket.org`, `SourceForge.net` oder `GitHub.com`, wobei letztere die wohl führende Plattform für Open-Source-Software ist. Zum Mining auf `GitHub.com` lassen sich grundsätzlich verschiedene Ansätze nutzen:

GitHub API. Einerseits wird durch `GitHub.com` die REST-Schnittstelle *GitHub API*[7] bereitgestellt. Über die Schnittstelle lassen sich verschiedene Anfragen zu Code-Repositorys stellen, so etwa zur Repository-Struktur, aber auch zu Metadaten wie beispielsweise den Autoren oder der dominanten Programmiersprache eines Repositorys. Jedoch ist die Anzahl von möglichen Anfragen limitiert, so sind bis zu 60 Anfragen pro Stunde ohne Authentifizierung und bis zu 5.000 Anfragen pro Stunde mit Authentifizierung erlaubt. Für das systematische und effektive Mining ist nur die letzte Variante sinnvoll.

GHTorrent. Alternativ stellt das Projekt *GHTorrent*[8] eine periodisch gespiegelte Datenbank zu allen Code-Repositorys auf `GitHub.com` zur Verfügung [5]. An diese lassen sich Anfragen entweder direkt über eine Webanwendung stellen oder die Datenbanktabellen werden in eine lokale Datenbank importiert und anschließend ausgewertet. Im Fokus des Projekts stehen jedoch Metadaten, so dass etwa die Repository-Struktur nicht in der Datenbank enthalten ist.

Web Scraping. Weiterhin besteht die Möglichkeit, die Webseite von `GitHub.com` zu nutzen, um Informationen zu Code-Repositorys durch Datenextraktion aus den HTML-Seiten der Webseite zu gewinnen.

Git. Schließlich kann bei Kenntnis der URL eines Repositorys dieses natürlich auch mittels *Git* heruntergeladen werden. Dies ist jedoch zeitaufwändig und für das systematische Mining nur eingeschränkt sinnvoll, da neben den Inhalten eines Repositorys beispielsweise auch immer dessen Historie mit geladen wird.

[7] `https://developer.github.com/v3` (21.8.2019)
[8] `http://ghtorrent.org` (21.8.2019)

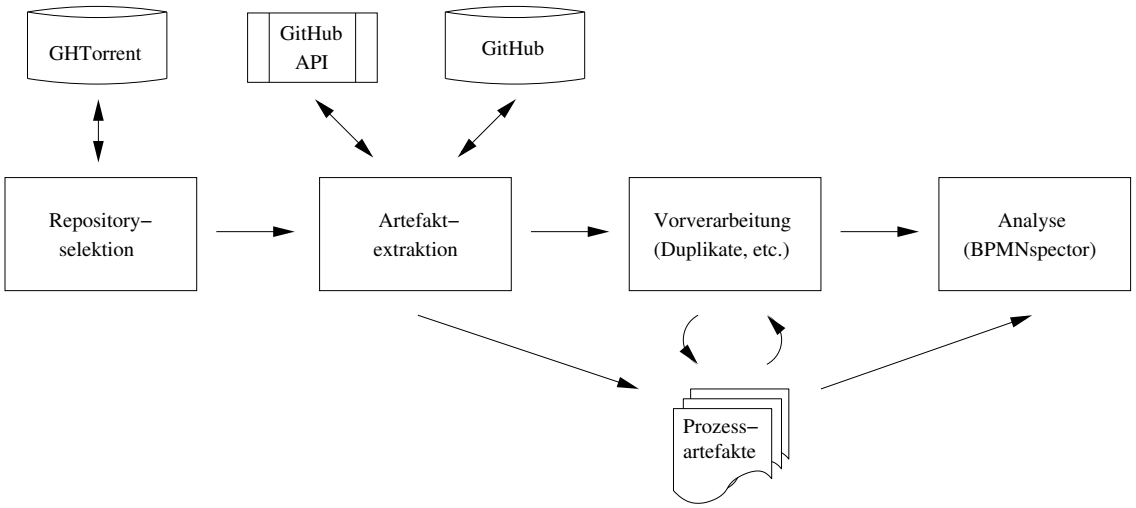


Abb. 1. Schematische Darstellung des realisierten Mining-Prozesses

Für unsere Arbeit wurde analog [6,10] eine Kombination der Ansätze gewählt (siehe auch Abbildung 1). Im ersten Schritt wurde unter Verwendung von *GHTorrent* eine Liste aller Code-Repositorys auf `GitHub.com` erzeugt. Anschließend wurden gelöschte Repositorys oder abgeleitete Repositorys (sogenannte *forked* Repositorys) aus dieser Liste wieder entfernt. Im nächsten Schritt wurden für eine Auswahl der verbleibenden Repositorys via *GitHub API* relevante Metadaten sowie die Repository-Struktur abgefragt und in einer lokalen Datenbank gespeichert. Anhand der Repository-Struktur wurde im Standardzweig (*Default Branch*) eines Repositorys nach möglichen BPMN-Prozessartefakten gesucht. Als Suchkriterium diente dabei das Auftreten der Zeichenkette `"bpmn"` im Dateinamen. Falls entsprechende Artefakte identifiziert werden konnten, wurde das Code-Repository mittels *Git* heruntergeladen und das Repository als auch die Artefakte mit der Datenbank verknüpft. Die dadurch gewonnenen BPMN-Prozessartefakte wurden anschließend weiter verarbeitet. So wurden alle Duplikate entfernt und die Dateien ihren Inhalten nach weiter selektiert. Einbezogen wurden Dateien im XML-Format mit dem Standardnamensraum `"http://www.omg.org/spec/BPMN/20100524/MODEL"`, als Indikator auf BPMN-Prozesse der Sprachversion 2.0 [1]. Im Ergebnis konnte auf diese Weise ein Corpus von in XML serialisierten BPMN-Prozessen generiert werden. Im Anschluss erfolgte der eigentliche Analyseschritt auf den gefundenen BPMN-Prozessen durch das Analysewerkzeug *BPMNspector*, dessen Ergebnisse zur weiteren Auswertung ebenfalls in die Datenbank geschrieben wurden.

3 Ergebnisse

Der vorangehend beschriebene Mining-Prozess wurde in Python implementiert und im Frühjahr 2019 ausgeführt (Implementierung und Ergebnisse online[9] [12]). Auf Grundlage des *GHTorrent*-Datensatzes vom 1.11.2018 wurden 61.632.173

[9] `https://github.com/ViktorStefanko/BPMN_Crawler` (21.8.2019)

Dateialter (Jahre)	< 1	1	2	3	4	5	6	> 6	k.A.
Anzahl Artefakte	7.656	5.154	3.291	2.344	712	690	985	404	70

Anzahl Änderungen (inkl. Erstellung)	1	2	3	4	> 4	k.A.
Anzahl Artefakte	16.285	3.378	620	572	427	24

Anzahl Duplikate	keine	1	2	3	4	5	6	7	> 7
Anzahl Artefakte	8.030	652	571	193	110	219	272	342	118

Dateiformat	Anzahl Artefakte
XML (BPMN 2.0)	16.907
XML (Andere)	384
Graphikformat (.jpg/.pdf/...)	1.635
Andere	2.380

Anzahl Entwickler	Anzahl Artefakte
1	19.595
2	982
3	605
> 3	100
k.A.	24

Tabelle 1. Charakteristika gefundener BPMN-Artefakte (insgesamt 21.306 Artefakte)

nicht gelöschte und nicht abgeleitete Code-Repositorys auf `GitHub.com` ermittelt. Da aufgrund von Zeitbeschränkungen nicht alle Repositorys berücksichtigt werden konnten, wurden analog [6] zufällig 10% ausgewählt. Das Mining von Prozessartefakten nahm in paralleler Ausführung mit vier Programminstanzen knapp 31 Tage in Anspruch. Diese Laufzeit ist vor allem der Limitierung von Anfragen durch die *GitHub API* geschuldet und kann durch Verwendung einer größeren Zahl von Programminstanzen und *GitHub API*-Nutzerkonten verringert werden. Nach Abschluss des Mining-Prozesses konnten unter den analysierten 6.163.217 Code-Repositorys, 1.251 Repositorys mit Prozessartefakten identifiziert werden, wobei in diesen insgesamt 21.306 mögliche BPMN-Artefakte, das heißt Dateien mit der Zeichenkette `"bpmn"` im Namen, gefunden wurden.

3.1 Charakterisierung der Prozessartefakte

Im nächsten Schritt wurden zunächst Metadaten für die identifizierten Repositorys und BPMN-Prozessartefakte untersucht. So lassen sich mittels des Werkzeugs *Code Maat*[10] das Alter gefundener Artefakte, die Anzahl an Änderungen und Entwicklern bestimmen (siehe auch Tabelle 1). Zu diesem Zweck wertet das Werkzeug die Versionshistorie der Artefakte in den Code-Repositorys aus.

Im Ergebnis wurde für die überwiegende Zahl von Prozessartefakten (16.101 oder 76%) ein Alter, das heißt ein Zeitraum seit Erstellung oder letzter Änderung, von nicht mehr als zwei Jahren bestimmt. Dies kann zum einen sicher auf den sprunghaften Anstieg der Anzahl von Code-Repositorys auf `GitHub.com` in den letzten Jahren zurückgeführt werden, lässt sich zum anderen aber als

[10] `https://github.com/adamtornhill/code-maat` (21.8.2019)

Indiz für die wachsende Verbreitung von BPMN auffassen. In Übereinstimmung zum Mining von UML-Modellen in [6,10] konnte für die große Mehrheit von BPMN-Prozessartefakten (16.285 oder 76%) festgestellt werden, dass es sich um statische, das heißt nicht modifizierte Inhalte von Code-Repositorys handelt. Mehrere Interpretationen sind für dieses Ergebnis möglich. So kann BPMN analog UML in der Prozessmodellierung, etwa zu Dokumentationszwecken, genutzt werden und daher eine eher statische Rolle in Repositorys spielen [6]. Auch ist der Einsatz von BPMN-Artefakten in Testsammlungen für den Test von Ausführungsumgebungen wie *Activiti* oder *Camunda* denkbar. Tatsächlich deuten die Dateinamen eines Teils der gefundenen BPMN-Prozesse genau darauf hin (beispielsweise `"InclusiveGatewayTest.testLoop.bpmn20.xml"`). Eine tiefergehende Analyse bleibt weiterführenden Arbeiten überlassen. Noch deutlicher als für die Anzahl an Änderungen fällt das Resultat für die Anzahl an Entwicklern von BPMN-Artefakten aus. So ergibt sich für einen geringen Anteil von Artefakten (1.687 oder 8%) eine Beteiligung von mehr als einem Entwickler.

Weiterhin wurde untersucht, inwiefern es sich bei den BPMN-Artefakten um originale Repository-Inhalte handelt. Knapp die Hälfte der Prozessartefakte (10.599) konnten dabei mittels des Werkzeugs *Duplicate Files Finder*[11] als Duplikate anderer Artefakte aus demselben oder aus anderen Code-Repositorys identifiziert werden. Die Duplikate repräsentieren 2.677 oder 13% der gefundenen Artefakte. In der überwiegenden Zahl treten dabei nicht mehr als fünf Duplikate für einen originalen Prozessartefakt auf. Jedoch gab es interessanterweise Fälle, in denen bis zu 89 Duplikate gefunden wurden. Berücksichtigt werden muss, dass die ermittelte Zahl an Duplikaten lediglich eine untere Grenze für deren tatsächliche Anzahl auf `GitHub.com` ist, da nur 10% der Code-Repositorys in den Mining-Prozess einbezogen wurden. Die genauere Untersuchung der Ursachen für die hohe Anzahl von Duplikaten ist Gegenstand weiterführender Arbeiten. Eine mögliche Erklärung könnte in manuell abgeleiteten Repositorys oder anderen Arten der Wiederverwendung von Repository-Inhalten liegen [6].

Werden die Dateiformate der Prozessartefakte betrachtet, bestätigt sich die Erwartung, dass es sich bei den meisten von ihnen (16.907 oder 79%) um in XML serialisierte BPMN-Prozesse handelt. Daneben spielen Grafik- und Dokumentenformate eine gewisse Rolle. Wiederum liegt der Anteil an Duplikaten für die identifizierten BPMN-Prozesse bei knapp der Hälfte (8.003 oder 47%). Insgesamt konnte auf diese Weise ein Corpus von 8.904 originalen in XML serialisierten Prozessen der BPMN-Sprachversion 2.0 generiert werden.

3.2 Resultate der Analyse mit *BPMNspector*

Im nachfolgenden Analyseschritt wurden die identifizierten BPMN-Prozesse mit *BPMNspector* analysiert. Ziel der Analyse war dabei, zu untersuchen, inwiefern die durch das Analysewerkzeug *BPMNspector* adressierten Syntax- und Semantikfehler sowie die Prüfung auf diese von praktischer Bedeutung ist.

[11] `http://doubles.sourceforge.net` (21.8.2019)

Prozesse ohne Regelverletzung	1.471
Prozesse mit Regelverletzung	7.376
keine Analyse möglich	57
Prozesse insgesamt	8.904

Regelverletzung	Ext.101	Ext.023	Ext.107	Ext.150	Ext.151
Betroffene Prozesse	5.299 (60%)	5.206 (58%)	5.112 (57%)	1.846 (21%)	1.796 (20%)

Regelverletzung	Ext.023	XsdCheck	Ext.107	Ext.092	Ext.101
Absolute Anzahl	58.015 (39%)	43.516 (29%)	7.398 (5%)	6.788 (5%)	6.699 (5%)

Regelverletzung	Kurzbeschreibung
Ext.023	Fehlender ein-/ausgehender Sequenzfluss für Knoten
Ext.092	Quellendefinition für Datenassoziation nicht eindeutig
Ext.101	Fehlender ausgehender Sequenzfluss für Startereignis
Ext.107	Endereignis definiert keinen eingehenden Sequenzfluss
Ext.150	Fehlender Sequenzfluss für Knoten (ohne Startereignis)
Ext.151	Fehlender Sequenzfluss für Knoten (ohne Endereignis)
XsdCheck	Verletzung des normativen XML-Schemas für BPMN

Tabelle 2. Top-5 der mittels *BPMNspector* identifizierten Regelverletzungen

Als Resultat der Analyse ergab sich nur für eine Minderheit der Prozesse keine Verletzung von Syntax- und Semantikregeln aus dem BPMN-Standard [1,3]. Hingegen lag für 7.376 Prozesse (83%) mindestens eine Regelverletzung vor. Dieses Ergebnis wurde so nicht erwartet und übertrifft die in [4] für eine Fallstudie mit 66 Prozessen berichteten Zahlen. Zwar wurden auch dort für die große Mehrheit der Prozesse Syntax- und Semantikverletzungen festgestellt, jedoch zumindest 24 Prozesse als fehlerfrei identifiziert. Offenbar treten die durch *BPMNspector* erkannten Regelverletzungen, zumindest bei der Verwendung von BPMN-Prozessen auf `GitHub.com`, sehr häufig auf. Wie bereits in Abschnitt 1 angedeutet, übernimmt *BPMNspector* dabei die Rolle eines *Linters* für BPMN. Dies wird für die am häufigsten identifizierten Regelverletzungen deutlich (siehe auch Tabelle 2). So überwiegen, übereinstimmend mit [4], sowohl bei Zählung betroffener Prozesse als auch bei absoluter Zählung, Verletzungen solcher Regeln, die einfache Konsistenzbedingungen für die Definition des Kontrollflusses vorgeben (Ext.023, Ext.101, Ext.107, Ext.150, Ext.151 [3]).

Bei näherer Betrachtung sticht insbesondere die Regelverletzung Ext.023 heraus.[12] Diese beschreibt die inkonsistente Definition des ein- oder ausgehenden Kontrollflusses für einen Prozessknoten. Zur Illustration diene das folgende Prozessbeispiel (Artefakt `"InitiatorTest.testInitiator.bpmn20.xml"`):

[12] vgl. `https://github.com/matthiasgeiger/BPMNspector-fixSeqFlow` (21.8.2019)

```
<process id="InitiatorProcess" isExecutable="true">
  <startEvent id="start" camunda:initiator="initiator" />
  <sequenceFlow id="flow1"
      sourceRef="start" targetRef="theTask" />
  <userTask id="theTask"
      name="my task" camunda:assignee="${initiator}" />
  <sequenceFlow id="flow2"
      sourceRef="theTask" targetRef="theEnd" />
  <endEvent id="theEnd" />
</process>
```

Der Prozess enthält drei Knoten: Start- und Endereignis und Aktivität `theTask`. Die Sequenzflüsse `flow1` und `flow2` sollen den Startknoten mit der Aktivität und die Aktivität mit dem Endknoten verbinden. Der BPMN-Sprachstandard [1] gibt nun vor, dass ein Sequenzfluss, etwas redundant, als `SequenceFlow`-Element zu definieren ist und dessen Quelle und Ziel das Element mittels der Attribute `incoming` und `outgoing` referenzieren sollen. Offensichtlich fehlen die Attribute hier sowohl für die Elemente zur Definition des Start- und Endereignis, als auch für das `userTask`-Element zur Definition der Aktivität, standardtreu müsste für letzteres beispielsweise stehen:

```
...
<userTask id="theTask"
    name="my task" camunda:assignee="${initiator}">
  <incoming>flow1</incoming>
  <outgoing>flow2</outgoing>
</task>
...
```

Das Analysewerkzeug *BPMNspector* identifiziert dementsprechend für diese Prozessdefinition Ext.023 als Regelverletzung. Versuchsweise wurden Regelverletzungen dieser Art für die Prozesse des Corpus behoben und die reparierten Prozesse danach noch einmal mit *BPMNspector* analysiert. Im Ergebnis hat sich die Zahl der Prozesse mit mindestens einer Regelverletzung um knapp die Hälfte auf 4.106 von 8.904 Prozessen (46%) verringert. Wie bereits in [4] angemerkt, scheint dies vor allem auf die mangelnde Analyseunterstützung in den zur Prozessmodellierung eingesetzten Entwicklungsumgebungen hinzudeuten und die Notwendigkeit von Werkzeugen wie *BPMNspector* zu unterstreichen.

4 Zusammenfassung

In diesem Beitrag haben wir exemplarisch für die domänenspezifische Sprache BPMN gezeigt, wie sich durch systematisches Mining auf `GitHub.com` eine umfassende Sammlung von Programmen, beziehungsweise Prozesse im Fall von BPMN, generieren lässt. Ein solcher Corpus kann dann für verschiedene Zwecke eingesetzt werden [10]. Hier haben wir die Validierung der praktischen Relevanz

eines Analysewerkzeugs für BPMN untersucht. Im Einzelnen wurden 10% der Repositorys auf `GitHub.com` nach BPMN-Prozessartefakten durchsucht. Im Ergebnis konnten 1.251 Repositorys mit insgesamt 21.306 Artefakten identifiziert werden. Unter diesen befanden sich 16.907 BPMN-Prozesse und nach Entfernung von Duplikaten ergab sich ein Corpus von 8.904 in XML serialisierten BPMN-Prozessen der Sprachversion 2.0. Die nachfolgende Analyse der Prozesse mittels *BPMNspector* ergab für 7.376 oder 83% der Prozesse mindestens eine Verletzung von Regeln des BPMN-Sprachstandards und unterstützt damit die Aussagen in [4] zur Notwendigkeit von Analysewerkzeugen wie *BPMNspector*.

Einschränkungen der Aussagekraft. Die Anwendungsdomäne von BPMN ist die IT-Unterstützung von Geschäftsprozessen. Diese finden vorwiegend im Wirtschaftssektor Anwendung, so dass Prozesse sehr oft das Produkt sind und daher nur selten veröffentlicht werden [11]. Demnach ist anzunehmen, dass die Verwendung von BPMN in Verbindung mit öffentlichen Code-Repositorys teilweise anderen Zwecken dient. Die Verallgemeinerung der Ergebnisse dieses Beitrags ist damit nur eingeschränkt möglich. Es bleibt weiterführenden Arbeiten überlassen zu untersuchen, inwiefern sich BPMN-Prozesse aus der natürlichen Anwendungsdomäne und aus öffentlichen Code-Repositorys ähneln.

Die Datenquelle für das Mining war auf `GitHub.com` beschränkt und bezog nur 10% der Repositorys mit ein. Zudem war der Ansatz für die Identifizierung von BPMN-Prozessen recht simpel und hat nicht alle Prozessartefakte erfasst. Die Zulässigkeit einer Verallgemeinerung oder Übertragung der Ergebnisse für alle BPMN-Artefakte auf `GitHub.com` oder auch für andere Plattformen muss daher noch näher untersucht werden. In Anbetracht der Anzahl identifizierter Prozesse und der Tatsache, dass `GitHub.com` eine der führenden Plattformen ist, sind wir jedoch davon überzeugt, dass die Ergebnisse ein Indiz auf die Verwendung von BPMN in öffentlichen Code-Repositorys liefern.

Eine stichprobenartige Inspektion der identifizierten BPMN-Artefakte hat eine Vielzahl von Prozessen erkennen lassen, die in Testsammlungen für den automatisierten Test von Ausführungsumgebungen verwendet werden (vergleiche mit Abschnitt 3.1). In Verbindung mit der hohen Anzahl an Duplikaten kann daher eine Verzerrung vorliegen, die eine Verallgemeinerung von Schlussfolgerungen auf Grundlage des generierten Corpus von BPMN-Prozessen einschränkt. Darüber hinaus ist davon auszugehen, dass ein Teil der Artefakte aus Code-Repositorys stammt, die zur Aus- und Weiterbildung eingesetzt werden und nicht der eigentlichen Anwendungsdomäne von BPMN entsprechen [6,10].

Literatur

1. *Business Process Model and Notation (BPMN), Version 2.0.* Object Management Group (OMG) Standard, 2011. – `https://www.omg.org/spec/BPMN/2.0/PDF`
2. Fahland, Dirk ; Favre, Cédric ; Jobstmann, Barbara ; Koehler, Jana ; Lohmann, Niels ; Völzer, Hagen ; Wolf, Karsten: Instantaneous Soundness Checking of Industrial Business Process Models. In: *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings*, Springer, 2009 (Lecture Notes in Computer Science 5701), S. 278–293

3. Geiger, Matthias: *BPMN 2.0 Process Model Serialization Constraints.* Bamberger Beiträge zur Wirtschaftsinformatik und angewandten Informatik, Nr. 92, 2013
4. Geiger, Matthias ; Neugebauer, Philipp ; Vorndran, Andreas: Automatic Standard Compliance Assessment of BPMN 2.0 Process Models. In: *Proceedings of the 9th Central European Workshop on Services and their Composition (ZEUS 2017), Lugano, Switzerland, February 13-14, 2017*, CEUR-WS.org, 2017 (CEUR Workshop Proceedings 1826), S. 4–10
5. Gousios, Georgios: The GHTorent dataset and tool suite. In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, IEEE Computer Society, 2013, S. 233–236
6. Hebig, Regina ; Quang, Truong H. ; Chaudron, Michel ; Robles, Gregorio ; Fernandez, Miguel A.: The Quest for Open Source Projects that use UML: Mining GitHub. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, ACM, 2016, S. 173–183
7. Heinze, Thomas S. ; Amme, Wolfram ; Moser, Simon: Static analysis and process model transformation for an advanced business process to Petri net mapping. In: *Software: Practice and Experience* 48 (2018), Nr. 1, S. 161–195
8. Heinze, Thomas S. ; Türker, Jasmin: Certified Information Flow Analysis of Service Implementations. In: *Proceedings of the 11th IEEE Conference on Service-Oriented Computing and Applications, SOCA 2018, Paris, France, November 20-22, 2018*, IEEE Computer Society, 2018, S. 177–184
9. Prinz, Thomas M. ; Amme, Wolfram: Practical Compiler-Based User Support during the Development of Business Processes. In: *Service-Oriented Computing - ICSOC 2013 Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2-5, 2013. Revised Selected Papers*, Springer, 2013 (Lecture Notes in Computer Science 8377), S. 40–53
10. Robles, Gregor ; Ho-Quang, Truong ; Hebig, Regina ; Chaudron, Michel ; Fernandez, Miguel A.: An extensive dataset of UML models in GitHub. In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, IEEE Computer Society, 2017, S. 519–522
11. Skouradaki, Marigianna ; Roller, Dieter ; Leymann, Frank ; Ferme, Vincenzo ; Pautasso, Cesare: On the Road to Benchmarking BPMN 2.0 Workflow Engines. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*, ACM, 2015, S. 301–304
12. Stefanko, Viktor: *Repository-Mining von BPMN-Prozessen.* unveröffentlichte Bachelorarbeit, Friedrich-Schiller-Universität Jena, 2019

Path-Specific Program Transformation

Jens Knoop

Compilers and Languages
Institute of Information Systems Engineering
Vienna University of Technology
knoop@complang.tuwien.ac.at

Abstract. Sometimes, it is a single program path which needs to be transformed in order to ensure a specific property for the whole program. A prominent example is a program path whose execution time exceeds the timing constraints of a real-time system. If the system is safety-critical, the possible violation of the timing constraints renders the whole program useless. Even if the path is known, traditional approaches to transform and optimize the path to execute fast enough to meet the timing constraints fail because the representation of the path is intertwined with those of many others in the representation of the program, e.g., its flow graph, which prevents the transformation and optimization of the path without affecting the semantics of other program paths.

In this talk, I present a *generic four step approach* allowing for a path-specific program transformation to cope with such situations. (1) Identifying a path violating a user-defined property, using a property-dependent program analysis. (2) [Optionally] Verifying, if possible that the path corresponds to an actual run of the program and is not spurious, using symbolic execution. (3) Isolating and separating the path from other paths in the representation of the program, using property-oriented program expansion. (4) Transforming and optimizing the path to meet the before violated property, using a property-dependent transformation. It is worth noting that this approach specifies a so-called *anytime* algorithm, which can be repeated as often as required or desired.

We illustrate our approach by means of an application for ensuring and verifying the adherence of safety-critical real-time systems to required timing constraints. In particular, we discuss trade-offs of our approach, sketch applications to other fields, and provide an outline of planned future research and work.

Keywords: Anytime Algorithm · Symbolic Execution · Property-oriented Program Expansion · Program Path Transformation and Optimization

Der Weg zum verteilten agentenbasierten Big Data Netzwerk im Logistikumfeld

Marianne Mauch

Lehrstuhl für Softwaretechnik, Institut für Informatik,
Friedrich-Schiller-Universität Jena, Germany
marianne.mauch@uni-jena.de

Abstract. Mobile Agenten bieten innovative Ansätze für komplexe und verteilte Datenstrukturen. Der Lehrstuhl für Softwaretechnik der Friedrich-Schiller-Universität Jena (FSU Jena) forscht seit vielen Jahren an mobilen Agenten und entwickelte in dieser Zeit diverse Multi-Agenten Systeme wie Tracy, Tracy2 und Ellipsis [1, 2]. Die vorliegende Arbeit entstand als Masterarbeit [4] im Rahmen des Projektes SMART DISTRIBUTION LOGISTIK (SDL). Am Beispiel dieses Projektes wurde ein Konzept entworfen, wie das in der Forschung entwickelte MAS Ellipsis in historisch gewachsene Bestandssysteme von Logistikunternehmen und in Bestandssysteme von Anbieter von Telematiklösungen integriert werden kann.

Keywords: mobile Agenten · Logistik · verteiltes Multi-Agenten System.

Die Branche der Medienlogistik befindet sich seit einigen Jahren im Umbruch und strukturiert im Zuge dessen auch ihre Geschäftsprozesse grundlegend um. Das eröffnet die Chance, verschiedene Elektro-Transportfahrzeuge mit Hilfe intelligenter Planungstools von Anfang an kostenoptimiert in Logistikprozesse zu integrieren. Das Projekt SMART DISTRIBUTION LOGISTIK (SDL) [8] gefördert durch das Technologieprogramm IKT für Elektromobilität III (IKT EM III) des Bundesministeriums für Wirtschaft und Energie entwickelt eine Informations- und Kommunikationstechnologie (IKT) - Systemplattform für die Medien- und Pharmalogistik. Eine mehrschichtige Planungsunterstützung für Standort-, Rahmen- und Tourenplanung und -steuerung und eine Optimierung des logistischen Systems durch Total Cost of Ownership (TCO) - basierte Faktorbewertung sind wesentliche Bestandteile dieser Plattform sowie die Umsetzung von Multi-Use-Konzepten wie das unternehmensübergreifende Sharing von Fracht und Fahrzeugen zur Verbesserung der Wirtschaftlichkeit von Elektrofahrzeugen.

Zum Verbundprojekt gehören zwei Hochschulen, zwei Medienlogistikunternehmen, ein Pharmalogistikunternehmen, ein Elektronikgerätehersteller sowie ein Anbieter von Telematiklösungen für Logistik und Transport (kurz: ATL). Das Teilprojekt der FSU Jena [9, 3] erstellt ein verteiltes agentenbasiertes Big Data Netzwerk für sichere E-Mobility-TCO-Analysen von autonomen Systemen

im Logistikumfeld (SDL EMAAS). Im Vordergrund steht eine offene, verzahnte und selbstkalibrierende Systemarchitektur mit mobilen Agenten in verteilten Systemen [7]. Die drei kooperierenden Logistikunternehmen im Projekt sowie der ATL haben auf verschiedenen Systemen viele sehr unterschiedliche Daten. Eine besondere Rolle in diesem sehr stark verteilten System mit vielen heterogenen Daten kommt den Agenten, der Migration der Agenten und deren Verhandlungen zu. Daten können durch deren Einsatz vor Ort gefiltert und analysiert werden. Unter Beachtung der Datensicherheit und Zuverlässigkeit migrieren Agenten von einem System zu einem anderen, realisieren Verhandlungen, filtern, analysieren Daten, führen weitere Aktionen aus, agieren und reisen weiter. Dies erfordert den Entwurf passender Verhandlungs- und Migrationsstrategien, welche zunächst nur statisch ausgelegt werden. Auf diese Art und Weise wird ein verteiltes Big-Data-Netzwerk zur unabhängigen Verwaltung von branchenspezifischen Daten aufgebaut.

Anbieter von Branchensoftware-Lösungen im Logistikumfeld haben fest definierte Strukturen in Form von Hard- und Software für tausende von Kunden. Aus Wartungs-, Stabilitäts- und Sicherheitsgründen werden nur Softwaresysteme betrieben, die diesen Vorgaben genügen. Innerhalb dieses Rahmens ist diese gelebte Software performant, stabil und somit relevant. Würde man ein völlig neues Konzept für mobile Agenten in diese Softwarelösung unterbringen wollen, müsste man diese vollständig umbauen und ein völlig neues System einschließlich eines Stagingkonzeptes für mehrere tausend Kunden parallel entwickeln. Diese Vorgehensweise ist unrealistisch. Aus diesem Grund muss ein intelligentes Architekturkonzept für mobile Agenten entwickelt werden, welches das Bestandssystem des ATL entsprechend einbindet.

Die Skepsis gegenüber dem unbekannten Agentenkonzept war groß. Auch die verbreitete Programmiersprache Java (für das MAS Ellipsis), stellte eine große Herausforderung dar. Die Telematiklösung des ATL basiert auf C# auf der Basis des .Net Frameworks. Die Prozesse und Schnittstellen der Logistikunternehmen sind inhomogen und oft noch papierbasiert. Es war unklar, ob aufgrund von Konzern- und Datenschutzrichtlinien fremde Hard- und Software durch die Partner akzeptiert werden würde.

Lokale Daten sollten lokal vorverarbeitet und analysiert werden, damit nach dem Prinzip der Datenminimalität nur Daten übertragen werden, die auf dem anderen Netzwerkknoten benötigt werden. Für ein verteiltes agentenbasiertes Netzwerk müssen die Agenten die Möglichkeit haben, zwischen den MAS über Rechnergrenzen hinweg zu migrieren. Damit werden Agenten in die Lage versetzt, Verhandlungen auf fremden Systemen im Sinne eines Forth-Party-Logistics-Provider (4PL) durchführen zu können. Diese Eigenschaft würde möglicherweise nicht in jedes Sicherheitskonzept einer Firma passen. Deshalb wurden beide Eigenschaften getrennt voneinander betrachtet.

Die Knoten dieses Netzwerkes wären der ATL und die drei Logistikpartner im Projekt und wären mögliche Orte für ein MAS mit unterschiedlichen Eigenschaften. Die Kanten wären die Kommunikationsschnittstellen. Es mussten verschiedene mögliche Szenarien zur Verteilung dieser MAS mit sehr unter-

schiedlichen Eigenschaften erarbeitet werden. Die erste Variante spiegelt die Vision eines verteilten Big-Data-Netzwerkes wieder. Jeder Netzwerkknoten wäre mit einem MAS ausgestattet, welcher autark verhandeln könnte. Das komplette Gegenteil dazu ist die Aktienbörse. Alle zur Verhandlung benötigten Daten sind auf einem System, das auch die Verhandlungen führt. Den Verhandlungspartnern steht nur ein Webinterface zur Verfügung. Die mittlere Variante wäre dann ein Marktplatz. Auf diesem wird auch gehandelt. Die Verhandlungspartner haben die Möglichkeit, mittels mobiler Agenten ihre Waren auf den Marktplatz zu bringen. Aufgrund vorbearbeiteter Daten würden nur die Daten übertragen werden, die für die Verhandlung benötigt werden. Diese drei grundlegenden Szenarien wurden weiter ausgearbeitet und bildeten eine Diskussionsgrundlage für kontinuierliche Gespräche mit den Entwicklungspartnern im Projekt über ein zu modellierendes verteiltes Netzwerk auf der Basis mobiler Agenten.

Zur Visualisierung der Verteilung und der Kommunikation der verschiedenen Organisationseinheiten innerhalb dieses zu modellierenden verteilten Netzwerkes wurden im Projekt vier Szenarien für den Einsatz eines MAS erarbeitet. Dafür wurde eine eigene Form der Darstellung gewählt, die keinen der in der Informatik üblichen Notationen entspricht. Nicht die Dokumentation und die Modellierung stand im Vordergrund, sondern das Verständnis der Projektpartner für den Ansatz der Agententechnologie. Aus Sicht der UML entspricht diese Darstellung am ehesten einem verallgemeinerten Verteilungsdiagramm, das lediglich aus Knoten und Kommunikationspfaden bzw. Kanten besteht. In den Szenarien werden alle Softwaresysteme innerhalb einer Organisationseinheit eines Projektpartners als ein Knoten dargestellt. Sind auf diesem Knoten als Komponente ein MAS installiert, werden in dieser Darstellungsform weitere Symbole in räumlicher Nähe der Knoten angezeigt. Für die Orte der Datenhaltung bzw. -verarbeitung und die Orte, von denen aus Verhandlungen mittels mobiler Agenten stattfinden können, wurden zwei eigene Symbole entworfen. Ist eines dieser Symbole an den Knoten zu finden, existiert auch ein MAS als Komponente auf diesem Knoten entweder mit einer Datenhaltungsinfrastruktur oder mit einer Kombination aus beiden genannten Infrastrukturen für Datenhaltung und Verhandlungen. Die Knoten werden durch Kanten in Form von Schnittstellen miteinander verbunden. Die Migrationsschnittstelle ist die Verbindung zwischen zwei Knoten und beschreibt die Schnittstelle, über die Agenten migrieren können. Die Datenschnittstelle ist die Verbindung zwischen zwei Knoten und beschreibt eine einfache Schnittstelle zur Übermittlung von Daten.

Ein mehrstufiger Entwicklungsprozess wurde gewählt, der einem angepassten Spiralmodell ähnelt. Nach ausführlichen Gesprächen und Analysen wurden im ersten Schritt wie im Szenario ,,Börsenmakler" eine Stand-alone-Lösung mittels des MAS Ellipsis entwickelt, die sich alle benötigten Daten von den Systemen der Logistikpartner holt, auf das Format der REST-Schnittstelle überträgt und die Übertragung startet. Im zweiten Schritt wurde ein verteiltes Netzwerk entworfen. Damit wurde aus dem Szenario ,,Börsenmakler" das Szenario ,,Marktplatz". Da zu diesem Zeitpunkt noch nicht klar war, welcher Logistikpartner Soft- und Hardware für ein MAS aufnimmt, konnte es auch eine Mischung aus beiden

werden. Basierend auf den Ergebnissen weiterführender Gespräche wird nun ein verteiltes agentenbasiertes Big Data Netzwerk entwickelt. Damit kann gezeigt werden, dass mit Hilfe mobiler Agenten zwischen den Logistikunternehmen im Sinne eines Logistikkonzeptes über mögliche Car- oder Cargo-Sharing-Events autonom verhandelt werden kann. Dadurch kann aus dem Szenario ,,Marktplatz" das Szenario ,,Verteiltes Big-Data-Netzwerk" Schritt für Schritt entwickelt werden.

Die im Projektverlauf zusammengefassten Teilergebnisse wurden in Bezug zu den vier Szenarien und dem im Projekt zu entwickelnden verteilten agentenbasierten Netzwerkes gebracht. Dabei wurde eine für dieses Projekt angepasste Herangehensweise für die Integration des MAS Ellipsis in Bestandssysteme von Unternehmen in Form eines verteilten Big-Data- Netzwerkes entwickelt. Ausgehend von einem einfachen aber konkreten Anwendungsfall im Projekt wird mit Hilfe des Spiralmodells Schritt für Schritt von der Stand-alone Variante der vier Szenarien vom ,,Börsenmakler" über den ,,Marktplatz" mit einem ersten verteilten Ansatz zum ,,Verteilten Big-Data-Netzwerk" vorgegangen. Sollte im Laufe des Projektes ein externer Anbieter dieses Konzept umsetzen wollen, würde das 4. Szenario greifen, der ,,Externe 4PL".

References

1. Braun, P., Rossak, W.: Mobile Agents. Morgan Kaufmann, San Francisco, USA (2005)
2. Schau, V.: Ellipsis-Konzeption und Implementierung einer vereinheitlichten und leistungsfähigen Architektur für Multi-Agenten Systeme (MAS) auf Basis frei verfügbarer IT-Infrastrukturkomponeneten unter besonderer Berücksichtigung internationaler Standards. PhD thesis, Friedrich-Schiller-Universität Jena, Deutschland (2012)
3. Oppolzer, T., Kretzschmar, J., Mauch, M., Schau, V., Rossak, W.: Agentenbasiertes Mischflottenmanagement in der innerstädtischen Logistik. In: Tagungsband des 14. Fachgesprächs ,,Ortsbezogene Anwendungen und Dienste" der GI-Fachgruppe KuVS Berlin, Deutschland (2017)
4. Mauch, M.: Einführung mobiler Agenten in das Projekt SDL. Masters thesis, Friedrich-Schiller-Universität Jena, Deutschland (2018)
5. Kretzschmar, J., Johlke, M., Rossak, W.: A TCO Analysis Tool based on Constraint Systems for City Logistics. In: VEHICULAR 2018, The Seventh International Conference on Advances in Vehicular Systems, Technologies and Applications (IARIA), Venedig, Italien (2018)
6. Kretzschmar, J., Eckardt, F.: A Blockchain Approach towards Cargo Sharing in Last Mile Logistics. In: VEHICULAR 2019, The Eighth International Conference on Advances in Vehicular Systems, Technologies and Applications (IARIA), Rom, Italien (2019)
7. Schäfer, A., Schau, V., Mauch, M.: Two layer approach: A modified Multi Agent System (MAS) as a network-independent adaptive framework for a distributed logistic network. In: 13th International KES Conference: Agents and multi-agent systems: Technologies and Applications, Malta (2019)
8. SDL Homepage, http://www.sdl-projekt.de/. Letzter Zugriff: 21 Aug 2019
9. SDL EMAAS Homepage, http://sdl.uni-jena.de/. Letzter Zugriff: 21 Aug 2019

Implementing a Mutation Testing Framework in Rust using Procedural Macros

Samuel Pilz

Compilers and Languages
Institute of Information Systems Engineering
Vienna University of Technology
e1327391@student.tuwien.ac.at

Abstract. Mutation testing is a powerful approach to evaluate the strength of a test suite. In this approach, the program under test is seeded with artificial faults, which should be disclosed by a good test suite. The implementation of a mutation testing framework requires typically language-specific trade-offs regarding the usually conflicting goals of performance, flexibility, and complexity. We argue that the unique set of features of Rust, in particular, its procedural macros, enables the development of a mutation testing framework reconciling these goals in an unprecedented manner.

In order to provide evidence for this claim, we implemented the mutation testing framework 'mutagen' and contributed it to the open source ecosystem of Rust. The key of 'mutagen' is to exploit the procedural macros of Rust, which allow for arbitrary transformations of the source code at compile-time. In our approach, we use procedural macros to bake all mutations into the compiled code. Therefore, we succeed without runtime instrumentation, which is a major source of poor performance in other mutation testing frameworks. In our approach, code and test suite have to be built only once, which yields a major performance improvement. Moreover, we can exploit the full power of Rust to implement the transformations for baking the mutations into the source code, since procedural macros are an integral language part of Rust. This increases flexibility and decreases complexity in addition to enabling our performance gains. Moreover, it makes our framework easily extensible.

In this talk we present our approach, highlight especially our usage of procedural macros, report on first experiences with 'mutagen,' and demonstrate its usage. Applying 'mutagen' to a set of small libraries has shown to be straightforward and effective to reveal weaknesses in test suites. As future work, we are planning to apply 'mutagen' to further libraries central to the ecosystem of Rust, including also well-known large ones.

Keywords: Rust · Mutation Testing · Procedural Macros

Von Pizza zu Java–TX

Martin Plümicke

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

Zusammenfassung Seit den 1990iger Jahren werden Eigenschaften, die aus funktionalen Programmiersprachen bekannt sind, in objektorientierte Sprachen übernommen. In diesem Artikel wird die Entwicklung der Programmiersprache Java aufgezeigt. Eine erste Erweiterung von Java, die Eigenschaften funktionaler Sprachen übernommen hat, war PIZZA. PIZZA hatte *parametrische Polymorphie*, *Funktionen höherer Ordnung* und *algebraische Datentypen*. In Java 5 wurde die *parametrische Polymorphie*, erweitert durch use-site Varianz, übernommen. In Java 8 kamen schließlich Lambda-Ausdrücke hinzu. Die Eigenschaft *Typinferenz*, die funktionale Sprachen sehr flexibel macht, ist in Java nur sehr eingeschränkt verfügbar. Die Erweiterung Java-TX verfügt über einen globalen Typinferenzalgorithmus, der es ermöglicht, Java Code ohne jede Typangabe, bei gleichzeitiger voller statischer Typsicherheit, zu schreiben.

1 Einleitung und Motivation

In die ersten Versionen von Java waren weder parametrische Polymorphie noch Funktionen höherer Ordnung verfügbar. Das führte zu oftmals sehr langem und übersichtlichem und damit fehleranfälligen Code. Die fehlende parametrische Polymorphie führte dazu, dass man bei jeder Entnahme aus einer Collection eine Typecast brauchte, um das jeweilige Element wieder in seinem Ausgangstype verwenden zu können:

```
Collection v = new Vector();
v.addElement(new Integer(1));
Integer i = (Integer)v.elementAt(0);
```

Dieses Problem lässt sich durch parametrische Polymorphie lösen, was seit Version 5 in Java möglich ist.
Wenn man vor Version 8 eine Callback–Funktion implementieren wollte, so musste zunächst eine Klasse implementiert werden, die die als Interface mit einer Methode bereitgestellte Callback–Funktion implementierte. Schließlich musste die Funktion mit einer Instanz dieser Klasse aufgerufen werden. Es gab immerhin das Konzept der anonymen inneren Klassen, das gleichzeitig die Implementierung und Instanziierung eines Interfaces ermöglichte. In folgendem Beispiel soll einem `AWT`–Button einem `ActionListener` hinzugefügt werden, der die Anzahl der Klicks zählt und diese ausgibt.

```
button.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        numClicks++;
        System.out.println("Button Clicked " +
           numClicks + " times");
    }
}
```

Dieses lässt sich erheblich einfacher durch Lambda-Ausdrücke lösen, die seit Version 8 in Java enthalten sind.
Das Konzept der parametrischen Polymorphie, ergänzt um Wildcards zur Realisierung der use-site Varianz, führte zu sehr langen Typausdrücken, die ein Benutzer in der Regel nicht schreiben möchte. So hätte z.B. die Matrix–Multiplikation definiert als Lambda-Ausdruck folgenden Typ:

```
mul: BiFunction<? super Vector<? extends Vector<? extends Integer>>,
               ? super Vector<? extends Vector<? extends Integer>>,
               ? extends MatrixOP>
```

In Java-TX wird dieses Problem durch globale Typinferenz gelöst, so dass Typen automatisch berechnet werden können.
Der Artikel ist wie folgt strukturiert. Im folgenden zweiten Abschnitt betrachten wir PIZZA mit den Eigenschaften *parametrische Polymorphie*, *Funktionen höherer Ordnung* und *algebraische Datentypen*. Im dritten Abschnitt betrachten die Erweiterungen im Java–Standard *parametrische Polymorphie mit Wildcards*, *Lambda-Ausdrücke* und *lokale Typinferenz*. Im vierten Abschnitt beschreiben wir die neuen Eigenschaften von Java-TX *echte Funktionstypen* und *globale Typinferenz* und geben einen Ausblick auf *Pattern–Matching*. Der Artikel schließt mit einer Zusammenfassung.

2 PIZZA

PIZZA [OW97] ist eine echte Obermenge des damaligen Java–Standards, welcher Java um die drei Eigenschaften *parametrische Polymorphie*, *Funktionen höherer Ordnung* und *algebraische Datentypen* erweiterte, die vorher nur in eher akademischen funktionalen Programmiersprachen enthalten waren.

2.1 Parametrische Polymorphie

Ein einfaches Beispiel für parametrische Polymorphie ist die Klasse `Pair`

```
class Pair<T> {
    T x; T y;
    Pair (T x, T y){ this.x = x; this.y = y; }
    void swap () { elem t = x; x = y; y = t; }
}
```

```
Pair<String> p = new Pair("world!", "Hello,");
p.swap();
  Pair<int> q = new Pair(22, 64);
q.swap();
```

Im Gegensatz zu späteren Java–Versionen werden beim `new`–Statement die Parameter weggelassen (`new Pair(...)`) und es ist möglich Basistypen als Parameter einzusetzen (`Pair<int>`). Um PIZZA typsicher zu halten, sind Typparameter invariant. Das bedeutet, dass für `Vector<T>` $\leq^*$ [1] `Collection<T>` zwar `Vector<int>` $\leq^*$ `Collection<int>` gilt, aber für `String` $\leq^*$ `Object` nicht gilt `Vector<String>` $\leq^*$ `Vector<Object>`. Dies ist ein starke Einschränkung, die in Java 5 durch Wildcards aufgelöst wurde.

2.2 Funktionen höherer Ordnung

PIZZA erlaubt sowohl echte Funktionstypen als auch Lambda-Ausdrücke. Im Allgemeinen wird ein Funktionstyp durch $(t_1, \ldots, t_n) \to t_0$ und ein Lambda-Ausdruck durch `fun` $(t_1\ x_1, \ldots, t_n\ x_n) \to t_0\ stmt$ dargestellt. Das folgende Beispiel zeigt beides:

```
class Radix{
      int n = 0;
      (char)->boolean radix(int r){
          return fun (char c)->boolean{
              n++;
              return '0' <= c && c<'0'+r;};
      }

      String test (){
          (char)->boolean f = radix(8);
          return f('0')+" "+f('9')+" "+n;
      }
}
```

In PIZZA gibt es kein Subtyping von Funktionstypen.

2.3 Algebraische Datentypen und Pattern-Matching

In PIZZA ist es möglich algebraische Datentypen zu definieren. Das folgenden Beispiel zeigt in den ersten drei Zeilen der Klasse die Deklaration des Datentyps `List`:

```
class List {
    case Nil;
    case Cons(char head, List tail);
```

[1] $\leq^*$ steht für das Subtyping.

```
    List append (List ys) {
        switch (this){
            case Nil:
                return ys;
            case Cons(char x, List xs):
                return Cons(x, xs.append(ys));}
    }
}

List zs =
   Cons('a',Cons('b',Nil)).append(Cons('c',Nil));
```

In der Methode append wird dann Pattern-Matching demonstriert. In der letzten Zeile wird schließlich der Aufruf der jeweiligen Daten–Konstruktoren gezeigt.

3 Java 5 bis 10

Mit der Version 5 von Java [GJSB05] im Jahre 2004 wurde begonnen Eigenschaften aus funktionalen Sprachen in Java zu übernehmen. In Java 5 war es die *parametrische Polymorphie*. Im Gegensatz zu PIZZA sind die Typparameter in Java nicht invariant. In Java 8 kamen die Lambda-Ausdrücke hinzu. Dabei wurde aber als Typkonzept nicht auf die Funktionstypen, sondern auf sogenannte functional interfaces gesetzt. In Java 10 wurde schließlich die lokale Typinferenz durch die Einführung des Schlüsselwortes `var` als Typplatzhalter für lokale Variablen vollendet.

3.1 Parametrische Polymorphie (Generics)

Im Gegensatz zu PIZZA erlaubt Java die sogenannte use-site Varianz für Typparameter. D.h. bei der Benutzung eines Datentyps wird festgelegt, ob die Typparameter kovariant oder kontravariant sind.
Betrachten wir zunächst Beispiele, die das grundsätzliche Problem zeigen, warum ein Typsystem durch Varianzen unsicher werden kann.

Kovarianz

```
Vector<Integer> v_int = new Vector<Integer>();
Vector<Object> v_obj = v_int//Kovarianz
                            //ist illegal;
v_obj.addElement(new Object());
```

In diesem Fall würde ein `Object` zu einem Vector von `Integer`s hinzugefügt.

Kontravarianz Man könnte auf die Idee kommen Kontravarianz zuzulassen.

```
Vector<Object> v_obj = new Vector<Object>();
v_obj.addElement(new Object());
Vector<Integer> v_int = v_obj;//Kontravarianz
                              //ist illegal
v_int.addElement(1);
Integer i = v_obj.elementAt(0);
```

Dabei ist es unproblematisch einen `Integer`–Wert 1 implizit über `v_int` zu dem Objekt `v_obj` hinzuzufügen. Allerdings würde dann der Variablen `i` vom Typ `Integer` ein Element vom Typ `Object` zugeordnet, was nicht korrekt wäre.

Wildcards Dennoch kann eine Zuweisung wie `Vector<Object> v_obj = v_int` im ersten oder `Vector<Integer> v_int = v_obj` im zweiten Beispiel erwünscht und sinnvoll sein.
Dazu wurden in Java 5 [GJSB05] Wildcards eingeführt, die eine eingeschränkte Form von existentiellen Typen darstellen. Mit Wildcards kann use-site Ko- und Kontravarianz festgelegt werden.

Passen wir zunächst das erste Beispiel an, das Kovarianz zeigt:

```
Vector<Integer> v_int = new Vector<Integer>();
Vector<? extends Object> v_obj = v_int// korrekt
v_obj .addElement(new Object());//illegal
Object o = v_obj.elementAt(0);
```

Durch den Wildcard `? extends Object` wird die Zuweisung möglich. Nun ist aber das Hinzufügen eines Elements unmöglich, da über den Typ der Elemente des Vectors nur bekannt ist, dass er von `Object` erbt. Es ist aber möglich eine Element vom Typ `Object` zu entnehmen.

Analog passen wir das zweite Beispiel an, das Kontravarianz zeigt:

```
Vector<Object> v_obj = new Vector<Object>();
v_obj.addElement(new Object());
Vector<? super Integer> v_int = v_obj;
v_jnt.addElement(1);
Integer i = v_int.elementAt(0);  //illegal
```

Hier wird für den Typ der Elemente des Vectors `v_int` festgelegt, dass es ein Supertyp von `Integer` ist. Dementsprechend kann zwar ein `Integer`–Element hinzugefügt werden, aber über den Typ eines entnommenen Elements kann nur ausgesagt werden, dass es ein Supertyp von `Integer` ist, Somit kann er nicht einer `Integer`–Variablen zugewiesen werden.

3.2 Lambda-Ausdrücke

In Java 8 [GJS+15] wurden Lambda-Ausdrücke zur Sprache hinzugefügt. Dabei wurde nach längerer Diskussion auf echte Funktionstypen, wie wir sie aus PIZZA kennen (vgl. Abschnitt 2.2), verzichtet. Stattdessen wurden *functional interfaces*

als *target types* von Lambda-Ausdrücke eingeführt. Functional interfaces sind im Wesentlichen Interfaces mit einer Methode, die den Typ des Lambda-Ausdrucks repräsentiert. Des weiteren wurde der Begriff des *compatible target types* eingeführt, der die Typkorrektheit beschreibt.
Folgendes Beispiel zeigt den Ansatz.

```
interface Id<A>  {
    A id (A x);
}

Id<Integer> id = x -> x;
```

Dieser Ansatz hat den großen Vorteil, dass sehr viele *Callback–Funktionen*, die in den Java-Bibliotheken bereits als function interfaces existierten, mit Lambda-Ausdrücken implementiert werden können.
Betrachten wir nochmals das Beispiel aus der Einleitung (Abschnitt 1):

```
button.addActionListener(
   e -> {
        numClicks++;
        System.out.println("Button Clicked " +
           numClicks + " times");
    }
}
```

Man sieht, dass nur noch die Funktion selbst und keine Klasse und keine Instanz der Klasse mehr definiert werden muss.
Gleichzeitig hat der Ansatz aber den Nachteil, dass echte Funktionstypen nur simuliert werden können, das Subtyping auf Funktionstypen und die direkte Anwendung von Lambda-Ausdrücken auf Argumente nur sehr unschön realisierbar sind. Für nähere Ausführungen verweisen wir auch [PS17].

3.3 Lokale Typinferenz

Lokal Typinferenz wurde theoretisch in [PT00,OZZ01] beschrieben. In Java wurden Elemente der lokalen Typinferenz sukzessive eingeführt. In Java 5 wurde die automatische Typbestimmung für Instanzen von Generics realisiert. Für

```
Vector<Integer> v = ...
v.elementAt(0);
```

wird der Typ von `v.elementAt(0)` automatisch als `Integer` bestimmt.
In Java 7 wurde der sogenannte Diamond–Operator eingeführt. Das heißt, dass beim `new`–Operator keine neuerliche Angabe der Parameter mehr nötig ist.

```
Vector<Integer> v = new Vector<>();
```

Dies ist vergleichbar mit der Syntax in PIZZA (vgl. Abschnitt 2.1), wobei im Gegensatz zu PIZZA hier die Klammern <> notwendig sind, um die parametrisierten Typen von sogenannten *raw–types* unterschieden zu können, die aus Kompatibilitätsgründen auch bei höheren Java–Version noch enthalten sind.

In Java 8 können bei vergebenen target types die Typen der Lambda-Ausdrücke automatisch berechnet werden. Dabei wird der Returntyp im Gegensatz zu PIZZA (vgl. Abschnitt 2.2) immer automatisch berechnet. Die Argumente können mit und ohne Typangabe geschrieben werden: $(t_1\ x_1, \ldots, t_n\ x_n) \rightarrow stmt \mid expr$ bzw. $(x_1, \ldots, x_n) \rightarrow stmt \mid expr$
In Java 10 wurde schließlich noch das `var`–Schlüsselwort als Typplatzhalter für lokale Variablen eingeführt. Für `var s = "hallo"` wird für `s` dar Typ `String` inferiert.
Mit Version 10 hat Java aus den funktionalen Sprachen die *parametrische Polymorphie* erweitert um Wildcards, die *Lambda-Ausdrücke* und Anteile der Typinferenz (*lokale Typinferenz*) übernommen. In Java gibt es bisher keine *globale Typinferenz* (auslassen aller Typinformationen und automatische Berechnung zur Übersetzungszeit), keine *echten Funktionstypen* und keine *algebraischen Datentypen.*

4 Java-TX

Java-TX (TX steht für *Type eXtended*) wendet sich diesen drei in Java noch fehlenden Eigenschaften zu. Das Herzstück von Java-TX ist die Typinferenz. Zunächst betrachten wir aber die Einführung von echten Funktionstypen, weil wir diese für die Typinferenz mit Lambda-Ausdrücken notwendig sind.

Integration von echte Funktionstypen und functional interfaces In Abschnitt 3.2 ist die Realisierung von Lambda-Ausdrücken mit functional interfaces als target types beschrieben. Die Nachteile dieses Ansatzes haben wir in [PS17] dargestellt. Weiterhin haben wir dort beschrieben wie wir Funktionstypen analog zu Scala [OAC+19] in Java-TX eingeführt haben.
Der Ansatz in Java-TX ist echte Funktionstypen einzuführen, gleichzeitig aber das target typing mit functional interfaces zu erhalten und beide Ansätze zu integrieren.
Dazu haben wir in Java-TX zwei Bündel von speziellen functional interfaces ergänzt:

```
FunN$$<-A1, ... , -AN, +R> {
    R apply(A1 arg1, ... AN argN);
}

FunVoidN$$<-A1, ... , -AN> {
    R apply(A1 arg1, ... AN argN);
}
```

Die Interfaces haben im Gegensatz zu sonstigen Typen in Java declaration-site Varianz. Das heißt, dass bereits bei der Deklaration festgelegt wird, welche Parameter kovariant und welche kontravariant sind. - steht dabei für Kontravarianz und + für Kovarianz. Weiterhin sind für die Interfaces Wildcards als Parameter

nicht zulässig. Lambda-Ausdrücke werden durch diese Interfaces explizit typisiert.
Als Beispiel betrachten wir aus JavaFX die Methode `setOnAction` der Klasse `javafx.scene.control.Button`. Sie erwartet als Argument einen `javafx.event.EventHandler`. Das Interface `javafx.event.EventHandler` ist ein functional interface:

```
interface EventHandler<T extends javafx.event.Event>
{
    void handle(T event);
}.
```

Wir definieren zunächst einen explizit getypten Lambda-Ausdruck:

```
Fun1Void$$<ActionEvent> helloworld =
    event -> System.out.println("Hello World!");
```

Diesen geben wir nun als Argument der Methode `setOnAction` mit:

```
Button btn = new Button ();
btn.setOnAction(helloworld);
```

Das Beispiel zeigt die Integration der beiden Konzepte *Typisierung von Lambda-Ausdrücken durch echte Funktionstypen* und *target typing von Lambda-Ausdrücken.* Dem Argument der Methode `setOnAction` mit target type `EventHandler` wird die explizit getypte Variable `helloworld`, belegt mit einem Lambda-Ausdruck, zugeordnet.

4.1 Globale Typinferenz

Globale Typinferenz meint im Gegensatz zu lokaler Typinferenz, dass alle Typen eines Java Programms weggelassen werden können und diese zur Übersetzungszeit inferiert werden. Das heißt, dass Java die statische Typeigenschaft behält. Dies gilt insbesondere auch für rekursive Lambda-Ausdrücke. Für den Typinferenzalgorithmus verweisen wir auf [Plü15]. Drei Beispiele, die die Mächtigkeit der Typinferenz zeigen geben wir hier an.
Das erste Beispiel ist die Fakultät, definiert als rekursiver Lambda-Ausdruck.

```
class Faculty {
    fact[2] = (x) -> {
            if (x == 1) {
                return 1;
            }
            else {
              return x * (fact.apply(x-1));
            }
```

[2] In Java-TX kann im Gegensatz zum Java–Standard einem Attribut direkt ein Lambda-Ausdruck zugewiesen werden. In Standard Java ist dies nur über einen Konstruktor möglich.

```
        };
    }
```

Der Typinferenzalgorithmus berechnet für das Attribut `fact` den echten Funktionstyp `Fun1$$<Integer, Integer>` .
Das zweite Beispiel zeigt die zusätzliche Möglichkeiten, die die Typinferenz für das Overloading bietet.

```
class OL     {
   m(x) { return x + x; }

   m(x) { return x || x; }
}

class OLMain   {
      main(x) {
               var ol;
               ol = new OL();
               return ol.m(x);
      }
 }

 System.out.println(ol.main(2));
 System.out.println(ol.main(2.0));
 System.out.println(ol.main("applied to a string "));
 System.out.println(ol.main(true));
```

In der Klasse `OL` wird eine überladene Methode `m` für die Addition und eine Methode `m` für das boole'sche *oder* deklariert. In der Funktion `main` der Klasse `OLMain` wird `m` der Klasse `OL` aufgerufen. Dies ergibt nun je nach Argument entweder die eine oder die andere Methode. Dementsprechend ist `main` nun für Werte der Typen `Integer`, `Double`, `String` und `Boolean` aufrufbar.
Das dritte Beispiel (vgl. zugehöriges Beispiel im Abschnitt 1) zeigt, dass der Typinferenzalgorithmus auch Typen mit Wildcards bestimmt.
In der Klasse `MatrixOP` (Abb. 1), die eine Erweiterung von `Vector<Vector<Integer>>` ist, wird ein Attribut `mul` deklariert. `mul` wird ein Lambda-Ausdruck zugeordnet, der die Matrix–Multiplikation berechnet. Als Typ für `mul` wird berechnet:

```
mul: Fun2$$<Vector<? extends Vector<? extends Integer>>,
            Vector<? extends Vector<? extends Integer>>,
            MatrixOP>
```

4.2 Pattern–Matching

Pattern–Matching ist derzeit in Java-TX noch nicht implementiert. Hier werden erste Überlegungen zur Realisierung vorgestellt.
In PIZZA wird Pattern–Matching mit Hilfe von algebraischen Datentypen realisiert (vgl. Abschnitt 2.3). Die algebraischen Datentypen werden dann in Java

```
public class MatrixOP extends Vector<Vector<Integer>> {
  mul = (m1, m2) -> {
    var ret = new MatrixOP();
    var i = 0;
    while(i < m1.size()) {
      var v1 = m1.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) *
              m2.elementAt(k).elementAt(j);
          k++; }
        v2.addElement(erg);
        j++; }
      ret.addElement(v2);
      i++;}
    return ret;}
```

Abbildung 1. Matrix–Multiplikation

Klassen übersetzt. Wir wollen in Java-TX auf explizite Deklaration algebraischer Datentypen verzichten und mit Hilfe der Konstruktoren und Attributen der Java Klassen Pattern–Matching ermöglichen. Des weiteren wollen wir Pattern–Matching nicht nur über die `switch`–`case`–Anweisung, sondern ähnlich wie in funktionalen Sprachen direkt im Methodenkopf ermöglichen.
Betrachten wir als Beispiel eine erweiterte Liste mit geordneten Elementen:

```
class List<T extends Comparator<T>> {
    T head;
    List<T> tail;

    List(T head, List<T> tail) {
        this.head = head;
        this.tail = tail;
    }
}
```

Als Methode, die Pattern–Matching benutzt, betrachten wir `merge`. Ähnlich wie in Haskell soll Pattern–Matching über Attributzugriffe (Record Syntax) und *as–Pattern (@)* ermöglicht werden. Durch die in Java-TX enthaltende Typinferenz kann Pattern–Matching genauso wie in funktionalen Sprachen ohne Typangabe in den Argumenten der Methoden benutzt werden.

```
  merge (x, null) {
```

```
    return x;
  }
  merge(null, y) {
    return y;
  }
  merge (l1@List{head = hd1, tail = tl1},
         l2@List{head = hd2, tail = tl2}) {
      if (hd1.compare(hd2) <= 0) {
          return new List(hd1, merge(tl1, l2));
      }
      else if  (head.compare(hd) > 0) {
         return new List(hd2, merge(l1, tl2);
      }
  }
```

Ergänzend müssen noch Überlegungen zu Vererbung und partiell definierten Funktionen angestellt werden.

Zusammenfassung

In diesem Artikel haben wir die Integration von Eigenschaften, die ursprünglich in funktionalen Programmiersprachen enthalten waren, in die objektorientierte Programmiersprache Java aufgezeigt. Zunächst präsentierten wir die Sprache PIZZA, eine echte Obermenge des damaligen Standards von Java, die *parametrische Polymorphie*, *Funktionen höherer Ordnung* und *algebraische Datentypen* enthält. In einem zweiten Schritt zeigten wir wie Eigenschaften von PIZZA in veränderter Form in den Java Standard übernommen wurden. Dabei sind insbesondere Generics mit der Möglichkeit von use-site Varianz und Lambda-Ausdrücke mit functional interfaces als target types zu nennen. In einem dritten Schritt haben wir Java-TX betrachtet. Java-TX ist eine Erweiterung von Java 8, in der globale Typinferenz und echte Funktionstypen integriert wurden. Schließlich haben wir in einem Ausblick gezeigt, wie Pattern–Matching in Java-TX integrierbar wäre.

Literatur

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Language Specification.* The Java series. Java SE 8 edition, 2015.

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

[OAC+19] Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2019. Version 2.13.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 146–159, New York, NY, USA, 1997. ACM.

[OZZ01] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. *Proc. 28th ACM Symposium on Principles of Programming Languages*, 36(3):41–53, 2001.

[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.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

Renaissance: Benchmarking Suite for Parallel Applications on the JVM

Aleksandar Prokopec[1], Andrea Rosà[2], David Leopoldseder[3], Gilles Duboscq[1], Petr Tůma[4], Martin Studener[3], Lubomír Bulej[4], Yudi Zheng[1], Alex Villazón[5], Doug Simon[1], Thomas Würthinger[1], and Walter Binder[2]

[1] Oracle Labs, Switzerland
{aleksandar.prokopec,gilles.m.duboscq,yudi.zheng,doug.simon,thomas.wuerthinger}@oracle.com
[2] Università della Svizzera italiana, Switzerland
{andrea.rosa,walter.binder}@usi.ch
[3] Johannes Kepler Universität Linz, Austria
david.leopoldseder@jku.at,martinstudener@gmail.com
[4] Charles University, Czech Republic
{petr.tuma,bulej}@d3s.mff.cuni.cz
[5] Universidad Privada Boliviana, Bolivia
avillazon@upb.edu

1 Extended Abstract

To demonstrate that a compiler optimization, a memory management algorithm, or a synchronization technique is useful, researchers need benchmarks that demonstrate the desired behavior and, at the same time, capture representative aspects of real-world applications. During the last decade, multiple new programming paradigms have appeared on the Java Virtual Machine (JVM), including functional programming, big-data processing, parallel and concurrent programming, message passing, stream processing, and machine learning. The JVM has evolved as a platform, too. New features—such as method handles, variable handles, the `invokedynamic` instruction, lambdas, atomic and relaxed memory operations—present new challenges for just-in-time (JIT) compilers and runtime environments.

Workloads exercising the above features potentially present new optimization opportunities for compilers and virtual machines. Unfortunately, existing benchmark suites (such as DaCapo [2], ScalaBench [5], or SPECjvm2008 [1]) do not capture these new features, because they were made in a time when such workloads did not exist. Moreover, such suites do not specifically focus on concurrency and parallelism.

To bridge this gap, we propose Renaissance [4], a new representative set of benchmarks that covers modern JVM concurrency and parallelism paradigms. Renaissance consists of 25 benchmarks (summarized in Table 1) representative of common modern patterns (including, but not limited to, big data, machine learning, and functional programming) relying on multiple existing state-of-the-art Java and Scala frameworks. The suite can be useful to optimize just-in-time

Table 1. Summary of benchmarks included in Renaissance (version 0.10.0, latest at time of writing).

Benchmark	Description	Focus
akka-uct	Unbalanced Cobwebbed Tree computation using Akka.	actors, message-passing
als	Alternating Least Squares algorithm using Spark.	data-parallel, compute-bound
chi-square	Computes a Chi-Square Test in parallel using Spark ML.	data-parallel, machine learning
db-shootout	Parallel shootout test on Java in-memory databases.	query-processing, data structures
dec-tree	Classification decision tree algorithm using Spark ML.	data-parallel, machine learning
dotty	Compiles a Scala codebase using the Dotty compiler for Scala.	data-structures, synchronization
finagle-chirper	Simulates a microblogging service using Twitter Finagle.	network stack, futures, atomics
finagle-http	Simulates a high server load with Twitter Finagle and Netty.	network stack, message-passing
fj-kmeans	K-means algorithm using the Fork/Join framework.	task-parallel, concurrent data structures
future-genetic	Genetic algorithm function optimization using Jenetics.	task-parallel, contention
gauss-mix	Computes a Gaussian mixture model using expectation-maximization.	machine learning
log-regression	Performs the logistic regression algorithm on a large dataset.	data-parallel, machine learning
mnemonics	Solves the phone mnemonics problem using JDK streams.	streaming
movie-lens	Recommender for the MovieLens dataset using Spark ML.	data-parallel, compute-bound
naive-bayes	Multinomial Naive Bayes algorithm using Spark ML.	data-parallel, machine learning
neo4j-analytics	Analytical queries and transactions on the Neo4J database.	query processing, transactions
page-rank	PageRank using the Apache Spark framework.	data-parallel, atomics
par-mnemonics	Solves the phone mnemonics problem using parallel JDK streams.	parallel streaming
philosophers	Dining philosophers using the ScalaSTM framework.	STM, atomics, guarded blocks
reactors	A set of message-passing workloads encoded in the Reactors framework.	actors, message-passing, critical sections
rx-scrabble	Solves the Scrabble puzzle using the RxJava framework.	streaming
scala-kmeans	Runs the K-Means algorithm using Scala collections.	machine learning
scala-stm-bench7	STMBench7 workload using the ScalaSTM framework.	STM, atomics
scrabble	Solves the Scrabble puzzle using Java 8 Streams.	data-parallel, memory-bound
streams-mnemonics	Computes phone mnemonics using Java 8 Streams.	data-parallel, memory-bound

(JIT) compilers, interpreters, garbage collections, as well as tools such as profilers, debuggers, or static analyzers.

To obtain these benchmarks, we gathered more than 100 candidate workloads, both manually and by scanning an online corpus of GitHub projects. We then defined and collected a set of metrics able to capture the use of concurrency-related features as well as the use of code patterns commonly associated with abstractions of object-oriented programs. We used these metrics to detect potentially interesting workloads, and to ensure that the selection is sufficiently diverse. Thanks to PCA analysis, we revealed that the set of benchmarks selected for inclusion covers the metric space differently than the existing benchmark suites.

To confirm that the selected benchmarks are useful, we analyzed them for performance-critical patterns. We demonstrated that the proposed benchmarks reveal new opportunities for JIT compilers by implementing four new optimizations in the Graal JIT compiler [3]. These optimizations have a considerably smaller impact on existing suites, indicating that Renaissance helped in identifying new compiler optimizations. We also identified three existing optimizations whose performance impact is prominent. Furthermore, by comparing two production-quality JIT compilers, Graal and HotSpot C2, we determined that performance varies much more on Renaissance than on other benchmark suites.

We also compared the complexity of the Renaissance workloads with those of other suites, by evaluating six Chidamber and Kemerer metrics, and by inspecting the compiled code size and the hot method count of all the benchmarks. Our results show that the proposed benchmark suite is as complex as DaCapo and ScalaBench, and much more complex than SPECjvm2008.

Renaissance is intended to be an open-source, collaborative project, in which the community can propose and improve benchmark workloads. Renaissance is publicly available at `https://renaissance.dev`. More information on the suite and on the described analyses can be found in our previous publication [4].

References

1. SPECjvm2008 (2008), `https://www.spec.org/jvm2008/`
2. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis. SIGPLAN Not. **41**(10), 169–190 (Oct 2006)
3. Duboscq, G., Würthinger, T., Stadler, L., Wimmer, C., Simon, D., Mössenböck, H.: An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In: VMIL. pp. 1–10 (2013)
4. Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tůma, P., Studener, M., Bulej, L., Zheng, Y., Villazón, A., Simon, D., Würthinger, T., Binder, W.: Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In: PLDI. pp. 31–47 (2019)

5. Sewe, A., Mezini, M., Sarimbekov, A., Binder, W.: Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In: OOPSLA. pp. 657–676 (2011)

Improving Stability and Portability of Haskell IDE Engine: A Case Study

Hannes Siebenhandl

Compilers and Languages
Institute of Information Systems Engineering
Vienna University of Technology
e1327006@student.tuwien.ac.at

Abstract. Haskell is a pure functional programming language that is well known in academia and industry. Despite its widespread usage, its development tools are not yet as mature as other programming languages', especially its integrated development environments, their stability, portability, and documentation.
In the course of contributing to the open source Haskell IDE Engine project, we focused on improving stability, portability, and documentation. To improve portability, we re-designed the existing build system, enhancing reproducibility and reducing sources of errors by providing a unified build script that checks for common mistakes. This simplifies proper usage of the system for application programmers. To improve stability, we reviewed existing plugins for potential crashes and documented usage preconditions. This enabled bug fixes by ourselves and other contributors to Haskell IDE Engine and its plugin ecosystem. Hoogle, for example, crashed when its database was accidentally corrupted by an application programmer.
In our talk (and in this extended abstract) we provide an overview of Haskell IDE Engine, highlight its internal architecture, and present our approach to improve stability, portability, and documentation. In particular, we report on promising positive experiences with our improvements which lead to a better user experience of Haskell IDE Engine. Finally, we discuss open issues and topics of planned future research and work.

Keywords: Haskell · Haskell IDE Engine · Development Tools

1 Background and Motivation

Haskell is a pure functional programming language that is well known in academia and industry. Despite its widespread usage, its development tools are not yet as mature as in other programming languages, especially when it comes to integrated development environments. However, this ought to improve with the progress of the current Haskell IDE Engine project, which differs from related other attempts of creating an integrated development environment for Haskell as it collaborates with the de-facto standard compiler GHC,[1] the Glasgow Haskell

[1] http://haskell.org/ghc/

Compiler, and aims to be extensible. Moreover, Haskell IDE Engine does not strive for duplicating efforts since the Haskell ecosystem provides already powerful tools to transform Haskell source code, lint code, and format.

2 Overview of Haskell IDE Engine

The Haskell IDE Engine project consists of four main components: (1) an implementation of Microsoft's Language Server Protocol,[2] (2) an abstraction over GHC to build Haskell package components, (3) an API for plugins, and (4) an implementation of the Language Server, which combines the previous three components. Together and combined, these four components constitute the core of Haskell IDE Engine and are essential for its success.

Supporting multiple (editor) front-ends and back-ends leads to the well known $M \times N$ problem. In the context of Haskell IDE Engine, supporting an editor requires extra maintenance for a back-end, and for each front-end, multiple back-ends need to be supported. Haskell IDE Engine solves this $M \times N$ problem using the Language Server Protocol which provides a common interface that each front-end editor and back-end program can use to communicate with in the fashion of the UNCOL concept proposed by Melvin E. Conway [Co58].

Extensibility is achieved through the concept of plugins that are easy to write and have access to gathered information about type-checked modules. Plugins can access information via the plugin API. Making these plugins as powerful as possible requires that requests must be deferrable until the necessary information is available. Free monads allow for solving this problem. While plugins can easily be written, they currently lack separation from the main application, increasing the compilation time considerably.

To build Haskell packages, a compiler is needed. Currently, 'ghc-mod' [3] provides an abstraction over GHC's compiler errors, warnings and build instructions. It is the most error-prone component due to the complexity surrounding Haskell project management and is currently in the process of a re-design.

The Language Server is the component that contains the implementations of plugins, their configuration, and provides the actual interface of the Haskell IDE Engine.

3 Improvements

Over the course of contributing to the Haskell IDE Engine open-source project, we identified various problems and difficult situations that we wanted to improve. Firstly, stability of building the projects in a reproducible and portable way proved difficult due to a complex sequence of installation steps. Secondly, stability of the Language Server itself varied, it crashed because some components did not check their required preconditions correctly.

[2] https://microsoft.github.io/language-server-protocol/
[3] https://github.com/DanielG/ghc-mod

We succeeded in improving the build-stability of the project by replacing the previous, platform dependent build instructions with a build system based on 'shake.' [4] Additionally, the new build system reduces sources of errors by checking for common errors, which simplifies proper usage of the system for application programmers.

Improving stability of the Language Server proved to be more difficult. Each plugin had to be validated for correctness. In the process, we documented required preconditions and added test-cases. This led to the discovery of multiple bugs that we and other collaborators of the Haskell IDE Engine project were able to fix.

4 Conclusion

Our initial practical experiences with our contributions to Haskell IDE Engine show that they increase the stability, portability, and documentation of the project. However, the overall project is not yet finished. Especially support for multiple Haskell components is still missing. Another issue which we did not yet mention and discuss is performance of the Language Server. Performance is currently considered a problem of only secondary importance. However, once building projects will be reliable, performance issues will gain priority. Possibly, this might improve with the Google Summer of Code 2019 [5], where there are multiple projects aiming at further enhancing the underlying Haskell project build system and performance of the Language Server.

References

[Co58] Melvin E. Conway. *Proposal for an UNCOL.* Communications of the ACM 1(3):5, 1958.

[4] https://shakebuild.com/

[5] https://developers.google.com/open-source/gsoc/timeline

Subsidiary Meta-Programming for Higher-Level Programming Concepts

Daniel Speicher and Armin B. Cremers

B-IT, University Bonn, Bonn, Germany
{dsp, abc}@bit.uni-bonn.de

Abstract. While programming language development is the realm of precise semantics and analytical concepts, developers describe their implementation strategies and design rules often by examples and metaphors. The informal nature of this conceptualization gives developers the freedom to transfer ideas from one context to another context and to adapt them to specific situations. However, this makes it difficult to provide a tool that continuously offers distinct quality feedback. We therefore suggest to codify and clarify the executable code quality knowledge iteratively and incrementally, and to implement it at different levels of generality, where more specific layers may adapt generic layers. The suggested approach is subsidiary to language evolution as well as to the creative design process. Codified constructive concepts may highlight and safeguard the structure of the interplay of language elements, libraries and applications. They may serve as a precursor to programming language features. The adaptable implementation makes it easy to rely on existing expressions of design ideas in the code and to integrate design ideas that are specific to a certain context.

Keywords: Code Quality · Iterative and Incremental Development · Meta Programing · Executable Code Quality Knowledge.

What does it take to provide a helpful tool that continuously offers distinct quality feedback? Most importantly any false alarm diminishes the trust a developer has in the tool and increases her habituation to unjustified warnings. This paper suggests to encode Executable Code Quality Knowledge in a schema that is friendly to iterative and incremental development and describes tools that help evolving this kind of encoded knowledge.

After a quick look at the informal nature of developer concepts (1) we will describe based on our technical infrastructure (2) a model for executable code quality knowledge (3) and present short examples (4). We will sketch the process that we envision (5) and after illustrating that we may use the model on the model (6) conclude (7).

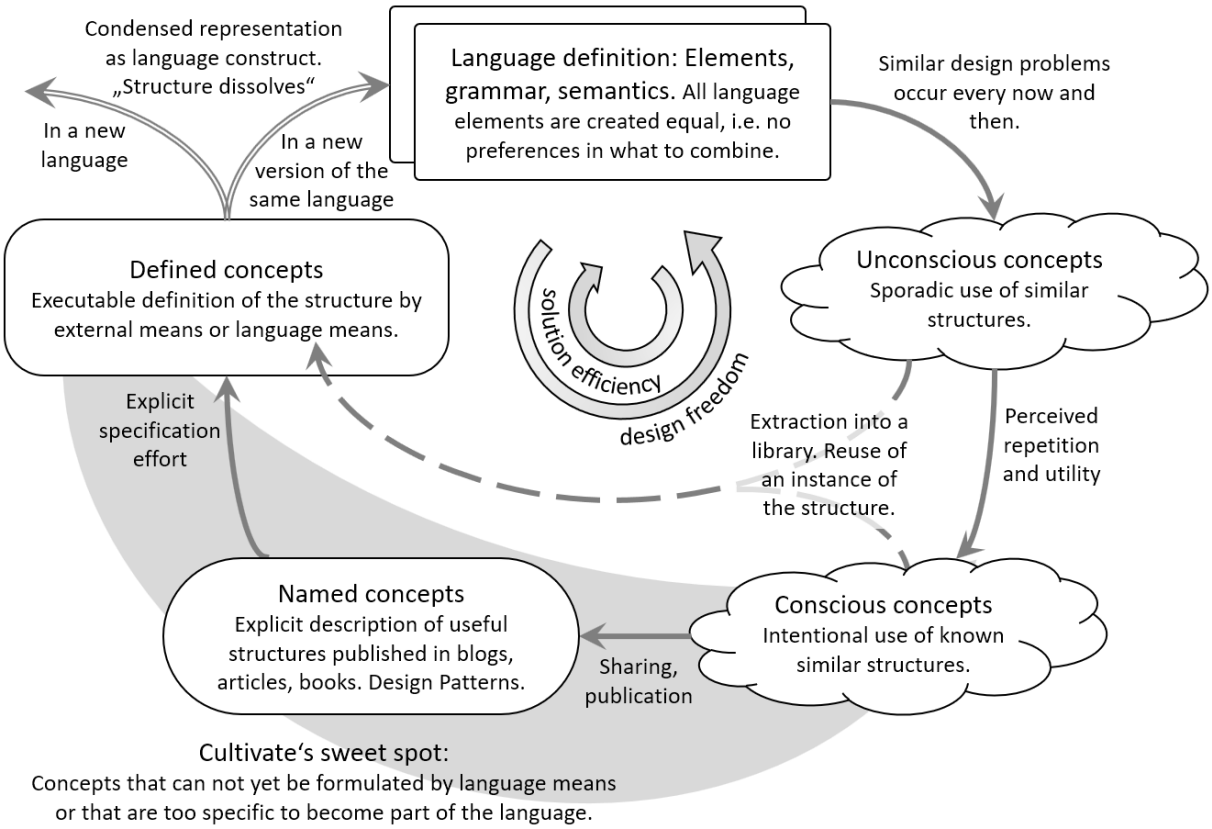


Fig. 1. How concepts of code structure appear and sometimes disappear again.

1 Prologue

1.1 Design Patterns as Proven Solutions in a Context

One way to look at design patterns [2] is to consider them as proven solutions in a context to reoccurring problems. Part of the context may be the programming language as some patterns become considerably simpler in a dynamic language [7] or in an Aspect-Oriented Language [4]. While individual design patterns might be a transient phenomena, design patterns will arise again. Here is a list of examples where we see aspects of this evolution.[1]

- Async-Await language construct replacing Promises
- Dynamic Dispatch simulated with C switch case statements
- Type hints precursor to real typing in PHP
- Policies instead of around advices (AspectJ)
- Extensible objects pattern instead of mixins/inter-type declarations
- Properties like in Java Beans are language constructs in ObjectPascal or .Net

[1] The reader might recognize that we are trying to make a humble comment on the extensively discussed question of whether design patterns are missing language features http://wiki.c2.com/?AreDesignPatternsMissingLanguageFeatures. More examples may be found at this link.

– Encapsulated downcasts as workaround for missing generic types
– Hand-coded observer chains before full blown reactive programming

As soon as a pattern is perfectly formalized[2] it might be only one step away from dissolving into language constructs. Conversely we should not be too surprised if most pattern that did not yet dissolve have some degree of freedom. Some patterns may evolve in a specific context and not of general use. While some design pattern might disappear into new language versions, developers will find recurring solutions in this new language as well.

1.2 Analytical and Constructive Human Concepts

Most concepts computer scientists use are *analytical concepts*. They have a clear definition. Something falls under a concept (is in its "extension") if and only if it fulfills certain conditions (its "intension"). Whether something is true can be found through logic inference. Johnson-Laird list in [5] two further kinds of concepts. The second are *natural concepts* that rely on the sciences. And then there is "[a]nother class of concepts [...] exemplified by games, chairs, and melodies. They are neither analytic nor based on empirical theories. They have instead, what might be termed a constructive foundation, that is, they have no objective correlates in the physical world. No scientific theory postulates an analysis of what counts as a game or a chair or a melody." These concepts are called *constructive concepts*.

Furthermore it is helpful to know (e.g. from [3]) that non-analytical concepts can often not be defined by necessary and sufficient conditions. They may have prototype structure ("a fruit is something like an apple") or exemplar structure ("a dog is an animal like Lassie next door") or some more theory built in. The extension of these concepts do not necessarily have sharp boundaries ("is an olive a fruit or not?"), a phenomenon we quickly realize in design patterns, once we consider it to be possible.

1.3 Quality Judgment and Higher Level Concepts

To understand the deeper reasons for the prevailing high number of false positives in bad smell detection ([9], [8]) we had conducted an in depth exploratory analysis into the nature of reasons (mostly unpublished, but an overview can be found here: [10]). Our hypothesis was that the developer had traded a general design rule for specific design ideas of higher value. We found not only generic ("data classes are actually useful") but as well very specific ideas ("only `DrawingView` holds the selection"). We were surprised to realize that concepts within the design rule had to be considered constructive concepts ("object creation by varying means" instead of just "object instantiation").

[2] Languages to formalize patterns are e.g. collected in [13].

2 Underlying Technical Infrastructure

We realize the Executable Code Quality Knowledge as Prolog predicates. The architecture overview in Fig. 2 illustrates how the Eclipse plug-in Cultivate[3] makes this possible with the help of the plug-ins JTransformer[4], PDT and JDT and SWI-Prolog[5] in the background. JTransformer translates each node of the

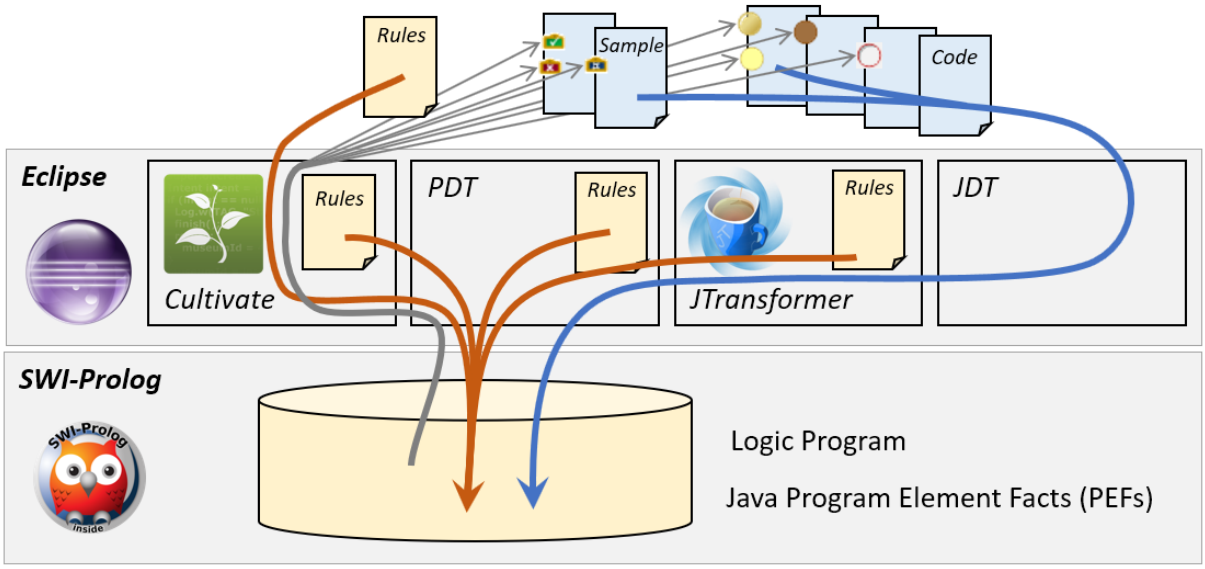


Fig. 2. Architecture Overview

Abstract Syntax Tree created by the JDT into a fact in the Prolog fact base. For a call it creates a fact with 8 arguments.[6] A call from within a method to the method itself can then e.g. be found with the following predicate:

```
recursive_call(Call) :-
    callT(Call, _, Method, _, _, Method, _, _).
```

A return statement passing a field value can be found with:

```
returns_field(Statement, Field) :-
    returnT(Statement, _, _, FieldRead),
    fieldAccessT(FieldRead, _, _, null, Field, _).
```

Note that Fig. 2 places some Prolog code in the workspace of the developer.

[3] https://sewiki.iai.uni-bonn.de/cultivate
[4] https://sewiki.iai.uni-bonn.de/jtransformer
[5] https://www.swi-prolog.org/
[6] The fact representation up to Java 8 is carefully documented at https://sewiki.iai.uni-bonn.de/research/jtransformer/api/java/pefs/4.2/java_pef_overview

3 Executable Code Quality Knowledge

We present a model for encoding code quality evaluations as well as analytical and constructive code concepts. The model is designed with the functional goal to support precise quality feedback and the non-functional goals of simplicity and evolvability.

There are two dimensions to executable code quality knowledge: truth and quality. We describe the means to encode which program elements fall under which concept in 3.1 and the means to encode quality judgment in 3.2. Everything is encoded by predicates so that the whole executable knowledge just consist of a logic program that follows certain conventions and is enriched by a few "meta-facts" specifying the role certain predicates are playing. The interplay is illustrated in Fig. 3 and Fig. 4. Predicates are represented by the class Published Predicate and some clauses by the class Augmentation. Classes in light gray and relationships drawn in black correspond to meta-facts. All other meta-model elements are purely conceptual although most distinctions and associations may easily be inferred from the structure of the logic program and the meta-facts.

3.1 Structures, Concepts and Signs

The sub-classification of Concept in Fig. 3 is not meant to be exclusive. Each concept is either elementary or structured and as well either analytical or constructive. Thus we have *analytical elementary concepts*, *analytical structured concepts*, *constructive elementary concepts*, and *constructive structured concepts*. We discuss the two dimensions of this classification separately.

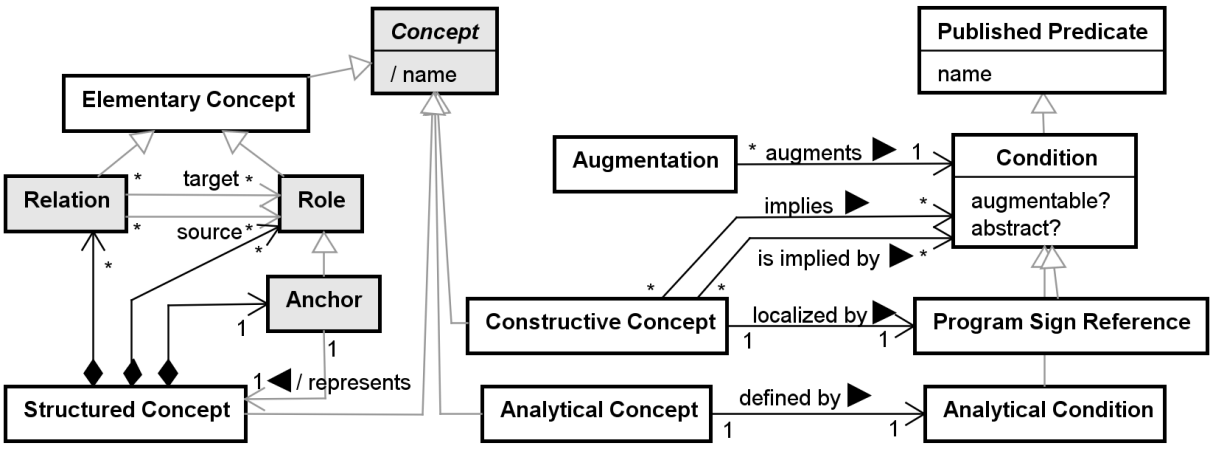


Fig. 3. Meta-Model for Concept Definitions

Elementary Concepts and Structured Concepts Sometimes we categorize single program elements: "This is 'a recursive call', 'a field return', 'a generalizable type declaration' or 'a singleton class'." We talk about single program

elements and the extension of such an *elementary concept* is a set of program elements. However, already in these simple examples these concepts hardly make sense without taking further program elements into account. A *structured concept* categorizes a few program elements together, defines the *roles* the single elements play and which *relations* hold between them. For simplicity, we expect the players of one specific role, the *anchor*, to represent the instance. The extension of such a concept is thus a set of structured collections of program elements. There are different but mostly obvious ways to describe such a structured collection, e.g. as a directed graph with multiple labels on nodes (role names) and edges (relation names) where only one node is labeled with the anchor name. Having an explicit anchor element makes it easy to refer to the instance and to distinguish instances. During concept definition the developer is pushed to consider the conditions that separate one instance from the other and to find a program element that represents these conditions.

Let us discuss the Observer pattern in these terms. What are the roles, what are the relations what is a good choice of an anchor? There are the roles for the involved types: A class Concrete Subject, an interface Observer, a class Concrete Observer and maybe an abstract class Subject. Some methods are essential for the pattern, at least the methods that notify when state changes and those that are called. There are at least the implements, extends, and calls relation. The Observer interface would not be a good anchor as many [Concrete] Subjects might refer to it. The [Concrete] Subjects is not a good anchor as it might notify different collections of Observers for different reasons. Thus we suggest to use the Observer Collection itself as anchor.

To encode the structural concept, create a file containing a Prolog module. The module name is the name of the concept. Add for the anchor an unary predicate, for the roles binary predicates and for the relations ternary predicates. The names of the predicates are considered the names of the corresponding anchors, roles and relations. Finally publish the predicates by adding facts of the form `cqc:anchor(<anchor name>)`, `cqc:role(<role name>)`, and `cqc:relation(<relation name>)`. For elementary concepts define a predicate in any module. The predicate name is considered the concept name. Add the metafact `cqc:concept(<concept name>)`. The central module `cqc` serves as a registry for the relevant predicates and prefixing facts or predicates with a module name is a way to add them to that module.

Analytical Concepts and Constructive Concepts For concepts like "recursive call" and "field return" we can give immediate definitions and we already illustrated our infrastructure with encodings of these concepts. These are two simplistic examples of *analytical concepts*. Involved concepts like a "generalizable type declaration" (i.e. a variable declaration that uses a type that could be changed into a more general type without loosing the type safety of the program, see e.g. [14] and [11]) belong into the same category. Putting potential questions of computability aside, we know whether a certain program element falls under the concept or not, we can give a clear definition, and this definition

is independent of aspects that the compiler ignores. Examples of *constructive concepts* are "singleton class" or "configuration method". It is easy to find prototypical cases for them where every developer would agree that a certain program element falls under the concept. But it is as well not difficult to imagine program elements where such a consensus would not be so easy be found: "Can a class with a public constructor be a singleton?" "May we call a class that has no own mechanism to avoid a second instance a singleton, if we may otherwise infer that there will only be one instance?" "What kind of methods does a configuration method call besides setter methods?" "How many objects may be affected by a configuration method?" "Could a configuration method contain a long running computation?" Thus, the extension of these concepts as well as its intension is subject to developer judgment. Since we want to use the executable code quality knowledge for distinct continuous feedback, we suggest to encode these concepts only as far as we are confident about them. We rely on the hints the programmer gave in the code. Anything that is essentially not interpreted by the compiler (like identifiers and placement in certain packages) is free to be used to signal intended concepts. We call these hints *program signs*. The developer may as well add necessary and sufficient conditions by adding a meta-fact `cqc:concept(<concept name>, [<necessary condition 1>, ...], [<sufficient condition 1>, ...])`.

Augmentable Conditions The program signs indicating the extension of a concept may only be known in a specific context. Some conditions should be adapted in the context of design ideas that are only present in a certain context. We might only be confident about the validity of an adaptation in a specific context. For all these cases we need an adaptation mechanism.

It is possible to add additional clauses to predicates defined in an earlier module from a later module. The predicate needs to be declared `multifile` in the original model (then it is *augmentable*). In the later module the head of the predicate needs to be prefixed with the name of the original module. This adds the clause to the original predicate and increases the solution set of the predicate.

3.2 Metrics, Smells and Shines

When we consider a "generalizable type declaration" an opportunity to improve the code, we should not use the corresponding predicate to define a concept but to define an *analytical smell*, expressing our judgment that something has to change. All smells are defined with respect to a concept, so that we may formulate very specific smells. A *constructive smell* is defined via a program sign reference and may be used to express that certain program signs not recommended or to boost expressions of strong improvement demand in names or comments. A *metric smell* is defined by a metric condition that uses some metrics. Feature envy that we mention in 4.3 is an example. All smells in [6] are of this form. The implementation supports not only smells but as well *shines*. They are technically

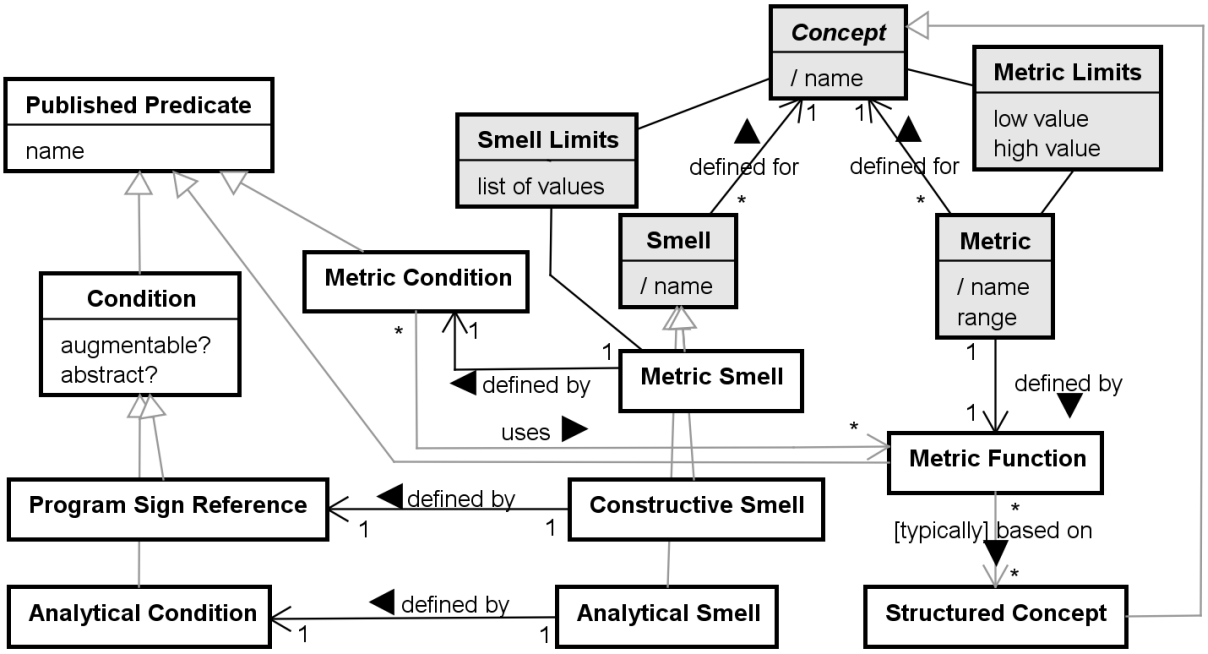


Fig. 4. Meta-Model for Qualities

completely identically to smells but are just interpreted positively. Thus imagine in Fig. 4 as well classes Shine with its three subclasses with the same outgoing associations and an association class Shine Limits.

4 Examples

The following three examples illustrate different aspects of our approach. We use the SINGLETON pattern to illustrate the feedback provided by the user interface. For a fragment of the corresponding code see Fig. 7. In 4.2 we mention a structure for which most of the work is done by the compiler. Finally we comment in 4.3 about the augmentation points in the analytical structure below the FEATURE ENVY smell.

4.1 Singleton Pattern

The three Java editors and two structure views in Fig. 5 show the two instances of the SINGLETON pattern in JHotDraw 5.1. The definitions behind the quality feedback and the structure will be illustrated to some part in Fig. 7 and Fig. 8.

The very idea of a SINGLETON class is to have not more than one instance of the respective class. Consequently, there is a bad smell (●) on the INSTANTIATION "`new Iconkit(this)`" in the method `init()` of the class `DrawApplet` seen in the first editor: The SINGLETON class `Iconkit` is instantiated outside the sphere of control of the SINGLETON class itself. Conversely, there is a shine (●) on the SINGLETON class `Clipboard` in the third editor: All constructors of this class are private so that the class can not be instantiated in any other class. A slightly more resolute stance on what is to be expected of a SINGLETON is illustrated

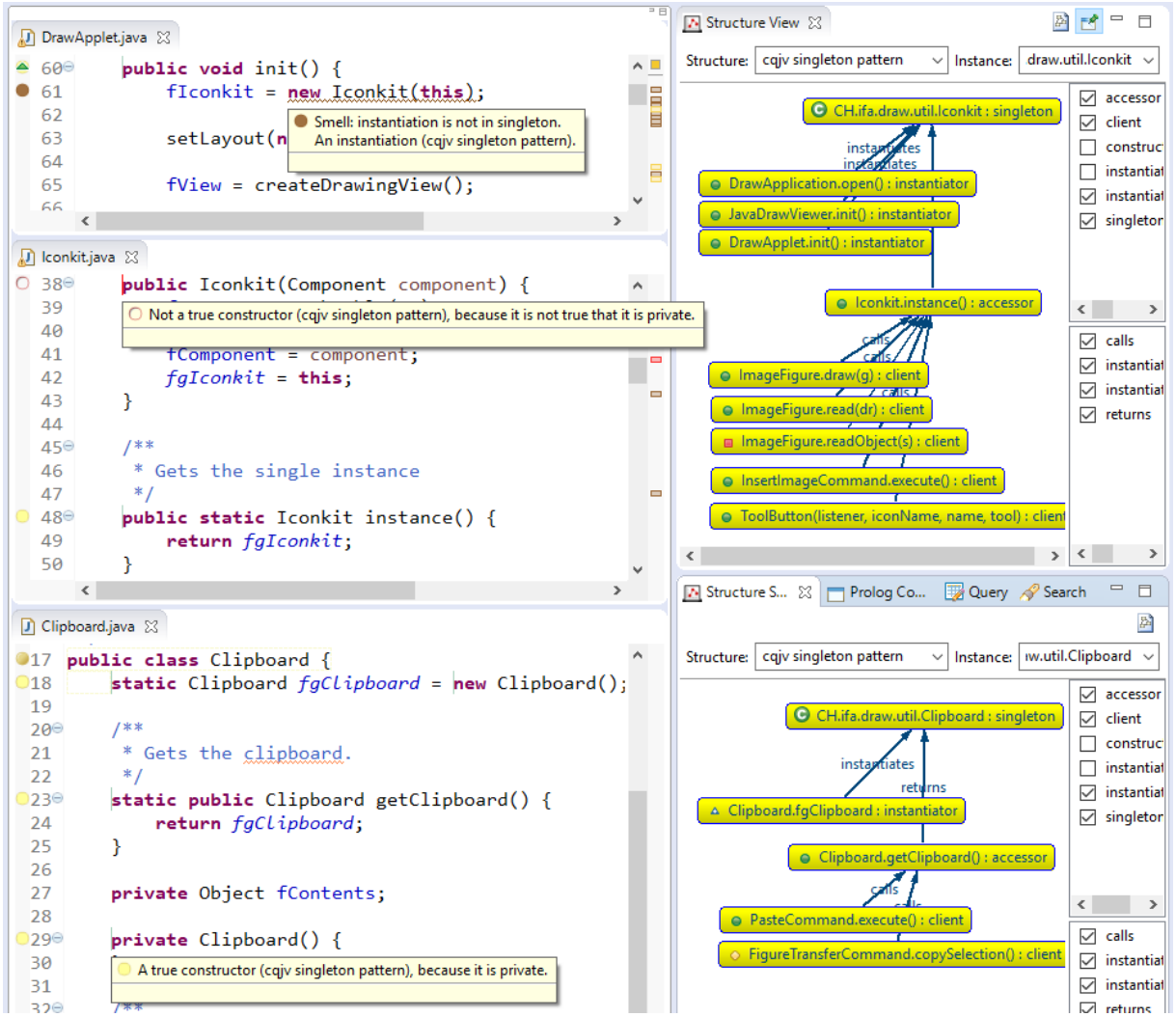


Fig. 5. Quality feedback and structure visualization in Cultivate.

on the Constructors. To be a true constructor of a singleton, we expect it to be private. The constructor `Iconkit(Component)` seen in the second editor does not fulfill (○) the idea of a constructor of a singleton as it is public. The constructor `Clipboard()` seen at the bottom of the third editor however is a true (○) constructor of a singleton as it is private. The structure views on the right show the Singleton classes, the Accessor methods to access the single instance, the Client methods that call the Accessor methods to access the instance, and finally the Instantiator executables that instantiate the Singleton. In the first view we can see that `DrawApplet` is not the only Instantiator of the Singleton `Iconkit`. In the second view there is only one. The structure view allows to navigate to the source code. A double click on the node "Iconkit.instance() : accessor" in the middle of the first structure view activates the editor for `Iconkit.java` and focuses on the Accessor method `instance()`. Which of the roles and relations are shown in the structure view can be selected on the right.

As a byproduct of the given model, the tool may search for *latent* instances of a concept, i.e. program element that fulfill all necessary conditions and at least one sufficient condition in case any are given. We give an example in Fig. 6.

For the two SINGLETONS shown in Fig. 5 we had added the knowledge that exactly these two classes are meant to be SINGLETONS to the project. If we remove this knowledge again, the tool could tell us (!) that these classes would fulfill the concept of a SINGLETON. We could decide to follow a naming convention to call the ACCESSOR method "`getInstance(..)`" or "`instance(..)`" and consider any class that has a static method with this name a SINGLETON. Using a RENAME refactoring on "`getClipboard()`" to name it "`getInstance()''` would express that the class is meant to be a SINGLETON and we would see again the feedback shown in Fig. 5.

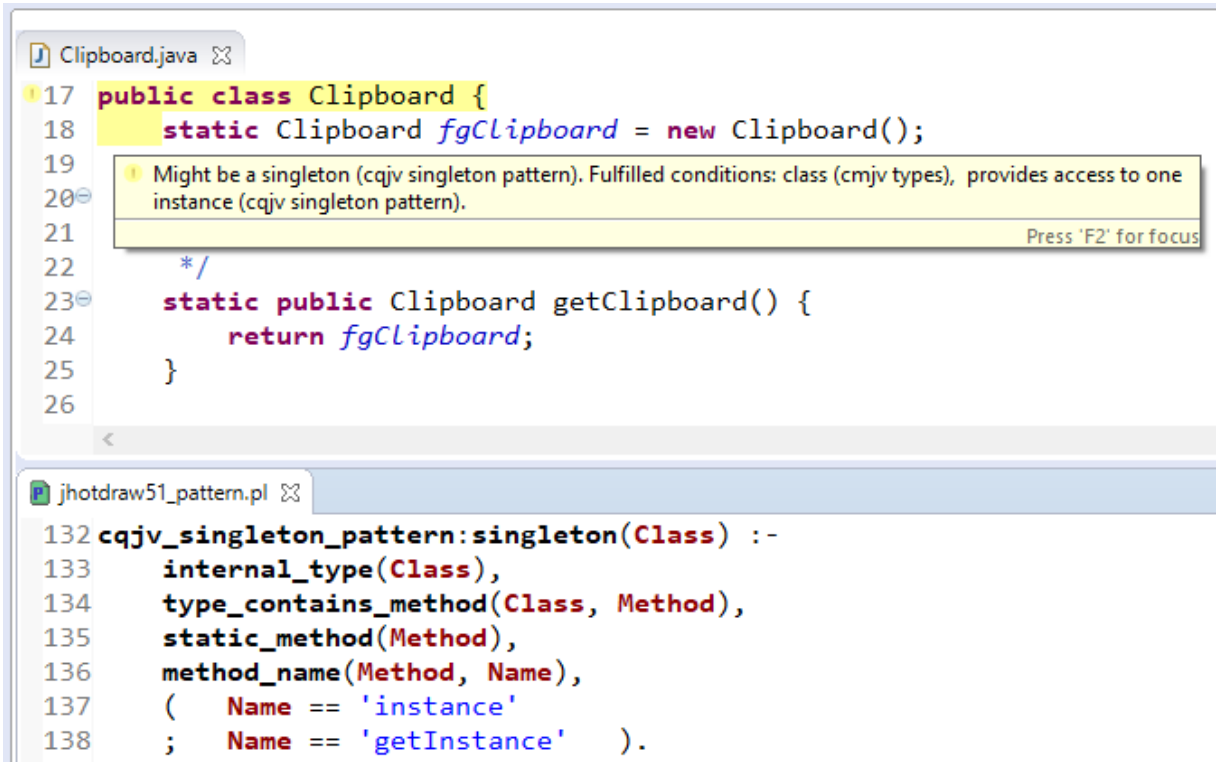


Fig. 6. A latent instance of the SINGLETON pattern.

4.2 Java Packages as Modules

The access modifiers on types in Java can be used to turn a package into a module enforcing access restrictions. Simon Brown[7] suggests that all but a few types in packages should not be public. In this way the compiler would check

[7] The generic idea to actually use accessibility modifiers on types to turn packages into modules is presented by Simon Brown in his talk "Modular Monoliths" at the conference GOTO Berlin 2018. It is embedded in a comprehensive discussion of different modularization styles (https://youtu.be/5OjqD-ow8GE?t=1980). Let us

the access rules imposed by the architecture. Listing 1.1 shows how to encode the core idea:

```
:- module(cqjv_c4_component, [protected_type/1]).
/* [... import predicates from other modules ...] */
:- multifile package_interface_type/1.
cqc:concept(protected_type, [is_type, is_not_public]).
protected_type(Type) :- type(Type), not(package_interface_type(Type)).
```

Listing 1.1. Encoding of the core idea of a package as component: All types but the PACKAGE INTERFACE TYPES should not be public.

We left `package_interface_type/1` open for augmentation so that the general rule can be configured in the specific context, where we know which types belong to the PACKAGE INTERFACE. Listing 1.2 illustrates how this might look like:

```
% All types named "*Service" belong to the package interface
cqjv_c4_component:package_interface_type(Type) :-
    type_name_ends_with(Type, 'Service').
% All "ports" in the hexagonal architecture belong to the package interface
cqjv_c4_component:package_interface_type(Type) :-
    port_type(Type).
```

Listing 1.2. Configuration of the package as module. Later we define all service types and all PORT TYPES (defined elsewhere) to be PACKAGE INTERFACE TYPES.

We express as much as possible with the means of the language and let the compiler do the actual access check. Only if we want to encode our intention to have a certain category of types as package interface, we need to encode this as additional code quality knowledge.

4.3 Feature Envy

We have defined an analytical structure for the FEATURE ENVY smell. A method has the code smell FEATURE ENVY, if it "seems more interested in a class other than the one it actually is in[, where the] most common focus of the envy is the

quickly summarize them. Assume the implementation consists of a Web controller providing the user interface (W), a service implementing business logic (B), its interface (I_B), and a data store (D) with an explicit interface (I_D). The structure of usage ($\rightarrow$) and interface implementation ($\Leftarrow$) within in the system ($\rightarrow W \rightarrow I_B \Leftarrow B \rightarrow I_D \Leftarrow D$) may then be separated in different ways into modules ($[\ldots]$): In a *layered architecture* the interfaces would mark the boundaries of the layers ($\rightarrow [\mathbf{W}] \rightarrow [\mathbf{I_B} \Leftarrow B] \rightarrow [\mathbf{I_D} \Leftarrow D]$). Alternatively the main separation criterion may be features so that for one feature elements of all three layers may belong to a *vertical feature slice* ($\rightarrow [\mathbf{W} \rightarrow I_B \Leftarrow B \rightarrow I_D \Leftarrow D]$). In a *hexagonal architecture* (or "ports and adapters" or "clean architecture" or "onion architecture") the module containing the business logic has not outgoing dependencies ($\rightarrow [\mathbf{W}] \rightarrow [\mathbf{I_B} \Leftarrow B \rightarrow \mathbf{I_D}] \Leftarrow [D]$). Brown favors to *package by "component"* with a separated user interface ($\rightarrow [\mathbf{W}] \rightarrow [\mathbf{I_B} \Leftarrow B \rightarrow I_D \Leftarrow D]$). Depending on the module boundaries different types form the public interface (formated as bold).

data." [1, p. 80]. The specification follows the suggestions from R. Marinescu and M. Lanza in [6, pp. 84-87]

The anchor of the structure is the EXECUTABLE, the roles are ACCESS, ACCESSED_FIELD, OWN_CLASS, ACCESSED_OWN_FIELD, ACCESSED_FOREIGN_FIELD and ACCESSED_FOREIGN_CLASS. The relations are EXECUTABLE_CONTAINS_ACCESS, ACCESSES, ACCESSED_FIELD_IS_IN_CLASS, IS_IN, EXTENDS. Counting role players gives four metrics: ACCESS__TO__OWN__DATA, ACCESS__TO__FOREIGN__DATA, FOREIGN__DATA__PROVIDERS, LOCALITY__OF__ATTRIBUTE__ACCESS. Based on these metrics the smell FEATURE__ENVY is defined as:

```
feature_envy(Executable, SomeFeatures, LowLocality, FewProviders) :-
    access_to_foreign_data(Executable, ATFD), ATFD > SomeFeatures,
    locality_of_attribute_access(Executable, LAA), LAA < LowLocality,
    foreign_data_providers(Executable, FDP), FDP =< FewProviders.
```

Listing 1.3. Implementation of the smell FEATURE ENVY.

Although the smell might seem to be purely analytical, we found the need to define three augmentation points for constructive ideas:

- METHOD_MAINLY_ACCESSES_FIELD/2 – The method mainly accesses a field so that a call to the method expresses as much envy as a direct field access. Trivial accessor methods, i.e. methods that do nothing but accessing the field, are a prototypical example and already subsumed by this predicate.
- ALMOST_COMMONLY_OWNED_FIELD/1 – Fields that could be considered almost common property so that no executable has a reason to get envious about them. Constants are already considered to be almost commonly owned fields.
- EXECUTABLE_HAS_FREE_ACCESS_TO_CLASS/2 – Some classes are meant to be accessed freely by certain executables. The prototypical example here are VISIT METHODS in the CONCRETE VISITORS in the VISITOR PATTERN. These methods are meant to work on the attributes of the VISITED ELEMENT.

5 Organic Growth of Code Quality Knowledge

We use annotated code samples as test cases for the executable code quality knowledge.[8] In Fig. 7 we see in the editors on the left Java files that contain a very prototypical example of the SINGLETON pattern annotated with expected analysis results. The Prolog file in the editor on the right defines a structure containing the most important roles and relations for this form of the pattern. A small configuration file shown at the lower left augments the structure with the information that the given class `Singleton` is indeed meant to be a SINGLETON.

[8] See https://sewiki.iai.uni-bonn.de/_media/private/daniel/public/wsr_2012_museums_slides.pdf for a comparison with three other testing strategies that we had employed before.

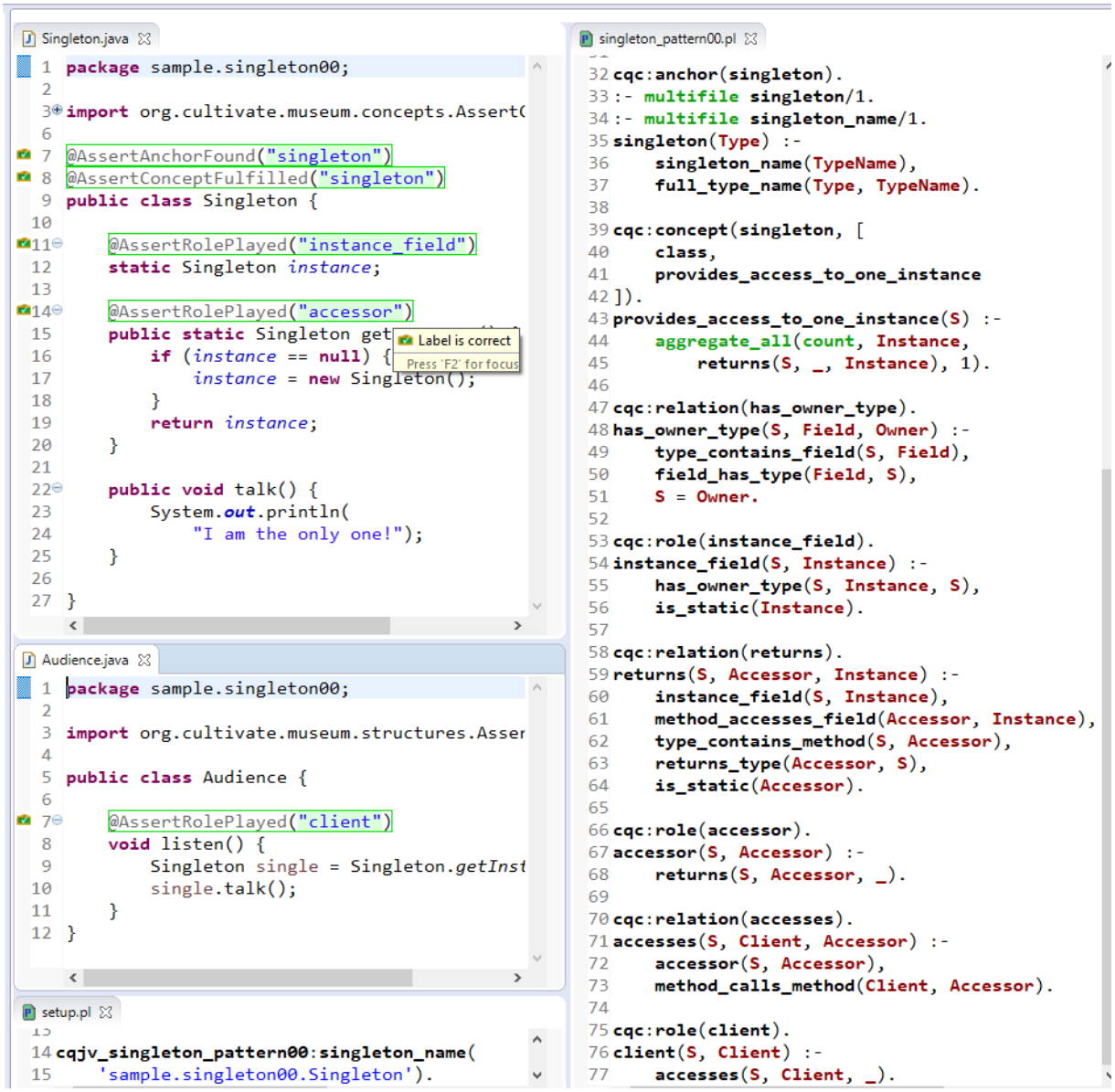


Fig. 7. Code museum and definition of the SINGLETON pattern, version 0.

The class `Singleton`, the field `instance` and the methods `getInstance()` and `client()` are annotated with the expected roles SINGLETON, INSTANCE, ACCESSOR and CLIENT respectively. Cultivate highlights the role names in light green indicating that the respective predicates actually match on the annotated program elements.

In Fig. 8 the Java files on the left present again the SINGLETON pattern but now with a variation we saw in the class `Iconkit` in Fig. 5: The SINGLETON is not instantiated in the SINGLETON itself. The Prolog file on the right has already some but not all of the changes expressed by the annotations in the Java files. The class `Audience` and the file `setup.pl` are like in the previous figure. The same is true for the upper part of the structure definition.

The annotations refer to the three new roles INSTANTIATOR, INSTANTIATION and CONSTRUCTOR OF A SINGLETON of the method `inviteTheSingle()`, the in-

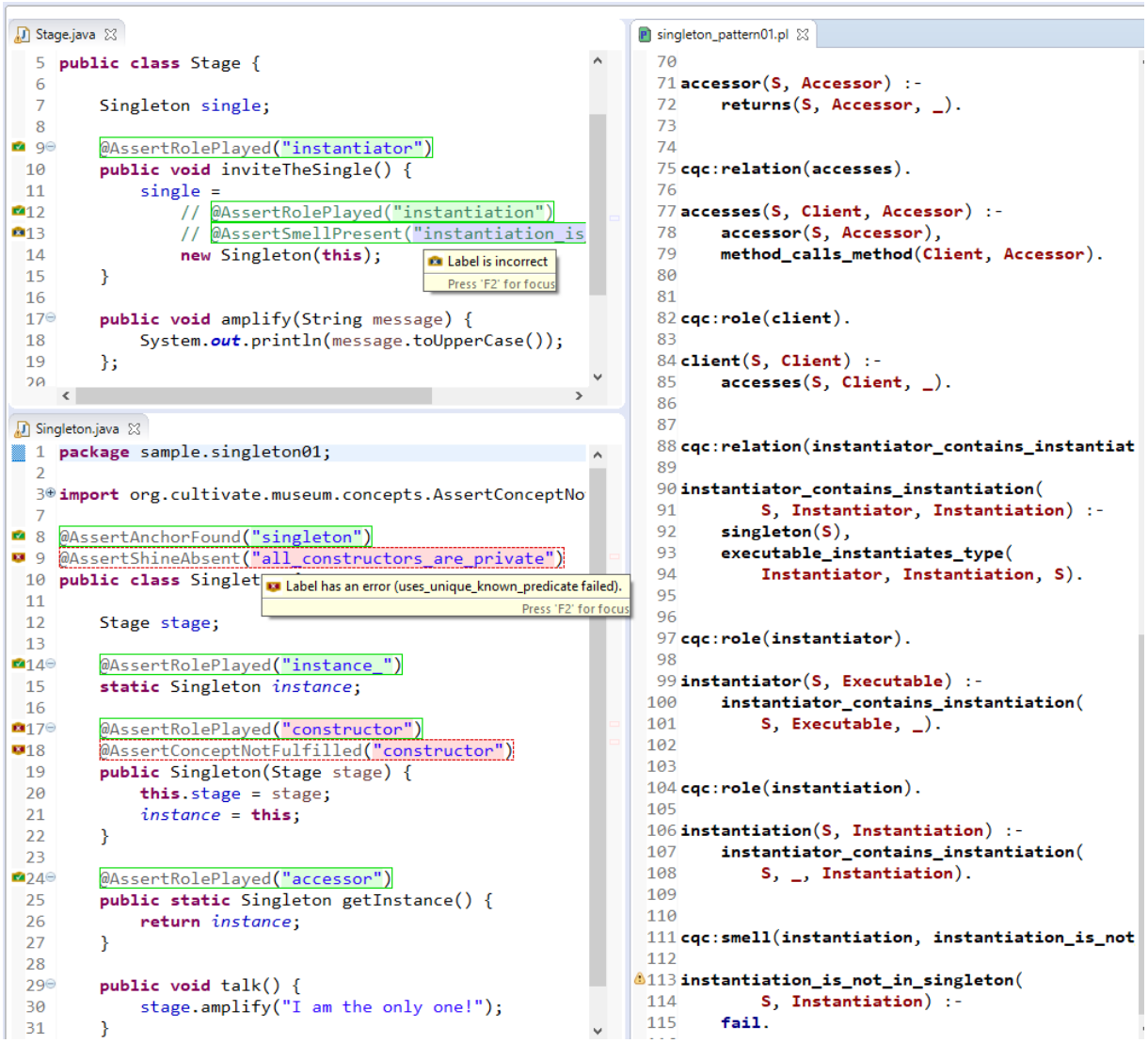


Fig. 8. Code museum and definition of the SINGLETON pattern, version 1.

stantiation expression "`new Singleton(this)`" and the constructor (in the general sense) `Singleton(Stage)`. On the INSTANTIATION we expect a smell INSTANTIATION IS NOT IN SINGLETON. We expect as well a shine ALL CONSTRUCTORS ARE PRIVATE but it should not match the given SINGLETON as this has a public constructor. This public constructor should as well be identified as being not a true CONSTRUCTOR OF A SINGLETON. To highlight that these two latter expectations are negative, these annotations are shown with a dashed red frame instead of a solid green one.

The previously defined roles as well as the roles INSTANTIATOR and INSTANTIATION apply where they are expected and the corresponding labels are thus highlighted in light green and an icon of a green frame is shown at the left border ( ). The smell INSTANTIATION IS NOT IN SINGLETON is already known as a predicate (see bottom of the structure definition) but does not yet apply where it is expected ( ). The absence of the shine ALL CONSTRUCTORS ARE PRIVATE can not even be tested as the corresponding predicate is not yet defined ( ).

The same holds for the role CONSTRUCTOR and its necessary conditions. The corresponding labels are thus highlighted in light red or light blue respectively.

Growing Executable Code Quality Knowledge Organically Based on the given model we suggest a process [12] that complements the practices of test-first programming and local refactoring (cycle in the upper left) with sample-first definition of meta-knowledge followed by global refactorings (cycle in the lower right). If the developer recognizes that a smell is a false positive, she may revise samples and rules (cycle on the right).

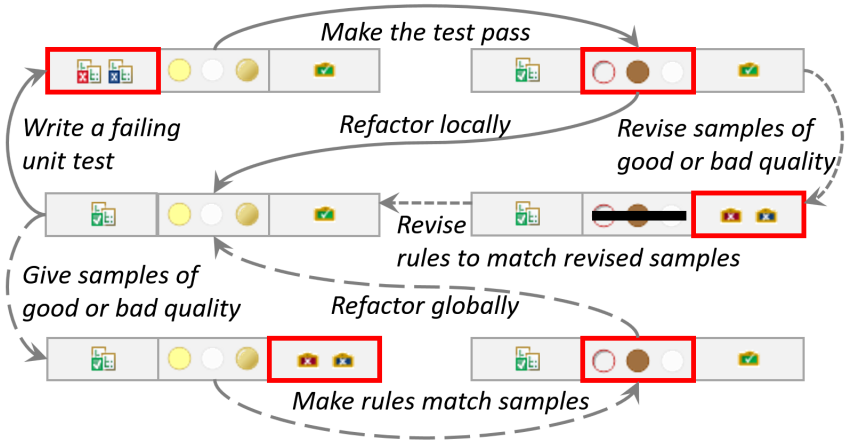


Fig. 9. The process of growing Executable Code Quality Knowledge organically.

Stepwise Increasing the Binding Character The model is designed to allow for iterative and incremental development. Although constructive concepts are by its very nature sometimes cloudy, defining them as part of the Executable Code Quality Knowledge is only helpful as far as they can be made precise. If they can not be made precise now or should not be enforced now, the model leaves the opportunity to defined them later.

Assume we have a rule r that describes a potentially desirable quality of certain program elements. We can start by just querying the system for those program elements that fulfill the rule r. If we find that the quality is worthwhile to be considered further, we may add a *shine* highlighting r program elements characterized by a certain *concept* c_1. In case we come to the conclusion that we really want to have this property, at least on a potentially narrower concept c_2, we could define a *smell* $\neg r$ on c_2. If we finally come to the conclusion that program elements that are meant to fall under a certain concept c_3 do not really fulfill its expectations if this property is not true, we may add r to the necessary conditions of c_3.

$$\text{query } r < \text{analytical concept } r < \text{shine } r < \text{smell } \neg r < \text{necessary condition } r$$

"When a team lets the executable code-quality-knowledge grow organically, they can restrain the rules to those concepts and smells they thoroughly understand and consider relevant enough to commit to them and to let them be checked automatically." [12]

6 Executable Code Quality Knowledge on Executable Code Quality Knowledge

Our definitions in 3 did not rely on the arguments of the predicates being identifier of Java Program Element Facts. We may as well define predicates analyzing predicates. As IDs we can use module names, predicate indicators and clause references.

As augmentation is not immediately visible in the code an additional visualization could be helpful. In Fig. 10 we see that the augmentable predicates `singleton/1` and `singleton_name/1` get augmented from the modules `jhotdraw51_pattern` and `cqjv_museum_singleton_setup`.

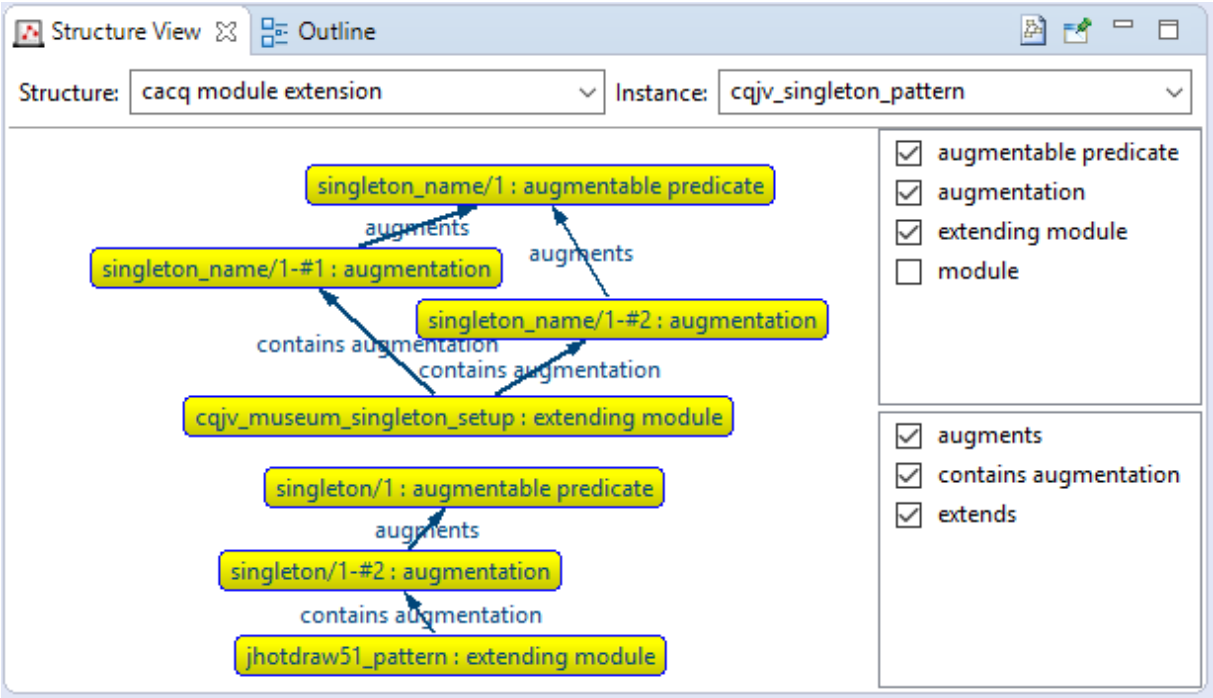


Fig. 10. A structure within the Executable Code Quality Knowledge can be visualized with the same means as a structure in code. This is the structure of module extension.

```
:- module(cacq_module_extension, [module_/1, augmentable_predicate/2, augmentation/2,
    extending_module/2, extends/3, contains_augmentation/3, augments/3]).
:- use_module(cultivate_core('cqc'), []).
:- use_module('module_util', [clause_defined_in_module/2]).
:- use_module('module_structure', [own_module/1, own_predicate/2]).
```

```
cqc:anchor(module_).
module_(M) :- own_module(M), once(augmentable_predicate(M, _)).

cqc:role(augmentable_predicate).
augmentable_predicate(M, M:Functor/Arity) :-
    own_predicate(M, M:Functor/Arity),
    functor(Head, Functor, Arity),
    predicate_property(M:Head, (multifile)).

cqc:relation(contains_augmentation).
contains_augmentation(M, ExtendingModule, Augmentation) :-
    augmentable_predicate(M, M:Functor/Arity),
    functor(Head, Functor, Arity),
    clause(M:Head, _, Augmentation),
    clause_defined_in_module(Augmentation, ExtendingModule),
    M \= ExtendingModule.

/* [... simpler definitions for remaining roles and relation ...] */
```

Listing 1.4. The module extension structure based on reflective access to Prolog.

We further defined shines for modules and predicates being documented and a smell for importing predicates through the user module. For anchor, role and relation definitions we defined the necessary conditions to have the correct arity, having a unique name and being exported. Finally we decided that the structure needs to have at least one role.

7 Conclusion

Design concepts are often constructive concepts and premature formalizations may distort them thus leading to numbing false warnings. In the absence of a formal definition, the discussion of the extension and the intension of a concept become two complementary concerns. To localize constructive concepts we rely on the intended extension as expressed by the developer with program signs. We allow to define necessary and sufficient conditions as approximation of the potentially cloudy intension of the concept.

The extension of a concept may be given on the layer, where it is known. New concepts on a higher layer may be used to augment adaptation points in smell definitions. Amendments may as well be reserved for layers where the confidence is high enough.

Using annotated code samples as guidance for the implementation of Executable Code Quality Knowledge allows to grow it organically. The immediate feedback makes iterative and incremental development fun. The binding character of a desirable property may be increase step-by-step. Finally we illustrated how to define executable code quality knowledge about Executable Code Quality Knowledge.

References

1. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2018)
2. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
3. Geeraerts, D.: Theories of Lexical Semantics (2009). https://doi.org/10.1093/acprof:oso/9780198700302.001.0001, http://www.oxfordscholarship.com/view/10.1093/acprof:oso/9780198700302.001.0001/acprof-9780198700302
4. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and aspectJ. ACM SIGPLAN Notices **37**(11), 161–173 (nov 2002). https://doi.org/10.1145/583854.582436, http://portal.acm.org/citation.cfm?doid=583854.582436http://dl.acm.org/citation.cfm?id=582436
5. Johnson-Laird, P.N.: Human and machine thinking. In: John M. MacEachran Memorial Lecture Series, Oct, 1990, U Alberta, Edmonton, AB, Canada. Lawrence Erlbaum Associates, Inc (1993)
6. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, 1 edn. (2006)
7. Norvig, P.: Design Patterns in Dynamic Programming pp. 1–76 (1996), http://www.norvig.com/design-patterns/
8. Sharma, T., Spinellis, D.: A survey on software smells. Journal of Systems and Software **138**, 158–173 (2018). https://doi.org/10.1016/j.jss.2017.12.034, http://linkinghub.elsevier.com/retrieve/pii/S0164121217303114
9. Speicher, D.: Code Quality Cultivation. In: Fred, A., Dietz, J.L.G., Liu, K., Filipe, J. (eds.) Knowledge Discovery, Knowlege Engineering and Knowledge Management, Communications in Computer and Information Science, vol. 348, pp. 334–349. Springer Berlin Heidelberg (2013)
10. Speicher, D.: Polishing Design Flaw Definitions. 18. Workshop Software Reengineering und -Evolution (WSRE 2016), Bad Honnef, 2. - 4. May 2016, Softwaretechnik-Trends, Band 36, Heft 2 (2016)
11. Speicher, D., Appeltauer, M., Kniesel, G.: Code Analyses for Refactoring by Source Code Patterns and Logical Queries. In: Dig, D. (ed.) 1st Workshop on Refactoring Tools (WRT 2007), in conjunction with 21st European Conference on Object-Oriented Programming, July 30 - August 03, 2007. pp. 17–20. Berlin (2007), http://netfiles.uiuc.edu/dig/RefactoringWorkshop/
12. Speicher, D., Dong, T., Bauckhage, C., Cremers, A.B.: Growing Executable Code-Quality-Knowledge Organically. 20. Workshop Software Reengineering und -Evolution (WSRE 2018), Bad Honnef, 2. - 4. May 2018, Softwaretechnik-Trends, Band 38, Heft 2 pp. 61–62 (2018)
13. Taibi, T.: Design Pattern Formalization Techniques. IGI Global, Hershey, PA, USA (2007)
14. Tip, F., Kie\.zun, A., Bäumer, D.: Refactoring for generalization using type constraints. In: OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications. vol. 38, pp. 13–26. ACM Press, New York, NY, USA (nov 2003)

A type system for javascript

Andreas Stadelmeier

stan@hb.dhbw-stuttgart.de

1 Introduction

In this work I want to present two popular type systems for javascript: Flow and Typescript. These are two open source projects managed by Facebook (Flow) and Microsoft (Typescript).

The interesting part of those type systems is that they have been developed over 15 years after the javascript programming language itself. Therefore programmers have been using javascript for years without a static type check and the new type systems are now able to adapt to existing programming styles and frameworks.

The following chapters introduce Flow and Typescript and focus on their respective type inference algorithms.

2 Flow

Flow enables global type inference for javascript programs. It mostly set the focus on performance, refinements, precision and a good integration with javascript projects.

I will demonstrate the core functionality and idea behind Flows type inference algorithm with a simple example shown in Figure 2.

```
double(42);
double(null);

function double(param){
    console.log("Double of " + param);
    if(param != null){
        return param * 2;
    }else{
        return 0;
    }
}
```

Fig. 1. Flow example Line 5 spawns a type error: `Cannot add "Double of" and param because null is incompatible with string.`

The function `double` is called two times in this example. Once with a number and once with a `null` value. Flow builds a graph of type flows and recognizes that both types can reach the statement `console.log("Double of " + param)`. The `+`-Operator expects a `string` or a `number` for both its parameters. So when the `null` type reaches this operator Flow throws an error.

The if-branch in the next line triggers a type refinement by filtering out `null` values. Only the `number` value is able to reach the then-branch of the if-statement. That is the reason why Flow accepts the statement `param * 2` as correct.

So Flow does not really infer types, but just checks if there is a possible flow of types which could lead to a type error during runtime. Flow has many more features regarding refinements and invalidation of refined variables as described in [2]. But in the core it comes down to this type flows.

2.1 Type inference algorithm

The core of Flow's global type inference algorithm has a complexity of $O(n)$. But this good performance comes with a few downsides. The biggest one is, that Flow is not able to infer generic types. The following program is perfectly fine but will produce an error when checked by Flow.

```
function identity(x){
  return x;
}

var v: string = identity("A String");
var v3: number = identity(42);
```

Additionally there is neither method overloading nor use side variance allowed in Flow.

2.2 Conclusion

Flow makes some smart cuts in the powerness of its type inference algorithm, which enables Flow to process huge codebases in a very small timeframe. It tries to adapt to the way a javascript program is written and focuses on trying to find the most common mistakes with the least ammount of type annotations added by the programmer. The few false type errors can easily be fixed by adding type annotations.

The missing method overloading is not a heavy downside, because the idea behind Flow is to find errors in existing Javascipt projects and not to extend the language by new features. Overloading would be a new feature which isn't possible in standard javascript anyways.

3 Typescript

Typescript has a more class based approach to a typing system for javascript and provides a local type inference algorithm [1] comparable to those used by Java 11 or Scala. When a function definition is made in a typed context or a variable is instantiated at its declaration the type can be infered. For example:

```
let x = 3; //x: number is infered

//Typescript infers the type of the parameter mouseEvent:
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.button);   //<- OK
};
```

Especially Typescript could benifit from more type inference. Javascript is a dynamic language with very short edit-refresh cycle. Its dynamic type systems allows for quick adjustments to the code without adjusting all the types with it. Here a type inference algorithm could help by still statically checking the code but without having the user write down all of the type annotations himself.

4 Conclusion

Flow and Typescript bring some new interesting ideas in consideration of type inference and type systems to a broad audience. It's interesting to see how static types can be added to a dynamic programming language and how the users can benefit from this. Both are open source projects and heavily shaped onto the users needs.

Also both compilers come with an API which has the ability to extract an abstract syntax tree from javascript code. This gives scientists and programmers the possibility to add their own type system, type inference algorithm or linter ontop of the Flow or Typescript parser.

References

1. Typescript type inference description. https://github.com/Microsoft/TypeScript-Handbook/blob/master/pages/Type%20Inference.md, accessed: 2019-08-15
2. Chaudhuri, A., Vekris, P., Goldman, S., Roch, M., Levi, G.: Fast and precise type checking for javascript. CoRR **abs/1708.08021** (2017), http://arxiv.org/abs/1708.08021

Selbstspezialisierende Parserkombinatoren – Experimente mit praktischer partieller Auswertung in konventionellen Programmiersprachen

Baltasar Trancón y Widemann and Markus Lepper

semantics GmbH

Parserkombinatoren

Klassische Syntax-Werkzeuge des Compilerbaus sind in der Regel *generativ* und *monolithisch*: Aus einer Grammatik wird nach Analyse ein entsprechender Parser als abgeschlossenes Softwareartefakt erzeugt.

Als Alternative werden, besonders in deklarativen Sprachen, Parserkombinatoren verwendet. [2] Diese sind im Gegensatz *eingebettet* und *kompositional*: Jeder Ausdruck der Syntaxbeschreibung ist ein Programm der Implementierungssprache des Parsers, und selbst bereits ein vollständiger Parser, sprich ein Objekt welches seine eigene Interpretation implementiert. An die Stelle einer Grammatik tritt ein rekursives Gleichungssystem von Parsern, zu dessen Erzeugung alle Mechanismen der Umgebungssprache zur Modularisierung und Wiederverwendung verwendet werden können. Aus dieser Struktur folgt die natürliche Arbeitsweise eines kombinatorischen Parsers, nämlich im rekursiven Abstieg modular interpretiert. Diese Vorgehensweise bietet maximale Flexibilität bei der Konstruktion und Verwendung von Parsern, ist aber offensichtlich nicht besonders effizient.

Eine naheliegende Idee ist es, zur Verbesserung der Effizienz zwischen der Erzeugung des Parsers und seiner Verwendung eine Programmspezialisierung durchzuführen. [1] Kürzlich wurde praktisch gezeigt, dass dabei nicht nur die direkte Struktur des Parsers ausgenutzt werden kann, sondern auch eine induktive Form der Grammatik-Analyse stattfinden kann, deren Mächtigkeit annähernd LL(k) entspricht. [3] Allerdings setzt der Ansatz die Unterstützung von homoikonischer Metaprogrammierung in der Umgebungssprache voraus, was in den meisten konventionellen Programmiersprachen nicht gegeben ist. In der vorliegenden Arbeit wird ein alternativer Ansatz mit geringeren Anforderungen beschrieben.

Ramus

Das Projekt Ramus untersucht moderne Methoden der Automatisierung von Syntax-Werkzeugen. [6] Die Umgebungs- und Implementierungssprache der Werkzeuge ist Java. Ein zentraler Bestandteil ist eine Parserkombinator-Bibliothek, die sowohl zur (direkten manuellen und algorithmischen Meta-) Programmierung von kompositionalen Parsern, als auch als Zielsprache für die Generierung von monolithischen Parsern aus Grammatiken verwendet werden kann.

Der Entwurf der Ramus-Parserkombinatoren folgt, trotz der Einbettung in eine objektorientierte Umgebung, im Wesentlichen den Prinzipien funktionaler Programmierung. Parser vom Typ P sind dabei zustandsfreie Funktionsobjekte, welche in erster Näherung Folgen von Eingabesymbolen I auf Reaktionen, sprich Paare von Ausgabewerten O und unverarbeiteten Resteingaben abbilden. Nebeneffekte des Parsierens werden dadurch modelliert, dass die Parser Morphismen in der Kleisli-Kategorie einer Monade M sind:

$$P = I^* \to M(O \times I^*)$$

Im einfachsten Fall, der Modellierung von Nichtdeterminismus durch $M = _^*$, ergibt sich als Resultat ein Suchbaum. Ramus-Parser unterstützen neben dieser Form von Nichtdeterminismus auch die Annotation von Suchbäumen mit Diagnose-Nachrichten und die qualifizierte Zurückweisung einer Eingabe.

Die Bibliothek enthält eine Vielzahl der „üblichen“ atomaren Parser und Operatoren zur Konstruktion komplexerer Parser, so dass EBNF-artige Ausdrücke gebildet werden können. Weitere können vom Benutzer frei hinzugefügt werden, da die Schnittstelle als abstrakte Basisklasse `Parser` öffentlich und vererbbar ist.

Da die Umgebungssprache Java, im Gegensatz zur in [2] verwendeten Sprache Haskell, Ausdrücke strikt auswertet, können Grammatiken nicht einfach als rekursive Gleichungssysteme denotiert werden. Stattdessen steht aber ein universeller Fixpunktkombinator fix : $(P \to P) \to P$ zur Verfügung, mit dem rekursive Beziehungen explizit konstruiert werden können.

Neben der modularen Interpretation (der Anwendung des Funktionsobjektes) implementieren Ramus-Parserkombinatoren eine einfache induktive Syntaxanalyse, welche in etwa den FIRST-Mengen von LL(k) entspricht. Da die Rekursionsstruktur explizit durch Fixpunktkombinatoren gegeben ist, kann diese Information modular, vollautomatisch und für den Benutzer transparent zur Konstruktionszeit durch beschränkte Fixpunktiteration berechnet und zur lokalen Optimierung nichtdeterministischer Suche verwendet werden.

Spezialisierung

Programmspezialisierung zur Laufzeit ist traditionell die Domäne von homoikonischen mehrstufigen Metaprogrammiersprachen. In MetaOCaml, der Implementierungssprache von [3], gibt es beispielsweise zu dem Ausdruck `3 + 4`, welcher zu 7 auswertet, auch den Ausdruck `.<3 + 4>.`, welcher zu einem Codefragment auswertet, das gegebenenfalls die Addition von 3 und 4 durchführt. Codefragmente sind Bürger erster Klasse und werden im Typsystem sowie beim Pattern Matching strukturell konsistent unterstützt.

Die Sprache Java ist im Gegensatz dazu eine konventionell einstufige Sprache, bietet aber immerhin mit der `ClassLoader`-Schnittstelle die Möglichkeit der dynamischen Erweiterung der Codebasis des laufenden Programms in Form von Bytecode für die virtuelle Maschine JVM. Das Projekt LLJava [5] beinhaltet Modelle und Werkzeuge für die praktisch effiziente Konstruktion und Analyse von JVM-Bytecode. Das neue experimentelle Unterprojekt LLJava-live

unterstützt speziell die Erzeugung von Code mit Referenzen in das laufende Programm, mit dem Zweck der Ausführung in demselben. Damit wird eine ähnlich starke Ausdrucksfähigkeit (allerdings natürlich weniger konkret lesbar) für mehrstufige Metaprogrammierung wie in homoikonischen Sprachen erreicht. LLjava-live eignet sich besonders zur Spezialisierung von Programmen in einbetteten Kombinatorsprachen, wie kürzlich an der komplexen nichtdeterministischen Pattern-Matching-Sprache Paisley demonstriert. [4]

Konkret wird die Spezialisierung dadurch implementiert, dass jedes Objekt der Klasse `Parser`, erzeugt als Kombinatorausdruck, eine Methode `Parser compile()` besitzt, welche ein verhaltensgleiches Objekt von einer ad-hoc definierten und geladenen neuen Subklasse zurückliefert. Deren Code kann monolithisch den gesamten komplexen Kombinatorausdruck implementieren, und dabei Optimierungspotenziale nutzen, ohne dass die strenge Modularität der Definitionen dafür aufgegeben werden muss. Tatsächlich kann die Codeerzeugung exakt analog zur Interpretation modularisiert werden; mit der Erzeugung von Code zum Rücksprung in den Interpreter (der zweistufigen Eta-Expansion), existiert außerdem ein sicheres Default-Verhalten, so daß die Implementierung der Codeerzeugung insbesondere bei benutzerdefinierten Kombinatoren transparent optional ist.

Diese Strategie der dynamischen Codeerzeugung bedingt mehrere spezifische Herausforderungen: Zum einen müssen die durch Anwendung von Fixpunktkombinatoren erzeugten referenziellen Zyklen erkannt und in tatsächliche Rekursion übersetzt werden. Zum anderen besitzt die Java-Plattform mit ihrem just-in-time-Compiler bereits einen orthogonalen Mechanismus zur zweistufigen Codeerzeugung, dessen Auswirkung auf die praktische Effizienz im Zusammenhang mit dynamischer Programmspezialisierung sowohl theoretisch als auch empirisch interessant ist.

Literatur

[1] Y. Futamura. “Partial Evaluation of Computation Process— An Approach to a Compiler-Compiler. (Revised Reprint)”. In: *Higher-Order and Symbolic Computation* 12 (1999), S. 381–391. DOI: 10.1023/A:1010095604496.

[2] G. Hutton und E. Meijer. “Monadic Parsing in Haskell”. In: *J. Funct. Program.* 8.4 (1998), S. 437–444. DOI: 10.1017/S0956796898003050.

[3] N. Krishnaswami und J. Yallop. “A Typed, Algebraic Approach to Parsing”. In: *Proc. 40th PLDI*. In press. ACM, 2019. DOI: 10.1145/3314221.3314625.

[4] B. Trancón y Widemann und M. Lepper. “Improving the Performance of the Paisley Pattern-Matching EDSL by Staged Combinatorial Compilation”. In: *Proc. 27th WFLP*. Im Druck. 2019.

[5] B. Trancón y Widemann und M. Lepper. “LLJava: Minimalist Structured Programming on the Java Virtual Machine”. In: *Proc. 13th PPPJ*. ACM, 2016. DOI: 10.1145/2972206.2972218.

[6] B. Trancón y Widemann und M. Lepper. “Syntaxanalyse auf Wiedervorlage”. In: *Proc. 17. KPS*. Technischer Bericht 2014/02. Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg, 2013, S. 151–154.