Tagungsband zum 21. Kolloquium Programmiersprachen und Grundlagen der Programmierung

KPS 2021

Michael Hanus

Kai-Oliver Prott

Tagungsband zum 21. Kolloquium Programmiersprachen und Grundlagen der Programmierung

KPS 2021

Michael Hanus

Kai-Oliver Prott

Kiel Computer Science Series (KCSS) 2021/7 dated 2021-09-30

URN:NBN urn:nbn:de:gbv:8:1-zs-00000382-a5
ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

The author can be contacted via mh@informatik.uni-kiel.de

Published by the Department of Computer Science, Kiel University

https://www.informatik.uni-kiel.de/~mh/kps2021/

Please cite as:

▻ Michael Hanus, Kai-Oliver Prott *Tagungsband zum 21. Kolloquium Programmiersprachen und Grundlagen der Programmierung* Number 2021/7 in Kiel Computer Science Series. Department of Computer Science, 2021. Faculty of Engineering, Kiel University.

```
@proceedings{kps2021,
  editor    = {Hanus, Michael and Prott, Kai-Oliver},
  title     = {Tagungsband zum 21. Kolloquium Programmiersprachen und Grundlagen der Programmierung},
  publisher = {Department of Computer Science, Kiel University},
  year      = {2021},
  number    = {2021/7},
  isbn      = {},
  doi       = {10.21941/kcss/2021/7},
  address   = "Kiel, Germany",
  series    = {Kiel Computer Science Series},
}
```

© Michael Hanus, Kai-Oliver Prott and all authors of articles herein, 2021

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit http://www.informatik.uni-kiel.de/kcss for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

Inhaltsverzeichnis

Vorwort vii

Beiträge 1

Parallel Helper - Erkennung von Fehlermuster hinsichtlich Nebenläufigkeit, Asynchronität und Parallelisierung in C# — *Christoph Amrein, Luc Bläser* 2
Invariants and Correctness of Asynchronous Distributed Systems — *Annette Bieniusa* 3
Towards Robustness Testing of Functions Operating on Dynamic Data Structures — *Jan H. Boockman, Kerstin Jacob, Gerald Lüttgen* 5
Solver-Aided Verification of Declarative Programs - Non-Failure Condition Inference — *Niels Bunkenburg* 20
Towards Type-based Programm Synthesis for Software Product Lines — *Boris Düdder* 21
The Essence of Closures - A language design perspective — *M. Anton Ertl* 26
Weakest Preexpectation Reasoning for Probabilistic Concurrent Pointer Programs — *Ira Fesefeldt* 34
Can Logic Programming be Liberated from Backtracking? — *Michael Hanus* . . . 40
Meta Operatoren in MOSTflexiPL — *Christian Heinlein* 48
Effectful Effects and Contextual Effect Polymorphism — *Samuel Pilz* 55
Making a Monadic Curry - The Quest for a Complete and Fast(er) Compiler Implementation — *Kai-Oliver Prott, Finn Teegen* 56
Combinatory Differentiation: From Category Theory to High-Performance Automatic Differentiation — *Fritz Henglein* 71
STUBBER: Compiling Source Code into Bytecode without Dependencies for Java Code Clone Detection — *André Schäfer, Wolfram Amme,Thomas S. Heinze* 74

Verwendung von Klonerkennung zum Auffinden von Siguaturen von Malware-Familien: Eine Fallstudie über FinSpy — *Nils Scheidweiler, André Schäfer, Wolfram Amme,Thomas S. Heinze* 75
A formal type system and type checker for Erlang — *Albert Schimpf, Annette Bieniusa, Stefan Wehr* 76
Effect Systems in Haskell — *Hannes Siebenhandl* 77
Generalizing Java Type Unification - Adding bounded type variables to unification output — *Andreas Stadelmeier, Martin Plümicke* 82
Can Programming be Liberated From the Functional Style? - Für die Einführung des Plurals in die (objektorientierte) Programmierung — *Friedrich Steimann* . 99
Constraint-Logische Objektorientierte Programmierung mit Muli — *Hendrik Winkelmann* 115
Heterogene Übersetzung von echten Funktionstypen in Java-TX — *Etienne Zink, Martin Plümicke* 126

Vorwort

Das 21. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2021) setzt eine traditionelle Reihe von Arbeitstagungen fort, die 1980 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 Kolloquien finden seitdem in etwa zweijährlichem Rhythmus statt. Aus den ursprünglich drei Arbeitsgruppen sind in der Zwischenzeit weitere Forschungsgruppen in ganz Deutschland und darüberhinaus hervorgegangen. Heute präsentiert sich die Veranstaltung als ein offenes Forum für alle interessierten deutschsprachigen Wissenschaftlerinnen und Wissenschaftler. Es bietet einen zwanglosen Austausch neuer Ideen und Ergebnisse aus den Forschungsbereichen Entwurf und Implementierung von Programmiersprachen sowie Grundlagen und Methodik des Programmierens.

Die numehr über 40-jährige Tradition dieser Treffen wird sichtbar in der Liste der bisherigen Tagungsorte und veranstaltenden Institutionen:

2019	Baiersbronn	DHBW Stuttgart
2017	Weimar	Uni Jena
2015	Pörtschach am Wörthersee	TU Wien
2013	Lutherstadt Wittenberg	Uni Halle-Wittenberg
2011	Schloss Raesfeld, 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 21. Kolloquium dieser Reihe fand mit 35 Teilnehmenden vom 27. bis 29. September 2021 in Kiel statt und wurde von der Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion des Instituts für Informatik der Christian-Albrechts-Universität zu Kiel organisiert. Dieser Tagungsband enthält die wissenschaftlichen Beiträge, die bei diesem Kolloquium präsentiert wurden. Unser Dank gilt allen Autorinnen und Autoren für ihre Beiträge, die ein interessantes Spektrum der Forschung in dem Bereich der Programmiersprachen und Methodik der Programmierung abdecken.

September 2021

Michael Hanus
Kai-Oliver Prott

Beiträge

Parallel Helper

Erkennung von Fehlermuster hinsichtlich Nebenläufigkeit, Asynchronität und Parallelisierung in C#

Christoph Amrein
christoph.amrein@ost.ch
Ostschweizer Fachhochschule

Luc Bläser
luc.blaeser@ost.ch
Ostschweizer Fachhochschule

Abstract

Bug Patterns bezeichnen spezifische Fehlerkonstellationen in Programmcode, die meist lokal, einfach und schnell zu erkennen sind. Für den Aspekt der Nebenläufigkeit ist das Spektrum der Bug Patterns in heutigen Code-Analyse-Tools noch eher spärlich. Gerade für Programmiersprachen wie C# mit einer sehr breiten Feature-Parallelität hinsichtlich Nebenläufigkeit, Asynchronität und Parallelität ergeben sich etliche sprachspezifische Fallen, die sich als Bug Patterns gut fassen lassen.

Deshalb haben wir Parallel Helper entwickelt: Es ist eine statische Code-Analyse für C# mit einem umfangreichen Set von 57 Bug Patterns in der Nische von Concurrency, Asynchronität und Parallelität. Die Bug Patterns haben wir aus Praxisprojekten über einen Zeitraum von mehr als fünf Jahren gesammelt und bauen diese stetig aus. Als Roslyn-basierte Codeanalyse erkennt es die Fehlermuster auch in sehr grossen Code in Sekundenschnelle. Wir zeigen den aktuellen Stand des Projekts und geben einen Überblick über die implementierten Bug Patterns.

Invariants and Correctness of Asynchronous Distributed Systems

Annette Bieniusa
bieniusa@cs.uni-kl.de
TU Kaiserslautern

Collaborative applications such as shared documents, task lists, and calendars, are gaining more and more importance. Since they are central to organizing work and private life, users rely on their availability any place and any time. In particular, experiencing disconnectivity due to network issues has become a source of frustration.

The offline-first paradigm for designing highly-available applications poses interesting challenges for software engineers. Concurrent data accesses can no longer be directly synchronized and coordinated among different users and devices. The resulting challenges range from architectural decisions to make code and data available during offline periods, and extend to adaptations regarding the application semantics and user interface.

To prevent data corruption or loss, conflicting accesses must be identified and addressed accordingly, either at the data storage, the middleware, or at the application level. Though a number of distributed systems and frameworks provide corresponding technical solutions, there is no established process for distributed system engineers to choose the optimal synchronization and data replication strategy for a specific application.

In this talk, we will categorize different strategies for identifying and handling conflicting data accesses while preserving the users' intention. While domain-driven design guidelines are an informal, but highly relevant aid for developers (Braun et al., 2021), verification tools (Zeller, Bieniusa, and Poetzsch-Heffter, 2021; Kaki et al., 2018; Nair, Petri, and Shapiro, 2019; Balegas et al., 2018) approach the problem with formal rigor. We will further show how mechanisms for conflict resolution are reflected in different programming frameworks and programming language abstractions. Finally, we will discuss implications regarding composability and extensibility of applications and the interplay with other application aspects such as authentication and access control (Yanakieva et al., 2021).

References

Balegas, Valter et al. (2018). "IPA: invariant-preserving applications for weakly consistent replicated databases". In: *Proc. VLDB Endow.* 12.4, pp. 404–418. DOI: 10.14778/3297753.3297760. URL: http://www.vldb.org/pvldb/vol12/p404-balegas.pdf.

Braun, Susanne et al. (2021). "Tackling consistency-related design challenges of distributed data-intensive systems - an action research study". In: *CoRR* abs/2108.03758. arXiv: 2108.03758. URL: https://arxiv.org/abs/2108.03758.

Kaki, Gowtham et al. (Oct. 2018). "Safe replication through bounded concurrency verification". In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276534. URL: https://doi.org/10.1145/3276534.

Nair, Sreeja S., Gustavo Petri, and Marc Shapiro (2019). "Invariant safety for distributed applications". In: *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2019, Dresden, Germany, March 25-28, 2019*. ACM, 4:1–4:7. ISBN: 978-1-4503-6276-4. DOI: 10.1145/3301419.3323970. URL: https://doi.org/10.1145/3301419.3323970.

Yanakieva, Elena et al. (2021). "Access control conflict resolution in distributed file systems using crdts". In: *PaPoC@EuroSys 2021, 8th Workshop on Principles and Practice of Consistency for Distributed Data, Online Event, United Kingdom, April 26, 2021*. ACM, 1:1–1:3. DOI: 10.1145/3447865.3457970. URL: https://doi.org/10.1145/3447865.3457970.

Zeller, Peter, Annette Bieniusa, and Arnd Poetzsch-Heffter (2021). "Combining state- and event-based semantics to verify highly available applications". In: *Sci. Comput. Program.* 210, p. 102687. DOI: 10.1016/j.scico.2021.102687. URL: https://doi.org/10.1016/j.scico.2021.102687.

Towards Robustness Testing of Functions Operating on Dynamic Data Structures*

Jan H. Boockmann, Kerstin Jacob, and Gerald Lüttgen
{jan.boockmann, kerstin.jacob, gerald.luettgen}@swt-bamberg.de
Software Technologies Research Group, University of Bamberg, Germany

Abstract
This paper reports our ongoing research on a novel approach for the automatic generation of negative test inputs for the robustness testing of functions operating on complex dynamic data structures. A specification of a data structure in separation logic is used to first generate positive test inputs, i.e., data structure instances. A mutation operator is then applied to produce corrupt data structure instances as test inputs suited for robustness testing. Our approach is implemented in a prototype tool and evaluated on standard dynamic data structures. Preliminary results shown that few small test inputs suffice for achieving high code coverage when testing methods for data structure validation.

1 Introduction

Software testing activities typically focus on positive test cases to assess whether the system under test behaves as expected [Myers, 2004]. Negative test cases evaluating system robustness are often neglected. Here, robustness is understood as "the degree to which it[the system] functions correctly in the presence of exceptional inputs or stressful environmental conditions" [International Organization for Standardization, 2010]. Accordingly, testing activities that focus on validating the software wrt. exceptional conditions such as invalid inputs, are called robustness testing.

Robustness testing has become increasingly important, because modern systems frequently reuse existing software components, are distributed by design, and use isolated execution techniques such as Intel SGX [McKeen et al., 2013] to operate securely in an untrusted cloud environment. These aspects all have

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

in common that potentially corrupted data is received by the system from an untrusted source. Thus, assuring that a system correctly validates received data is increasingly important to ensure robustness.

A key challenge for successful robustness testing is obtaining meaningful negative test inputs, which cause a preceding input validation to report a failure. Generating negative test inputs is a time-consuming activity when done manually. For mature test suites the number of negative tests can surpass the number of positive tests by a ratio of 5:1 [Beizer, 1995]. However, manual test input generation prevails as the current practice in industry [Nardo et al., 2015]. This is particularly challenging and error-prone when the input validation under test does not operate on primitive data types. Automatic generation of test inputs is an active field in research. Recent work also deals with the generation of test inputs for functions operating on complex data types [Charreteur, Botella, and Gotlieb, 2009]. However, existing techniques mainly focus on the generation of positive test inputs and are thus not suitable for robustness testing.

This paper reports our ongoing research on an automated approach for generating negative test inputs to test the robustness of functions operating on complex dynamic data structures. The approach shall support software testers in the critical and error-prone task of finding test inputs for robustness testing.

Our approach is a black-box technique that utilizes a formal specification phrased in the notion of separation logic [Reynolds, 2002] describing the expected dynamic data structure. Separation logic is suitable for modelling the program heap and the therein contained data structures, because the separating conjunction operator stresses the disjointness of heap segments. Our approach generates instances of positive test inputs based on the formal specification and subsequently mutates these on the graph representation level to produce the invalid inputs required for robustness testing. As a result, our technique yields test inputs representing corrupt data structure instances as output. The approach is implemented in a prototype tool and evaluated on standard dynamic data structures. Preliminary results show shown that it generates relevant test inputs for assessing the quality of input validation methods, while maintaining a small test suite size.

Structure Section 2 reviews existing automatic test input generation techniques, and the following Section 3 explains the foundations of our approach. Section 4 illustrates and Section 5 evaluates our approach, while Section 6 presents our conclusion and directions for future work.

2 Related Work

The automatic generation of test inputs has been intensively studied; the works of Ince, 1987; Edvardsson, 1999; McMinn, 2004; Galler and Aichernig, 2014 survey the status quo at their respective time of publication. Early approaches focused on generating test inputs that contain only primitive data types, whereas recent approaches also consider complex data types, and especially dynamic data structures such as binary trees. For example, the prototype system INKA [Gotlieb, Botella, and Rueher, 2000] generates only primitive type test inputs, but has been extended by Charreteur, Botella, and Gotlieb, 2009 to dynamic data structures.

Automatic test input generation techniques can be classified by their method for finding test inputs into random, white-box and black-box approaches [Myers, 2004]. White-box approaches utilize the program source code to construct test inputs that force the control flow along predefined paths so as to achieve certain code coverage criteria. Representative tools include the aforementioned INKA system, the work of Visvanathan and Gupta, 2002 which generates the linkage pattern among data structure nodes independently of its data values, and the work of Degrave, Schrijvers, and Vanhoof, 2009 which deals with arbitrary pointer-based data structures. Black-box approaches utilize a provided specification to derive suitable test inputs. The specification languages of existing tools are primarily descriptive formal languages and executable programming languages. For example, Korat [Milicevic et al., 2007] and UDITA [Gligoric et al., 2010] require data structure validation methods written in Java, whereas Senni and Fioravanti, 2012 employ an equally expressive type grammar, De Angelis et al., 2019 use PropEr user-defined types to generate test inputs for Erlang programs, and MiMIs [Dewey, Hairapetian, and Gavrilov, 2020] utilize Scala class definitions.

The expressiveness of the language employed by the aforementioned declarative tools limits their test input generation to tree-like structures, thus excluding structures with backlinks, e.g., doubly-linked lists. Recent work of Pham et al., 2019 employs context-sensitive lazy initialization in combination with a separation logic [Reynolds, 2002] encoding to precisely characterize program heaps and, thereby, can express and generate test inputs containing backlinks. However, they construct only positive test inputs based on the program under test.

Tree ::= empty | Tree x Tree

$$\begin{aligned}\mathrm{btSL}(n) &\overset{\text{def}}{=} (\mathrm{emp} \wedge n = nil)\\ &\vee\ (\exists l, r. n \mapsto \langle l, r\rangle * \mathrm{btSL}(l) * \mathrm{btSL}(r))\end{aligned}$$

Figure 1. Definition of a binary tree structure in accordance with Senni and Fioravanti, 2012 using a type grammar (left) and separation logic (right).

3 Foundations

This section outlines the key concepts of separation logic and mutation testing that are relevant for the subsequent presentation of our approach.

Separation Logic Separation logic [Reynolds, 2002] has successfully been used in research and industry projects for the verification and analysis of software containing complex heap-based data structures. The logic simplifies the modeling of program heaps via its separating conjunction operator "$*$", which divides the program heap into disjoint heap segments. Inductive data types can be used to characterize dynamic data structures, including linked structures with backlinks.

Figure 1 depicts two binary tree structure definitions (excluding potential payload data). The definition on the left uses the type grammar of Senni and Fioravanti, 2012 to define that a tree is either empty, or contains two sub-trees. On the right, a binary tree is defined inductively using the recursive separation logic predicate "btSL". Here, the base case of an empty tree, i.e., an empty heap segment, is expressed by the predefined predicate "emp", and uses value "nil" to denote that argument n is a null pointer. In the inductive case, operator "$\mapsto$" in $n \mapsto \langle l, r\rangle$ indicates that argument n points to a valid memory region containing two pointers denoted by "$\langle l, r\rangle$". The predicate is subsequently invoked twice with l and r as argument, respectively. Note that the separating conjunction requires the memory region of the left sub-tree, right sub-tree, and the current node to be disjoint, and thus guarantees the desired tree structure.

A data structure specification may serve as an executable specification for functions performing input validation, where valid data structure instances are supposed to return "true" whereas invalid instances are supposed to return "false". In this context, the aforementioned type grammar does not lend itself as an executable specification, as it does not explicitly model the program heap. In contrast, separation logic is suitable for explicitly modelling the program heap and the contained data structures. Thus, it allows us to encode positive and negative test inputs and can be considered to be an executable specification.

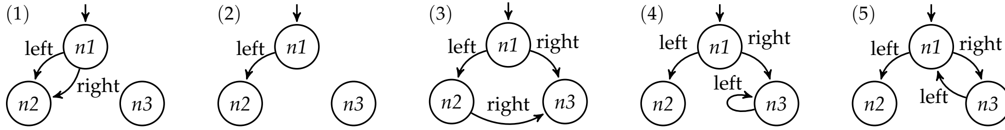


Figure 2. Five exemplary corrupt binary trees of size three that (1) yield the correct result, (2) incorrectly return two, (3) incorrectly return four, and (4–5) get stuck in an endless loop when provided as input to a naive counting function without a preceding input validation.

Mutation Testing Mutation testing [Jia and Harman, 2011] is a white-box testing technique to assess the quality of a test suite. Mutation operators introduce minor syntactic modifications in the program under test, and the test suite is then run against the mutated programs. If this run fails, the test suite has successfully detected the injected fault; the mutant is said to be killed. A mutant may remain undetected, either because the test suite is inadequate to detect the changed behavior introduced by the mutation, or because the mutant and the original program are functionally equivalent.

Our approach uses mutation operators to generate negative from positive test inputs so as to assess the quality of an input validation method. The separation logic specification, when understood as an executable specification, allows us to detect functionally equivalent mutants.

4 Approach

We propose a novel black-box technique for the automatic generation of dynamic data structures as test inputs. To the best of our knowledge, our technique is the first one that focuses on the generation of negative test inputs for the robustness testing of programs operating on dynamic data structures. As its input, our technique uses a formal specification phrased in separation logic and describing the expected data structure. As its output, our technique yields test inputs representing corrupt data structure instances. In contrast to related work in bounded-exhaustive testing [Coppit et al., 2005], we do not focus on the performance of generating test inputs, but instead aim to generate a small test suite of representative negative test inputs tailored to the data structure at hand.

Running Example As a running example, we consider a binary tree. Five exemplary corrupted binary trees, each having three nodes, are shown in Figure 2. The unlabeled edge without a source node indicates the root node, labeled edges

denote left and right pointers of a tree, and null pointers are not displayed. A naive node counting function without input validation may possibly get stuck in an endless loop if the input tree contains cycles, or may return an (in)correct result if tree nodes are unreachable or reachable via more than one path. Testing whether such invalid inputs are properly detected and reported is crucial for ensuring program robustness.

At program level, the detection and reporting of such invalid inputs is often implemented using input validation methods that check the data structure shape behind a pointer parameter and potentially further data constraints. Here, a shape describes the linkage pattern among data structure nodes, e.g., a graph may exhibit a binary tree shape.

Listing 1. `repOK` method from Korat [Milicevic et al., 2007] for validating a binary tree

```
1  public class BinaryTree {
2    public static class Node {public Node left; public Node right;}
3    public Node root; public int size;
4
5    public boolean repOK() {
6      if (root == null) {return size == 0;}
7      Set visited = new HashSet(); visited.add(root);
8      LinkedList workList = new LinkedList(); workList.add(root);
9      while (!workList.isEmpty()) {
10       Node current = (Node) workList.removeFirst();
11       if (current.left != null) {
12         if (!visited.add(current.left)) {return false;}
13         workList.add(current.left);}
14       if (current.right != null) {
15         if (!visited.add(current.right)) {return false;}
16         workList.add(current.right);}}
17     return (visited.size() == size);}
18 }
```

For example, Listing 1 shows the input validation method `repOK` of Milicevic et al., 2007 for detecting corrupt binary trees such as those presented in Figure 2. The method ensures that the number of reachable nodes matches the predefined size and that each node is reachable via a single path, thus also preventing cycles. Our approach generates the relevant inputs to test the error detection and reporting capabilities of such input validation methods wrt. to the shape of the underlying dynamic data structure.

Data Structure Specification As input, our approach requires a data structure specification in the form of an inductive separation logic predicate. We use Prolog to encode the structural constraint of the predicate and the notion of separated memory regions imposed by separation logic. In this context, we model the heap as a labeled graph, encoded in Prolog as a list of edges.

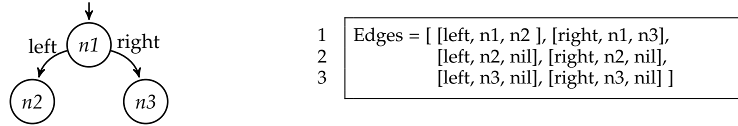


Figure 3. Exemplary binary tree (left) having three nodes and its corresponding Prolog encoding (right) as a list of labeled edges in format "[label, from, to]".

Figure 3 depicts an exemplary binary tree having three nodes and the list of edges encoding exactly this graph. We use "nil" to represent a null pointer value and explicit edge labels "left" and "right" to refer to both pointer field values.

Listing 2. Our Prolog encoding of a binary tree predicate following the notion of disjoint memory regions imposed by separation logic

```
1  btPL(nil, N, N, _).
2  btPL(T, N, N3, E) :− member(T, N), delete(N, T, N1),
3                       member([left, T, L], E), btPL(L, N1, N2, E),
4                       member([right, T, R], E), btPL(R, N2, N3, E).
```

Listing 2 depicts our Prolog encoding of a binary tree, which is semantically equivalent to the separation logic encoding presented in Figure 1. The two rules of the btPL predicate match the base and inductive case of a binary tree. The predicate btPL receives four arguments as input. The first argument models the current node, i.e., "nil" encoding a null pointer argument or "T" encoding a non-null pointer, and thus matches the argument n of btSL. The remaining arguments model the state of the execution environment. The second argument contains the initial consumable nodes. The third argument captures the consumable nodes remaining after evaluation, enabling us to detect if each node has been visited exactly once. The last argument is a list of edges representing the graph structure on which the predicate is evaluated. To ease readability, this state of the execution environment can also be encoded implicitly using a meta-interpreter [Boockmann and Lüttgen, 2019; Boockmann and Lüttgen, 2020].

Generation of Positive Test Inputs We use the backtracking search of the Prolog engine to generate candidate graphs of predefined size that satisfy the provided predicate. The sample query in Figure 4 derives graphs having exactly two nodes and satisfying predicate btPL. We require the invoked predicate (l. 7) to return an empty list of consumable nodes, thus the predicate consumes each node exactly once as stipulated by separation logic. The right of Figure 4 depicts the four binary trees obtained from evaluating the query on the left; these are all possible

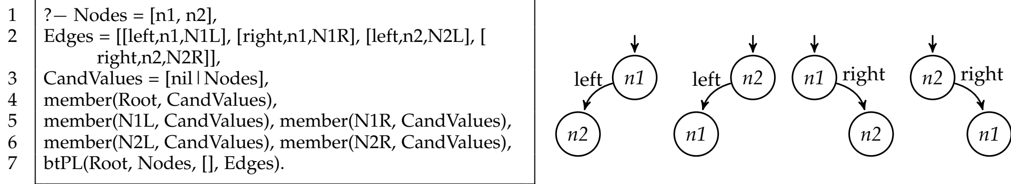


Figure 4. Left: Prolog query that characterizes the graphs that have exactly two nodes and satisfy the binary tree shape wrt. predicate btPL. Right: The graphs derived by this query.

binary trees that can be constructed using the two nodes *n1* and *n2*.

We aim to prevent the generation of isomorphic graphs, because they yield the same execution path in the program under test. Graph isomorphism splits the four generated binary trees in Figure 4 into two equivalence classes, because the first and last two graphs are isomorphic. Hence, we only keep one graph from each equivalence class, i.e., two graphs remain.

Table 1. Ten graphs obtained after exhaustively applying our single pointer field mutation operator to a candidate test input. New value and mutated pointer field combinations that do not change the input graph, are not displayed.

New value	Mutated pointer field				
	root	*n1.left*	*n1.right*	*n2.left*	*n2.right*
nil					
n1					
n2					

Generation of Negative Test Inputs via Data Structure Mutations Our approach employs mutation operators to generate negative from positive test inputs data structure mutations. In contrast to traditional mutation testing as described in Section 3, we aim to assess the quality of an input validation method. Thus, we mutate the positive test input, not the program. The input validation method under test has killed a mutated test input, if it reports "false".

The mutation operator must be chosen with care to ensure that a sufficiently but not unnecessarily large number of negative test inputs is generated. We propose the use of a single pointer field mutation, which exhaustively modifies each present pointer and sets it to any possible candidate value, i.e., nil and other present nodes. We believe that this operator suffices for the following reasons:

1. *Competent Programmer Hypothesis [Offutt, 1992]*: As programs containing bugs tend to be close to the correct version, simple changes simulate typical bug behavior.
2. *Coupling Effect Hypothesis [Offutt, 1992]:* Complex faults tend to arise from a combination of simple faults. Hence, a test suite that detects all simple faults also detects a complex fault with a high likelihood.
3. *Recursive Structure Hypothesis:* The data structures in scope for this work heavily rely on recursion to define their underlying shape. Thus, even minor deviations from the correct structure in a single region corrupt the overall shape. For example, a large binary tree with a single cycle is no longer a tree.

Accordingly, we assume that single mutations are sufficient to derive the relevant negative test inputs. This assumption is subject to evaluation in Section 5.

Table 1 presents the ten graphs obtained from exhaustively applying the proposed single mutation to the first graph in Figure 4. We use *root* to indicate the argument of btPL and *n1.left* to denote the left pointer field of node $n1$. Absent entries in the table indicate that this pointer field and value combination denotes the original graph provided as input. The number of thereby derived negative test inputs from a single positive graph with n nodes, a separation logic predicate arguments, and f fields equals $n \cdot (a + f \cdot n)$. Hence, our single positive test input yields $2 \cdot (1 + 2 \cdot 2) = 10$ negative test inputs.

Functionally Equivalent Mutated Graphs A key problem in traditional mutation testing is the existence of functionally equivalent mutants whose behavior is equivalent to the original non-mutated program. Thus, a test suite cannot kill such a mutant. An analogous problem arises for data structure mutations when the applied mutation operators do not corrupt the overall data structure.

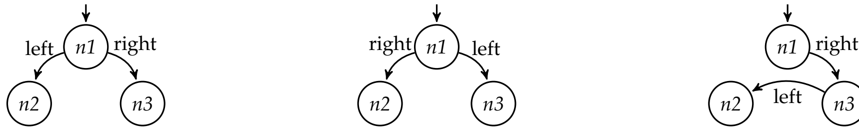


Figure 5. Three mutated binary tree data structures, which illustrate the archetypical cases of functional equivalence in the context of mutated graphs.

The problem of functionally equivalent data structure graphs quickly becomes apparent when considering multiple, e.g., two, mutations instead of a single mutation. Figure 5 depicts three graphs matching our binary tree predicate, which have been obtained using a double mutation from the graph shown on the left in Figure 3. These three examples illustrate the archetypical cases of functional equivalence in the context of mutated graphs:

1. The mutated graph is identical to the original graph, because two counter-active mutations were applied;
2. The mutated graph is not identical but isomorphic to the original graph;
3. The mutated graph is neither identical nor isomorphic to the original graph, but depicts a valid data structure instance wrt. the given predicate.

Note that the first of the aforementioned three archetypes is impossible when applying only a single mutation. Yet, the functional equivalence problem is already present when considering only single mutations: The second archetype of a non-identical but isomorphic graph is possible, e.g., when updating the entry pointer of a cyclic singly-linked list. The third archetype can occur if the data structure specification is not strict, e.g., if the specification of a binary tree would define a leaf as a node that either points to null or has a pointer to itself.

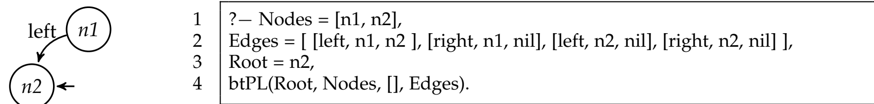


Figure 6. Mutated negative test input candidate (left) and the corresponding Prolog query (right) for checking whether the candidate is an instance of the binary tree predicate btPL.

Solving the traditional problem of identifying functionally equivalent mutants boils down to proving the functional equivalence of two programs, which is undecidable [Rice, 1953]. Showing functional equivalence of our mutated graphs

is a simpler problem, because it boils down to checking whether a mutated graph is still an instance of a particular data structure. Observe here that our Prolog predicates, e.g., btPL, cannot only be used for generating data structure instances, i.e., graphs, but also for testing whether a graph is an instance of a particular data structure. The Prolog query for doing so is analogous to Figure 4, but uses constants instead of variables for variable Edges and Root. Figure 6 depicts a mutated negative test input candidate on the left and the corresponding query for checking whether this test input is an instance of the binary tree data structure. Note that the query evaluates to false, i.e., the presented graph is not a binary tree, because not all nodes, i.e., node n1, are consumed during the evaluation. After filtering the candidate test inputs for functionally equivalent graphs, we filter out isomorphic graphs a second time in order to keep the number of negative test inputs small. For example, for the obtained negative sample test inputs shown in Figure 4, the graphs obtained from setting *root* to *n*2 and setting *n1.left* to *nil* are isomorphic, thus, only one of them is kept.

Table 2. The number of generated negative test inputs and their achieved instruction coverage (ins. cov.) and branch coverage (branch cov.) of the validation method, for each data structure combination. Validation methods for data structures binary tree (BT), singly-linked list (SLL) and circular doubly-linked list (CDLL) are taken from Korat [Milicevic et al., 2007]. We report results for three test input generation approaches: the exhaustive approach generates all possible graphs of a certain size that fail the validation method, the Korat approach generates all test inputs that fail the validation method, and our novel single mutation approach.

Data structure	Size (#nodes)	Exhaustive			Korat			Our Single Mutation		
		#negative test inputs	ins. cov.	branch cov.	#negative test inputs	ins. cov.	branch cov.	#negative test inputs	ins. cov.	branch cov.
BT	1	7	70%	56%	4	70%	56%	3	70%	56%
	2	239	94%	87%	48	94%	87%	15	94%	87%
	3	16 352	94%	87%	861	94%	87%	69	94%	87%
SLL	1	3	64%	50%	2	64%	50%	2	64%	50%
	2	25	91%	80%	5	91%	80%	5	91%	80%
	3	250	91%	80%	9	91%	80%	9	91%	80%
CDLL	1	7	56%	41%	4	56%	41%	3	56%	41%
	2	241	92%	83%	49	92%	83%	9	69%	58%
	3	16 378	92%	83%	864	92%	83%	11	69%	58%

5 Preliminary Evaluation

We have implemented our approach as a prototype tool using approx. 400 lines of TypeScript code. The source code of our prototype tool and the evaluation benchmark data are available online under an open source license at https://github.com/uniba-swt/KPS2021. The remainder of this section presents our three research questions (RQ), describes the chosen form of evaluation, and explains our results.

RQ1: Are small negative test inputs sufficient to achieve a high code coverage for data structure validation methods?

RQ2: Are the negative test inputs derived via single mutations sufficient to achieve a high code coverage for data structure validation methods?

RQ3: How does our approach compare to the Korat approach in terms of the amount of generated test inputs and achieved code coverage?

Our smallish evaluation benchmark includes standard dynamic data structures and validation methods from Korat [Milicevic et al., 2007]. We report the number of generated negative test inputs and the therewith achieved coverage for three approaches: An exhaustive approach, Korat, and our approach. The exhaustive approach, which generates all possible graphs of a certain size that fail the validation method, establishes a baseline for the maximum achievable number of test inputs and coverage. Table 2 shows the data obtained for this benchmark.

Regarding RQ1, Milicevic et al., 2007 showed in their evaluation of the Korat tool that small positive test inputs suffice to achieve a high coverage of functions operating on dynamic data structures. Analogously, we wish to show that small negative test inputs suffice to achieve a high code coverage of data structure validation methods. The obtained data shows that, with a size of 2, the test inputs of the exhaustive approach already achieve an instruction coverage of over 90%, and a branch coverage of at least 80%. Note that a code coverage of 100% cannot be achieved, as we only provide the input validation method with negative test inputs. Indeed, manual inspection confirms that these generated negative test inputs of size 2 only miss to visit lines/branches that lead to a positive result.

Regarding RQ2, we compare the coverage of our approach against the maximum achievable coverage. For the BT and SLL data structures, our approach yields the same coverage for sizes 1–3. This also holds for the CDLL with a single node, but our approach achieves a smaller coverage for 2–3 nodes. As manual inspection shows, our approach misses the test input, which resembles a correct CDLL with an incorrect number of nodes, i.e., at least one node is disconnected from the sub-graph reachable via the entry pointer. While similar test inputs

are indeed constructed for BT and SLL, our approach fails to do so for CDLL, because this would require a double instead of a single mutation.

Regarding RQ3, we focus on the number of generated test inputs, because the achieved coverage of Korat is equal to the exhaustive approach, which has been discussed in RQ2. Unsurprisingly, Korat and our single mutation approach generate less test inputs when compared to the exhaustive approach, because both filter out isomorphic inputs. However, our approach generates less or equal test inputs than Korat, because not necessarily all possible negative test inputs can be constructed using a single mutation. This difference is especially notable for BT with a size of 3, where the number of test inputs generated by Korat is one order of magnitude larger than our single mutation approach.

We plan to improve our preliminary evaluation by investigating how well our approach handles further data structures and different validation methods.

6 Conclusions

This paper presented ongoing research towards the robustness testing of functions operating on dynamic data structures. Our approach automatically generates negative test inputs for data structure validation methods from a data structure specification phrased in separation logic. A single pointer mutation operator generates negative from positive data structure instances obtained from the specification. A first preliminary evaluation of our prototypical implementation shows that, in most cases, single pointer mutations generate fewer test inputs than standard test input generation tools while achieving an equal coverage.

Future Work. We briefly discuss ideas for reducing the size of test suites. After test input generation, a manual inspection of the produced test can be employed to identify additional equivalence classes. Inspection can be enhanced by providing a suitable graph-based visualizer, similar to the Alloy visualizer [Jackson, 2006] but tailored to dynamic data structures. In addition, we propose to construct equivalence classes from the separation logic formulas of test inputs, where test inputs belong to the same class if they can be described by unifiable formulas.

During test input generation, semantic mutation operators introducing, e.g., cycles, or backlinks, could replace our syntactical single mutation. We further propose to derive test inputs from separation logic specifications directly, by identifying test inputs that cause the evaluation to fail at a particular Prolog term. Such an approach would use white-box test generation techniques at specification level, and thereby, behave as a black-box technique at program level.

References

Beizer, Boris (1995). *Black-box testing - techniques for functional testing of software and systems*. Wiley. ISBN: 978-0-471-12094-0.

Boockmann, Jan H. and Gerald Lüttgen (2019). "Shape inference from memory graphs (extended abstract)". In: *Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, pp. 69–71.

– (2020). "Learning data structure shapes from memory graphs". In: *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Vol. 73. EPiC Series in Computing. EasyChair, pp. 151–168. DOI: 10.29007/dhpw.

Charreteur, Florence, Bernard Botella, and Arnaud Gotlieb (2009). "Modelling dynamic memory management in constraint-based testing". In: *J. Syst. Softw.* 82.11, pp. 1755–1766. DOI: 10.1016/j.jss.2009.06.029.

Coppit, David et al. (2005). "Software assurance by bounded exhaustive testing". In: *IEEE Trans. Software Eng.* 31.4, pp. 328–339. DOI: 10.1109/TSE.2005.52. URL: https://doi.org/10.1109/TSE.2005.52.

De Angelis, Emanuele et al. (2019). "Property-based test case generators for free". In: *Tests and Proofs (TAP)*. Vol. 11823. LNCS. Springer, pp. 186–206. DOI: 10.1007/978-3-030-31157-5_12.

Degrave, François, Tom Schrijvers, and Wim Vanhoof (2009). "Towards a framework for constraint-based test case generation". In: *Logic-Based Program Synthesis and Transformation (LOPSTR)*. Vol. 6037. LNCS. Springer, pp. 128–142. DOI: 10.1007/978-3-642-12592-8_10.

Dewey, Kyle, Shant Hairapetian, and Miroslav Gavrilov (2020). "Mimis: simple, efficient, and fast bounded-exhaustive test case generators". In: *Software Testing, Validation and Verification (ICST)*. IEEE, pp. 51–62. DOI: 10.1109/ICST46399.2020.00016.

Edvardsson, Jon (1999). "A survey on automatic test data generation". In: *Second Conference on Computer Science and Engineering in Linköping*, pp. 21–28.

Galler, Stefan J. and Bernhard K. Aichernig (2014). "Survey on test data generation tools - an evaluation of white- and gray-box testing tools for C#, C++, Eiffel, and Java". In: *Int. J. Softw. Tools Technol. Transf.* 16.6, pp. 727–751. DOI: 10.1007/s10009-013-0272-3.

Gligoric, Milos et al. (2010). "Test generation through programming in UDITA". In: *Software Engineering (ICSE)*. ACM, pp. 225–234. DOI: 10.1145/1806799.1806835.

Gotlieb, Arnaud, Bernard Botella, and Michel Rueher (2000). "A CLP framework for computing structural test data". In: *Computational Logic (CL)*. Vol. 1861. LNCS. Springer, pp. 399–413. DOI: 10.1007/3-540-44957-4_27.

Ince, Darrel C. (1987). "The automatic generation of test data". In: *Comput. J.* 30.1, pp. 63–69. DOI: 10.1093/comjnl/30.1.63.

International Organization for Standardization (2010). "ISO/IEC/IEEE international standard - systems and software engineering – vocabulary". In: *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835.

Jackson, Daniel (2006). *Software abstractions - logic, language, and analysis*. MIT Press. ISBN: 978-0-262-10114-1.

Jia, Yue and Mark Harman (2011). "An analysis and survey of the development of mutation testing". In: *IEEE Trans. Software Eng.* 37.5, pp. 649–678. DOI: 10.1109/TSE.2010.62.

McKeen, Frank et al. (2013). "Innovative instructions and software model for isolated execution". In: *Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, p. 10. DOI: 10.1145/2487726.2488368.

McMinn, Phil (2004). "Search-based software test data generation: a survey". In: *Softw. Test. Verification Reliab.* 14.2, pp. 105–156. DOI: 10.1002/stvr.294.

Milicevic, Aleksandar et al. (2007). "Korat: A tool for generating structurally complex test inputs". In: *Software Engineering (ICSE)*. IEEE Computer Society, pp. 771–774. DOI: 10.1109/ICSE.2007.48.

Myers, Glenford J. (2004). *The art of software testing (2. ed.)* Wiley. ISBN: 978-0-471-46912-4. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471469122.html.

Nardo, Daniel Di et al. (2015). "Evolutionary robustness testing of data processing systems using models and data mutation (T)". In: *Automated Software Engineering (ASE)*. IEEE Computer Society, pp. 126–137. DOI: 10.1109/ASE.2015.13.

Offutt, A. Jefferson (1992). "Investigations of the software testing coupling effect". In: *ACM Trans. Softw. Eng. Methodol.* 1.1, pp. 5–20. DOI: 10.1145/125489.125473.

Pham, Long H. et al. (2019). "Enhancing symbolic execution of heap-based programs with separation logic for test input generation". In: *Automated Technology for Verification and Analysis (ATVA)*. Vol. 11781. LNCS. Springer, pp. 209–227. DOI: 10.1007/978-3-030-31784-3_12.

Reynolds, John C. (2002). "Separation logic: A logic for shared mutable data structures". In: *Logic in Computer Science (LICS)*. IEEE Computer Society, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

Rice, Henry Gordon (1953). "Classes of recursively enumerable sets and their decision problems". In: *American Mathematical Society* 74, pp. 358–366.

Senni, Valerio and Fabio Fioravanti (2012). "Generation of test data structures using constraint logic programming". In: *Tests and Proofs (TAP)*. Vol. 7305. LNCS. Springer, pp. 115–131. DOI: 10.1007/978-3-642-30473-6_10.

Visvanathan, Srinivas and Neelam Gupta (2002). "Generating test data for functions with pointer inputs". In: *Automated Software Engineering (ASE)*. IEEE Computer Society, p. 149. DOI: 10.1109/ASE.2002.1115007.

Solver-Aided Verification of Declarative Programs

Non-Failure Condition Inference

Niels Bunkenburg
nbu@informatik.uni-kiel.de
University of Kiel

Abstract
Anything that can go wrong will go wrong. Declarative languages avoid typical pitfalls related to lower level programming common in imperative programming languages. Nevertheless, declarative programs can go wrong, for example, when the type system does not reflect that an operation is defined partially. When we call a partial function with an unintended argument, program execution fails unless the failure can be handled. In order to specify under which conditions an operation cannot fail, we make use of *non-failure conditions* that check whether a function call fails for given arguments. Such conditions typically have to be written manually. For example, the *head* operation can return the first element of a list only if the list is non-empty. Since broad adoption of software verification practices hinges on simplicity and low overhead, automatically deriving such non-failure conditions where possible is a desirable goal. We build on existing work using SMT solvers for statically checking non-failure conditions in the functional-logic language Curry and present an approach for automatically deriving non-failure conditions not only for simple cases but even for more involved compound conditions.

Towards Type-based Program Synthesis for Software Product Lines

Boris Düdder
boris.d@di.ku.dk
Department of Computer Science
University of Copenhagen

1 Introduction

Modern software systems reuse existing source code from dependent frameworks and libraries, adding accidental complexity to today's software. The software supply chain for modern software products is complex and usually has a high fan-in of frameworks and libraries. Thus, these software products, on the other hand, are dependent on short-lived frameworks and libraries. The integration effort needs to be repeated for substituting deprecated frameworks and libraries, leading to a rapid value deprecation of software investments. This approach generates economic risks and, thus, is unsustainable for companies in the long run.

A significant amount of work for software developers is code and documentation navigation, comprehension, and connecting endpoints of software components. The early choice of programming languages for a project leads to an early lock-in into a development paradigm and supporting abstractions. Undoing such a choice later in the project is costly or even economically infeasible.

2 Background

The presented approach proposes a formal model of code composition and automatic code generation based on our tool called Combinatory Logic Synthesizer 3.0 (Bessai, 2019)[1] and provides a scalable model for software reuse (>6k lines). Furthermore, it supports code generation for multiple programming

[1]Project website under https://www.combinators.org

paradigms, i.e., object-oriented and functional, and various programming languages, e.g., Java, Python, Scala, necessary for modern software projects. The idea of meta-programming for program construction has been studied but still suffers from problems, e.g., in software evolution (expression problem or compositional evolution of software codebases, Bessai, Heineman, and Düdder, 2021).

Program synthesis and generation approaches are often focused on synthesizing syntactic programming language expressions (usually functional or procedural) from scratch and, then, lead to the problem of memory management, e.g., focusing on heap manipulation (Polikarpova and Sergey, 2019). Moreover, these approaches rarely support modern object-oriented languages and their environments (Austin et al., 2021), e.g., λ-functions, source code comments, or test cases.

In contrast, the composition-based synthesis and code generation approach focuses on software reuse of code or text templates with associated specifications. The code templates are usually represented as abstract syntax trees and allow for code manipulation and program transformation, e.g., creating or rearranging class bodies or methods.

A software product line (cf. K. Pohl, Böckle, and Der Linden, 2005) is a family of similar software systems from a shared set of software assets using a common means of production. Using a composition-based approach to generate software product line members from complex compositions of code fragments is based on a selection of features that are constraint by predefined selection logic, cf. Metzger and K. Pohl, 2014; Bessai, Düdder, et al., 2016b; Bessai, Düdder, et al., 2016a; Düdder, Heineman, and Rehof, 2016; R. Pohl et al., 2018. The management of these product variants is challenging (Metzger and K. Pohl, 2014), and software evolution increases these challenges.

A program synthesis approach, the combinatory logic synthesis (Rehof, 2013), based on program code components with logical specifications, allows to generate and regenerate program code in later stages, following the spirit of Davies and Pfenning, 2001, D'Antoni et al., 2021, and Düdder, Martens, and Rehof, 2014, and reduce the cost of early lock-ins. The logical specification is based on a Martin-Löf type discipline, allowing generated constructive proofs to be interpreted isomorphically as program transformers and blueprints for programs. These blueprints are well-typed functional programs in Scala or Haskell enumerating solution programs in object-oriented languages, such as Java, C#, Python, and Rust, or functional languages.

In a series of papers (Rehof and Urzyczyn, 2011; Rehof and Urzyczyn, 2012; Düdder, Martens, Rehof, and Urzyczyn, 2012; Düdder, Martens, and Rehof, 2013; Bessai, Chen, et al., 2018), we have laid the theoretical foundations for

understanding algorithmic and complexity of decidable inhabitation in subsystems of the intersection type system by Coppo and Dezani-Ciancaglini, 1980. In contrast to standard combinatory logic, where a fixed basis of combinators is usually considered, the inhabitation problem considered here is *relativized* to an arbitrary environment given as part of the input. This problem is undecidable for combinatory logic, even in simple types, cf. Düdder, Martens, Rehof, and Urzyczyn, 2012. We have introduced finite and bounded combinatory logic with intersection types and an extension with modal types as a possible foundation for type-based composition synthesis. The type environment represents the existing code fragments with a specification based on intersection types. The inhabitation goal is interpreted as a synthesis goal, and the resulting combinatorial term is the resulting blueprint program for the generation of the target code.

3 Talk

The talk will introduce type-based program synthesis for software systems. It will highlight the aspects of software evolution, some hindrances of object-oriented programming languages, and a technique for variability management in software product line engineering.

References

Austin, Jacob et al. (2021). "Program synthesis with large language models". In: *arXiv preprint arXiv:2108.07732*.

Bessai, Jan (2019). "A type-theoretic framework for software component synthesis." PhD thesis. Technical University of Dortmund, Germany.

Bessai, Jan, TzuChun Chen, et al. (Feb. 2018). "Mixin Composition Synthesis based on Intersection Types". In: *Logical Methods in Computer Science (LMCS)* Volume 14, Issue 1. DOI: 10.23638/LMCS-14(1:18)2018. URL: https://lmcs.episciences.org/4319.

Bessai, Jan, Boris Düdder, et al. (2016a). "A Long and Winding Road Towards Modular Synthesis". In: *Proceedings of 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9952. Springer, pp. 305–322.

– (2016b). "Combinatory Synthesis of Classes using Feature Grammars". In: *Proceedings of the 12th International Conference on Formal Aspects of Component Software (FACS)*. Ed. by C. Braga and P.C. Olveczky. Vol. 9539. Springer, pp. 1–18. DOI: 10.1007/978-3-319-28934-2_7.

Bessai, Jan, George T. Heineman, and Boris Düdder (2021). "Covariant Conversions (CoCo): A Design Pattern for Type-Safe Modular Software Evolution in Object-Oriented Systems ". In: *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP)*. Ed. by Manu Sridharan. Vol. 194. 5. Schloss Dagstuhl. DOI: 10.4230/LIPIcs.ECOOP.2021.5.

Coppo, Mario and Mariangiola Dezani-Ciancaglini (1980). "An Extension of Basic Functionality Theory for Lambda-Calculus". In: *Notre Dame Journal of Formal Logic* 21, pp. 685–693.

D'Antoni, Loris et al. (2021). "Programmable program synthesis". In: *International Conference on Computer Aided Verification*. Springer, pp. 84–109.

Davies, Rowan and Frank Pfenning (2001). "A Modal Analysis of Staged Computation". In: *Journal of the ACM* 48.3, pp. 555–604.

Düdder, Boris, George T. Heineman, and Jakob Rehof (2016). "ModSyn-PP: Modular Synthesis of Programs and Processes (Track Introduction)". In: *Proceedings of 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9952. Springer, pp. 263–267.

Düdder, Boris, Moritz Martens, and Jakob Rehof (2013). "Intersection Type Matching with Subtyping". In: *Proceedings of TLCA'13*. Vol. 7941. Lecture Notes in Computer Science. Springer, pp. 125–139.

– (2014). "Staged Composition Synthesis". In: *Proceedings of 23rd European Symposium on Programming (ESOP)*. Vol. 8410. LNCS. Springer, pp. 67–86. DOI: 10.1007/978-3-642-54833-8_5. URL: http://dx.doi.org/10.1007/978-3-642-54833-8_5.

Düdder, Boris, Moritz Martens, Jakob Rehof, and Paweł Urzyczyn (2012). "Bounded Combinatory Logic". In: *Proceedings of CSL'12*. Vol. 16. LIPIcs. Schloss Dagstuhl, pp. 243–258.

Metzger, Andreas and Klaus Pohl (2014). "Software product line engineering and variability management: achievements and challenges". In: *Future of Software Engineering Proceedings*. FOSE 2014. Hyderabad, India: Association for Computing Machinery, pp. 70–84. ISBN: 9781450328654. DOI: 10.1145/2593882.2593888. URL: https://doi.org/10.1145/2593882.2593888.

Pohl, Klaus, Günter Böckle, and Frank J van Der Linden (2005). *Software Product Line Engineering – Foundations, Principles, and Techniques*. Springer.

Pohl, Richard et al. (2018). "Variant management solution for large scale software product lines". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '18. Gothenburg, Sweden: Association for Computing Machinery, pp. 85–94. ISBN: 9781450356596. DOI: 10.1145/3183519.3183523. URL: https://doi.org/10.1145/3183519.3183523.

Polikarpova, Nadia and Ilya Sergey (2019). “Structuring the synthesis of heap-manipulating programs”. In: *Proceedings of the ACM on Programming Languages* 3.POPL, pp. 1–30.

Rehof, Jakob (Jan. 2013). “Towards Combinatory Logic Synthesis”. In: *BEAT'13, 1st International Workshop on Behavioural Types*. ACM.

Rehof, Jakob and Paweł Urzyczyn (2011). “Finite Combinatory Logic with Intersection Types”. In: *Proceedings of TLCA'11*. Vol. 6690. Lecture Notes in Computer Science. Springer, pp. 169–183.

– (2012). “The Complexity of Inhabitation with Explicit Intersection”. In: *Kozen Festschrift*. Vol. 7230. Lecture Notes in Computer Science. Springer, pp. 256–270.

The Essence of Closures

A language design perspective

M. Anton Ertl
anton@mips.complang.tuwien.ac.at
TU Wien

Abstract
Closures are originally associated with lexically scoped name binding. However, in the course of implementing closures in Gforth, it turned out that the actual function (the essence) of closures is to communicate data between closure creation and the closure execution (with the closure call usually being far from the closure creation). This paper presents a simple language extension for C: two-stage parameter passing, implemented with flat closures; the first stage creates a closure, the second stage calls it. Nested functions and access to outer locals are not needed.

1 Introduction

In the summer of 2018 I worked with Bernd Paysan on adding "closures" to Gforth [Ertl and Paysan, 2018]. At the time Gforth had local variables and nested definitions (known as quotations), but outer local variables were (and are) not visible inside quotations, in order to avoid the implementation complications of closures. But we wanted to allow programmers to do in Gforth what they do with closures in other languages.

I started out with the lexical scoping idea with some adaptions to avoid garbage collection and with explicit capture of outer locals for simpler implementation in terms of flat closures, and wrote a draft version of a paper about that [Ertl, 2018]. Then Bernd Paysan set out to implement these ideas, came up with further simplifications, and after a lot of back-and-forth we arrived at the final design and implementation. As a result, the paper had to be almost completely rewritten [Ertl and Paysan, 2018].

One interesting aspect is that the result is not just easier to implement, but also often more convenient to use than the original lexical-scoping-inspired syntax. As an example, here is a definition of `+field` using the draft syntax:

```
: +field ( u1 u "name" -- u2 )
    create over {: u1 :} +
    ['] alloth <[{: : u1 :} drop u1 + ;] set-does> ;
```

and the syntax of the final paper:

```
: +field ( u1 u "name" -- u2 )
    create
    over [{: u1 :}d drop u1 + ;] set-does>
    + ;
```

I will not explain the syntax in detail here, but just explain the decisive difference: In the draft version is was necessary to create a local `u1` with `{: u1 :}` that could then be passed (explicitly) into the closure (with `: u1`), while in the final version the value is passed as a parameter at closure creation and only turned into a local in the closure (`[{: u1 :}d`). Since then we have added a variant that does not require defining a local and expresses the same functionality as:

```
: +field ( u1 u "name" -- u2 )
    create
    over [n:d nip + ;] set-does>
    + ;
```

In this variant closure creation takes one item from the stack and stores it in the closure, and the closure call pushes this item.

Of course, these examples and the examples in the papers are hard to understand for a non-Forth audience, so I wrote this paper with a wider audience in mind, and use C with appropriate (unimplemented) extensions in the rest of this paper.

The main point of this paper is that it can be (and, in our experience, is) better for the implementation complexity, for the language, and for the programmers to provide what programmers want to do with closures more directly than by accessing outer locals in nested functions.

This paper may be of little interest to well-versed implementors of functional languages, but it may inspire language designers of other languages to widen their perspective, looking for other solutions than just implementing nested functions with access to outer locals. It may help them to find a better solution, without a detour as long as the one I have taken.

Section 2 gives an example where closures are useful. Section 3 introduces functions with two-stage parameter passing as a language-level way to build flat

closures directly. Section 4 shows how to implement the equivalent of writing to an outer local variable. Section 5 discusses how to manage the memory needed for closures. Finally, Section 6 describes related work.

A note on terminology: The term *closure* has been originally used for a lambda expression with (not really) free variables where these variables are bound by the lexically-scoped environment, then by language implementors for the data structure that implements this concept, and by programmers when they create such a thing, pass it around, or call it. It has seen quite a bit of expansion in meaning in that process, and this paper continues with that expansion, by using *closure* to mean any function/procedure/method that has a parameter that is determined at run-time[1], but is not in the parameter list of calls to the closure.

2 Motivation

When a function g takes another function f as parameter, g typically calls f with a certain number of arguments, and normally the argument is generated by f according to its internal logic. A classical example is a numeric integration function, which in C would go something like this:

```
double numint(double l, double h, double (*f)(double))
{
  ... (*f)(x);
  ...
  return result;
}
```

`numint()` calls `(*f)(x)` repeatedly with different x in order to approximate $\int_l^h f(x)dx$. Now consider the case that we want to compute $\int_l^h x^{-y}dx$, where y is a run-time parameter that is not changed inside `numint`. We would have to implement

```
double numint2(double l, double h,
             double (*f)(double,double), double y)
{
  ... (*f)(x,y);
  ...
  return result;
}
```

[1] This excludes passing through a global variable, which is bound at compile time, and makes the whole thing non-reentrant.

And additional versions of numint would be necessary for passing more parameters or parameters with other types.

Instead, we would like to bind y to a function at run-time and pass a function pointer to `numint()` that can be called with only an x argument. Closures provide this capability.

3 Two-stage parameter passing

A way to define such a closure is to pass parameters in two stages: In the first stage we pass y, creating the closure; we can then pass the closure to `numint()`, where it is called (repeatedly) with various x parameters. A C syntax extension for this might look as follows:

```
double g(double y)(double x) {
  return pow(x,-y);
}

/* inside a different function */
  r1 = numint(a, b, g(2.0));
  r2 = numint(a, b, g(3.0));
  r3 = numint(a, b, g(2.5));
```

The first `numint()` call contains a call to `g()`, which is evaluated first, and produces a closure where $y = 2.0$. Inside `numint()` the closure is then called with various values for x.

One benefit of this approach is that we can pass a value directly instead of having to assign it to an outer local first. Admittedly, with outer locals a function `g()` can be constructed that is called in a very similar way, but that function would pass a closure outwards (which requires explicit deallocation in a language like C), whereas at least in the `numint()` case stack allocation is sufficient for a closure created by the first-stage call to `g()`.

Implementationwise, the closure consists of the code address of `g` and the value of y. This kind of representation comes out of flat-closure-conversion [Dybvig, 1987], one possible way to implement a language with nested functions with access to outer locals.[2] So what we do here is to write the code such that flat closures are created directly.

[2] The mechanical conversion from nested functions to flat closures also shows that two-stage parameter passing is as powerful as nested functions.

This means that we cannot directly add a third stage, but (if we need that) have to do it by having a two-stage function that calls another two-stage function and passes the arguments of the first two stages to the second function as arguments:

```
int f1(int i1, int i2)(int i3) {
  return i1*i2*i3;
}
int(*)(int) f2(int i1)(int i2) {
  return f1(i1,i2);
}
```

Of course, if it turns out that this kind of extension is used a lot in your language, it could be extended into something more sophisticated, along the lines of what functional programming languages like ML do with currying and a more sophisticated compiler.

4 Changeable data

Until now these closures only allow passing data by value. What if we want to have read and write access to shared data? Well, the C way is to pass a pointer to the data. A simple example of that is:

```
int counter(int *p)(void) {
  return (*p)++;
}

/* inside a different function */
  int *c1p = malloc(sizeof(int));
  int *c2p = malloc(sizeof(int));
  *c1p = 5; /* initialize counters */
  *c2p = 0;
  int (*c1)() = counter(c1p);
  int (*c2)() = counter(c2p);
  int (*c1a)() = counter(c1p);
  printf("%d %d\n",(*c1)(),(*c2)()); // 5 0
  printf("%d\n",(*c1a)());           // 6
  printf("%d %d\n",(*c1)(),(*c2)()); // 7 1
```

In this example memory for two counters is created with `malloc()`, and two closures (`c1` and `c1a`) access one of the counters, while a third closure (`c2`) accesses

the other one. Each call to one of the closures increases the corresponding counter and returns its previous value.

When implementing languages with nested functions with access to outer locals and flat-closure conversion, this kind of code comes out of assignment conversion [Dybvig, 1987]. Again, here we write this code directly.

Explicitly allocating the memory for the counters is a bit cumbersome, and depending on the language you are designing, you may or may not want to provide a more convenient way to write this.

Moreover, your base language may not allow to pass pointers to memory around or another way may be preferable. E.g., in C++ it would be more convenient to use references rather than pointers to the memory. In Pascal, Modula-2, or Oberon I would probably use the `VAR` parameter syntax (and its pass-by-reference semantics) for this purpose.

5 Closure memory

Closures need memory at run-time, at least for storing (in our example) y, possibly also a pointer to the code for computing x^{-y}, and possibly some native code (called a trampoline) so that the closure can be involved like a regular function pointer.

In many languages closures are garbage-collected; others, like Algol 60, Pascal, and C++ only allow calling closures in a way that allows them to be stack-allocated.[3] In both cases the programmer does not have to deal with allocation and deallocation of the memory.

C (and Forth) does not have garbage collection, but instead uses either stack allocation and deallocation or `malloc()` and `free()` (or user-constructed memory management, such as region-based allocation or reference-counting). We could require the same restrictions as C++ to get by with stack allocation, but that is insufficient in many usage cases.

Therefore, closure memory is managed by the programmer, just like other memory. The programmer decides whether closures are allocated on the stack, or with which allocator they are allocated and destroyed.

In Gforth we have the general closure constructor that takes the allocator as parameter, and also shorthands for the common cases: stack, heap (`allocate` (like `malloc()`) and `free`), and dictionary (essentially allocation until the end of the process). I leave it to you to find a nice syntax for C or the language you are designing.

[3]One usually does not call closures in languages with this restriction “closures”, but here I do.

6 Related work

Our earlier work [Ertl and Paysan, 2018] describes a similar Forth extension in more detail, including performance results, and implementation data. At the time of that paper the closure implementation in Gforth cost 78 source lines of code (it has grown a little since then).

A number of other languages provide similar features, some through nested functions with access to outer locals, others with more special syntax. Of particular note are the C++11 lambdas[4], which inspired my first (too-complicated) approach: They are based on accessing outer locals, but allow capturing the outer locals explicitly, and allow to control whether a variable is captured by-value or by-reference. One major difference is that we need to assign the value to the outer local first rather than passing it as parameter as in the present approach. Interestingly, C++ does not offer explicit memory management of closures, and restricts its programmers to stack-related limitations.

Our implementation ideas were based on flat-closure conversion approach [Dybvig, 1987, Section 4.4] in combination with assignment conversion [Dybvig, 1987, Section 4.5]. A later work [Keep, Hearn, and Dybvig, 2012] discusses the kinds of optimizations that implementors of nested functions should perform. But the basic implementation ideas inspired the language side of the Gforth extension, and eventually the present paper, and there such optimizations are unnecessary (or left to the programmer).

However, there are also other ways to implement access to outer locals. Static link chains and the display [Fischer and LeBlanc, 1988] are particularly well-known. They keep each local in only one place, but have relatively complex and sometimes slow ways to access them.

7 Conclusion

It is not necessary to implement nested function with access to outer locals in order to provide the features to programmers that they want. Instead, functions with two-stage parameter passing allow a direct implementation in terms of flat closures, and this can even be nicer to use than accessing outer locals.

[4] https://en.wikipedia.org/wiki/Anonymous_function#C++_%28since_C++11%29

References

Dybvig, R. Kent (Apr. 1987). "Three implementation models for scheme". PhD thesis. University of North Carolina at Chapel Hill. URL: http://agl.cs.unm.edu/~williams/cs491/three-imp.pdf.

Ertl, M. Anton (2018). "General locals". Draft version of the published paper [Ertl and Paysan, 2018] that was mostly rewritten. URL: http://www.euroforth.org/ef18/drafts/ertl.pdf.

Ertl, M. Anton and Bernd Paysan (2018). "Closures — the Forth way". In: *34th EuroForth Conference*, pp. 17–30. URL: http://www.complang.tuwien.ac.at/papers/ertl%26paysan.pdf.

Fischer, Charles N. and Richard J. LeBlanc (1988). *Crafting a compiler*. Menlo Park, CA: Benjamin/Cummings.

Keep, Andrew W., Alex Hearn, and R. Kent Dybvig (2012). "Optimizing closures in O(0) time". In: *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*. Ed. by Olivier Danvy. ACM, pp. 30–35. ISBN: 978-1-4503-1895-2. URL: http://doi.acm.org/10.1145/2661103.

Weakest Preexpectation Reasoning for Probabilistic Concurrent Pointer Programs

Ira Fesefeldt
fesefeldt@cs.rwth-aachen.de
RWTH Aachen

Abstract
We present here ongoing research in order to extend a probabilistic variant of weakest precondition, the weakest preexpectation, to deal with concurrency. The idea is to merge concurrent separation logic and quantitative separation logic. While both handle heap-manipulating programs, the former handles concurrency, the later handles probabilities. Combining these gives raise to proof rules for programs supporting both randomization and concurrency.

1 Introduction

Formal verification for all three, probabilistic, heap-manipulating and concurrent programs is difficult. One useful tool to deal with heap-manipulating programs and for concurrent programs is separation logic [Ishtiaq and O'Hearn, 2001]. Separation logic extends classical first order logic by the separating conjunction operator that splits a heap in two partitions with a conjunction. Separation logic is thus a well behaving tool to reason about separate parts of the heap - something that often is required to reason about local memory. However, it also proved to be useful in order to give a notion of heap ownership. In concurrent programs, heaps are separated in several parts and the verificator then assigns every thread ownership of a part of the heap. This notion of disjoint concurrent separation logic could then be extended by having one part of shared ownership. Concurrent separation logic now imposes the additional requirement that the shared heap needs to be invariant [Vafeiadis, 2011].

On the probabilistic part, we use the weakest liberal preexpectation [Morgan, McIver, and Seidel, 1996]. Weakest liberal preexpectation[1] is an effort to extend

[1]Usually the non-liberal version is used. However, for this work only the liberal version seems to be suitable.

Dijkstra's weakest liberal preconditions [Dijkstra, 1968] to probabilistic programs by extending the domain of predicates (i.e. functions from states to boolean) to the domain of random variables (i.e. functions from states to reals between 0 and 1), which we usually call expectations. Weakest liberal preexpectation could then be used to prove that a given postcondition has a certain probability.

These two domains, separation logic and expectations, can, however, be joined into a new domain, which we call quantitative separation logic [Batz et al., 2019]. Quantitative separation logic is defined as a map from stack heap pairs to reals, using several operations that are well known from classic separation logic.

In the following, we will first present quantitative separation logic and afterwards present ongoing research on how to extend it to concurrency and end with an example where we reason about a consumer-producer having a lossy channel using these new proof rules.

2 Concurrent Quantitative Separation Logic

We define (concurrent) quantitative separation logic without defining an exact semantics. Instead, we define it in terms of a domain, a set of operations and proof rules for programs. In this way, quantitative separation logic, or short QSL, is just the set of all functions that map stacks (i.e. variable assignments) and heaps (maps from natural numbers to integers) to reals between 0 and 1.

2.1 Definition (Quantitative Separation Logic [Batz et al., 2019])**.** Let *Var* be a set of variables, then:

$$Stack = Var \to \mathbb{Z}$$
$$Heap = \mathbb{N} \to \mathbb{Z}$$
$$SL = Stack \times Heap \to \{true, false\}$$
$$QSL = Stack \times Heap \to [0,1]$$

We usually require heaps to be only defined on a finite part on their domain. The function $dom(h) \subseteq \mathbb{N}$ denotes the defined part of the domain of a heap h. Furthermore, $f \uplus g$ is the combination of functions f and g, which is only defined if their domains are disjoint. In order to transform classic (separation) logic into a quantitative version, we use Iverson brackets [Iverson, 1962]. For these, we write $[P]$ to express the function that maps (s,h) to 0 if $s,h \not\models P$ and 1 if $s,h \models P$. Moreover, we use the predicates emp to check for the heap with empty domain, $E \to E'$ for the heap that assigns exactly $E(s)$ to the value $E'(s)$ and $E \to _$ for the heap with domain $\{E(s)\}$.

To reason about the separation of heap, we use separating conjunctions in classic separation logic. In the quantitative version, we need to lift the conjunction of this separation operator to multiplication.

2.2 Definition (Quantitative Separating Conjunction [Batz et al., 2019])**.** For $f, g \in QSL$ we have the separating conjunction

$$(f * g)(s, h) = \sup\{f(s, h_1) \cdot g(s, h_2) \mid h_1 \uplus h_2 = h\}.$$

Using these ingredients, we can define a weakest liberal preexpectation (short wlp) for $f \in QSL$. We call the weakest liberal preexpectation the expectation $\mathbb{E}_C(f)$ to realize f after finishing the program C or to not terminate C, written as $wlp[C](f)$. If $f = [P]$ is a predicate, then the expected value of realizing P after finishing the program is the probability (including side conditions) to realize this predicate in a final state.

It is easy to define rules to allow us reasoning about disjoint concurrent programs, i.e. programs that have threads with disjoint variable sets. However, most concurrent programs - or at least most of the ones that we usually want to reason about - are not disjoint. To reason about these, we use a resource invariant inspired by Vafeiadis [Vafeiadis, 2011] in order to assign a part of the heap global ownership. To reason about the resource invariant, we use a safe variant of the weakest liberal preexpctation, or short $wslp$, that guarantees that the resource invariant is indeed invariant and underapproximates wlp. This shared memory can only be changed in atomic regions. For the resulting semantics (which we will not lay out in detail), new rules arise that are similar to the ones from Vafeiadis:

2.3 Definition (Proof Rules for Quantitative Concurrency)**.** Let $f, g \in QSL$, $P, RI \in SL$, where RI is the resource invariant, C, C_1, C_2 programs and $C_1 || C_2$ their concurrent composition. We have

$$wlp[C](f) = wslp[C](f \mid emp). \tag{2.4}$$

If C_1 does not write to any free variables from C_2, g, RI and C_2 does not write to any free variables from C_1, f, RI then

$$wslp[C_1 || C_2](f * g \mid RI) \geqslant wslp[C_1](f \mid RI) * wslp[C_2](g \mid RI). \tag{2.5}$$

If $wslp[C'](f * [RI] \mid emp) \geqslant g * [RI]$ and $C = C'$ is an atom or $C = \texttt{atom}\{C'\}$ then

$$wslp[C](f \mid RI) \geqslant g \tag{2.6}$$

and

$$wslp[C](f * [P] \mid RI) \geqslant wslp[C](f \mid P * RI). \tag{2.7}$$

```
while(c1 >= 0){
    {x1 := 1}
    [0.5]
    {x1 := 2};
    <z1 + c1> := x1;
    c1 := c1 - 1;
 }
```

Figure 1. Producer C_1

```
while(c2 >= 0){
    x2 := <z1+c2>;
    if(x2 != 0){
        {<z2+c2> := x2}
        [p]
        {<z2+c2> := -1}
        c2 := c2 - 1;
    }
}
```

Figure 2. Lossy channel C_2

```
while(c3 >= 0){
    x3 := <z2+c3>;
    if(x3 != 0){
        if(x3 != -1){
            l := l+1;
        }
        c3 := c3 - 1;
    }
}
```

Figure 3. Consumer C_3

2.1 Lossy Consumer Producer Example

As a short case-study, we examine a program C consisting of three threads: a producer C_1, a consumer C_3 and an unreliable channel C_2 between them. C_1 (see Figure 1) produces 1 or 2 (randomly with probability a half) and saves them in a global array indexed by z_1. We notate here `{C}[p]{C'}` for the probabilistic execution of either C with probability p or C' with probability $1 - p$. C_2 (see Figure 2) takes the results of C_1, throws them away with a probability of $1 - p$ or saves them in an array indexed by z_2 with a probability of p. Lastly, C_3 (see Figure 3) consumes the integers that have been correctly transferred in z_2 and increases a variable l after each consumption. If an integer was corrupted by C_2, the array indexed by z_2 only stores a -1.
We then create a program C that first initializes important variables and afterwards executes all three threads concurrently:

```
l := 0; c1, c2, c3 := k; (C1 || C2 || C3)
```

To reason about the probability realizing the condition $l = k + 1$, we use a resource invariant asserting that every value in the array indexed by z_1 and z_2 are either 0, 1 or 2:

$$\begin{aligned} RI = & \mathop{\circledast}_{i=0}^{k} ([z_1 + i \mapsto 0] + [z_1 + i \mapsto 1] + [z_1 + i \mapsto 2]) \\ & * \mathop{\circledast}_{i=0}^{k} ([z_2 + i \mapsto 0] + [z_2 + i \mapsto 1] + [z_2 + i \mapsto 2]) \end{aligned}$$

We use here a convex sum as a quantified version of the logical or. As long as all predicates we sum over are disjoint, the sum is indeed conservative regarding to

the logical or. Now, we can further reason about C:

$$\begin{aligned}
& wlp[C]([l = k+1]) \\
\geqslant & wlp[C]([l = k+1][k \geqslant 0] * [RI]) && \text{(monotonicity)} \\
= & wslp[C]([l = k+1][k \geqslant 0] * [RI] \mid emp) && \text{(by 2.4)} \\
\geqslant & wslp[C]([l = k+1][k \geqslant 0] \mid RI) && \text{(by 2.7)}
\end{aligned}$$

We can lower bound the probability that $l = k+1$ by a safe weakest liberal preexpectation that has the resource invariant RI. We can furthermore reason about the parallel composition (where we left out some of the details):

$$\begin{aligned}
& wslp[C_1 \,||\, C_2 \,||\, C_3](1 * 1 * [l = k+1] \mid RI) \\
\geqslant & wslp[C_1](1 \mid RI) * wslp[C_2](1 \mid RI) * wslp[C_3]([l = k+1] \mid RI) && \text{(by 2.5)} \\
& \ldots \\
\geqslant & (p^{c_2+1}[0 \leqslant c_2 \leqslant k] + [c_2 < 0])([-1 \leqslant c_3 \leqslant k][c_3 + l = k]) && (\ldots)
\end{aligned}$$

We computed the parallel composition by assigning each thread their own heap. In this case, every thread got an arbitrary part of the heap - the threads do not work on local heap memory anyway. Indeed, the program only accesses the shared memory, which is guarded by the resource invariant.

After further evaluating the initialization of the program, we can at last lower bound the probability to realize $l = k+1$ by $[k \geqslant 0]p^{k+1}$, which matches the probability we would expect in such a scenario.

3 Conclusion

We are eager to combine quantitative separation logic and concurrent separation logic into quantitative concurrent separation logic. Although this extension is still in its early steps and some soundness theorems are still in research, the idea itself looks promising given that we already could prove the probability for a useful predicate and the threads of Figures 1, 2 and 3.

It is still open whether more involved questions can also be proven and if an extension to the non-liberal variant of weakest preexpectation is viable and sound. On the other hand, there are more involved version of separation logic for concurrent programs, which could be lifted to probabilities, for example for multiple resources instead of only one [Vafeiadis, 2011].

References

Batz, Kevin et al. (Jan. 2019). "Quantitative separation logic: a logic for reasoning about probabilistic pointer programs". In: *Proc. ACM Program. Lang.* 3.POPL. DOI: 10.1145/3290347.

Dijkstra, E. W. (Sept. 1968). "A constructive approach to the problem of program correctness". In: *BIT Numerical Mathematics* 8.3, pp. 174–186. DOI: 10.1007/BF01933419.

Ishtiaq, Samin S. and Peter W. O'Hearn (Jan. 2001). "Bi as an assertion language for mutable data structures". In: *SIGPLAN Not.* 36.3, pp. 14–26. DOI: 10.1145/373243.375719.

Iverson, Kenneth E. (1962). *A programming language*. USA: John Wiley & Sons, Inc.

Morgan, Carroll, Annabelle McIver, and Karen Seidel (May 1996). "Probabilistic predicate transformers". In: *ACM Trans. Program. Lang. Syst.* 18.3, pp. 325–353. DOI: 10.1145/229542.229547.

Vafeiadis, Viktor (2011). "Concurrent separation logic and operational semantics". In: *Electronic Notes in Theoretical Computer Science* 276. (MFPS XXVII), pp. 335–351. DOI: 10.1016/j.entcs.2011.09.029.

Can Logic Programming Be Liberated from Backtracking?

– Extended Abstract –

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

Abstract
Logic programming is historically tight with Prolog and its backtracking search strategy. The use of backtracking was justified by efficiency reasons when Prolog was invented and is still present, although the incompleteness of backtracking destroys the elegant connection of logic programming and the underlying Horn clause logic. Moreover, it causes difficulties to teach logic programming. In this paper we argue that this is no longer necessary if new implementation approaches are taken into account.

1 Introduction

Logic programming was developed as a restriction of the general resolution principle [Robinson, 1965] to Horn clauses so that efficient linear (SLD-resolution) proofs can be constructed (see also Cohen [1988] for some historical background). It became popular when concrete implementations in the form of interpreters (and later compilers) for the programming language Prolog were available. Horn clauses and SLD-resolution are tightly connected to mathematical logic. The soundness and completeness of SLD-resolution establish the foundation of logic programming [Lloyd, 1987]. Unfortunately, the memory restrictions of computers at that time caused a gap between these theoretical foundations and the practice of logic programming in Prolog: non-deterministic computations are implemented by backtracking so that the theoretical completeness of SLD-resolution is lost. For instance, consider the definition of a Prolog predicate relating a list and its last element:

```
last([H|T],E) :- last(T,E).
last([E],E).
```

This definition works when the list is known:

```
?- last([1,2,3],E).
E = 3
```

One of the advantages of logic programming is the absence of fixed input and output parameters. Instead of providing a known value for an argument of a predicate, one can also call the predicate a free variable for this argument (as `E` above) so that a result is computed by binding this variable to some value. In practice, this advantage is often lost when non-deterministic search is implemented by backtracking, since infinite branches in a search tree might preclude the computation of valid answers. For instance, Prolog does not compute any result for the definition of `last`, as shown above, when the list is unknown, e.g., for the goal `last(L,3)`: the backtracking strategy causes an infinite chain of applications of the first rule.

Generally, the use of backtracking in logic programming has several disadvantages:

- ▻ The theoretical completeness of SLD-resolution is lost.
- ▻ It hinders the teaching of logic programming since beginners are often faced with the influence of the search strategy.
- ▻ Programmers have to think about the influence of backtracking to the success of computations—a contradiction to the idea of *declarative programming*.

These problems can be solved if backtracking is replaced by a complete search strategy. Thus, abandoning backtracking as a default for logic programming is similar to the removal of the von Neumann bottleneck by functional programming [Backus, 1978]: one obtains a higher, declarative programming style which frees the programmer from thinking about low-level control details.

Unfortunately, many aspects of Prolog, in particular, the connection to the external world (e.g., file system, networks, graphics) heavily depends on the backtracking strategy. Thus, in order to get a better logic programming language, we have to switch from Prolog to a paradigm supporting a clean and declarative connection to external resources, as developed in functional programming [Wadler, 1997]. Therefore, we consider in the following *functional logic* languages.

2 Functional Logic Programming

Functional logic languages [Antoy and Hanus, 2010] combine the main features of functional and logic languages in a single programming model. In particular, demand-driven evaluation of expressions is amalgamated with non-deterministic

search for values. In functional logic programs, operations are defined by rewrite rules, as in functional languages, but rules can be overlapping, as in logic languages. The archetype of an operation defined by overlapping rules is the non-deterministic *choice*, defined in the functional logic language Curry [Hanus (ed.), 2016] as the infix operator "?" as follows:

```
x ? _ = x
_ ? y = y
```

Hence, an expression like "`0 ? 1`" yields two values: `0` and `1`. In contrast to Prolog, the concrete strategy to compute these values, i.e., the search strategy, is not fixed in Curry. Implementations of Curry can support various search strategies. For instance, the Curry system KiCS2 [Braßel, Hanus, et al., 2011] has options to select different search strategies, like depth-first, breadth-first, iterative deepening, or parallel search.

Early implementations of functional logic languages, like PAKCS [Antoy and Hanus, 2000] or TOY [López-Fraguas and Sánchez-Hernández, 1999], used Prolog as a target language due to its built-in support for non-determinism. A drawback of this approach is that they inherit the incompleteness of Prolog's backtracking strategy. In order to get rid of this fixed search strategy, subsequent implementations are based on the idea to represent non-deterministic choices as data. Thus, instead of directly evaluating non-deterministic branches, the alternatives are returned as a tree structure so that search strategies can be defined as tree traversals, which supports an easy switch between different strategies.

We omit the details of actual techniques to evaluate expressions to such tree structures. It is sufficient to keep in mind that the operational semantics is based on graph transformations, like pull-tabbing [Antoy, 2011], but the details are not relevant for the programmer as long as a complete search strategy is used. In general, breadth-first search could be used. However, if the search space is finite, depth-first search is also reasonable.

3 Comparing Search Strategies

These theoretical considerations are useless if they are not supported by practical implementations. Instead of compiling functional logic languages into Prolog, one can compile them into a deterministic target language and add explicit support for non-determinism, as done with KiCS2 [Braßel, Hanus, et al., 2011] which compiles Curry programs into Haskell programs. The intermediate language ICurry [Antoy, Hanus, et al., 2020] is intended to compile Curry into imperative

target languages. It has been used to translate Curry to LLVM code [Antoy and Jost, 2016], to C or Python programs [Wittorf, 2018], and to Julia programs [Hanus and Teegen, 2021]. A recent implementation, called Curry2Go[1], compiles ICurry programs into Go[2] programs. Go is a statically typed language with garbage collection and direct support for CSP-like concurrency [Hoare, 1978] and lightweight threads (*goroutines*). The latter feature is used by Curry2Go to provide a fair search strategy which avoids the limitations of backtracking-based logic programming languages discussed above.

Due to the explicit handling of non-deterministic computations, Curry2Go supports various search strategies. The run-time system works with a queue or set of tasks (depending on the search strategy) where each task evaluates some non-deterministic branch. The difference between depth-first (DFS) and breadth-first (BFS) search amounts to a different strategy to add new tasks to the queue: DFS adds new tasks at the front and BFS adds them at the tail of the queue.

Each task evaluates an expression to some value (to be more precise, a head normal form) or is split into two new tasks if some non-deterministic choice occurs. If the evaluation of an expression does not terminate and non-deterministic choices do not occur, even a breadth-first search strategy might not compute existing values. For instance, consider the following contrived example:

```
idND :: a → a
idND n = loop ? n ? loop
```

where `loop` is non-terminating (e.g., defined by `loop = loop`). Semantically, `idND` is the identity function but, operationally, it is non-deterministically defined with looping alternatives. Although `0` is a value of `idND 0`, both DFS and BFS do not return any value but loop. To avoid such kind of incompleteness, Curry2Go also implements a *fair search* (FS) strategy. FS evaluates each task concurrently as a goroutine and collects the computed result in a channel where these goroutines write their computed results. More details about this implementation can be found in Böhm, Hanus, and Teegen [2021].

Table 1 shows the run times[3] (in seconds as the average of three runs) of some examples executed with different Curry systems and search strategies. PAKCS [Hanus, Antoy, et al., 2020], which is part of Debian and Ubuntu Linux distributions, compiles to Prolog (SWI-Prolog 8.0) and is based on backtrack-

[1] The source code is available at https://github.com/curry-language/curry2go. A distribution can be downloaded at https://www-ps.informatik.uni-kiel.de/curry2go/.

[2] https://golang.org/

[3] All benchmarks were executed on a Linux machine running running Debian 10 with an Intel Core i7-7700K (4.2GHz) processor with eight cores.

Table 1. Comparing Curry system with search strategies

Example	PAKCS	KiCS2		Curry2Go		
		DFS	BFS	DFS	BFS	FS
`nrev_4096`	8.28	0.43	0.44	1.16	1.17	1.16
`takPeano_24_16_8`	54.75	0.30	0.30	5.08	5.09	5.08
`primesHO_1000`	38.88	0.43	0.44	4.08	4.09	4.08
`psort_13`	16.46	0.77	2.88	5.20	5.27	5.43
`addNum_2`	0.19	0.98	1.77	0.44	0.43	0.41
`addNum_5`	0.22	3.21	5.18	1.06	1.06	0.45
`addNum_10`	0.29	10.03	15.55	2.48	2.48	0.69
`select_50`	0.14	0.61	0.67	0.11	0.11	0.08
`select_100`	0.45	4.97	5.25	0.14	0.14	0.10
`select_150`	1.08	21.25	26.14	0.23	0.23	0.12
`isort_primes4`	15.63	0.42	0.42	1.74	1.74	1.72
`psort_primes4`	155.95	0.40	0.42	1.72	1.72	0.94

ing. KICS2 [Braßel, Hanus, et al., 2011] compiles to Haskell (GHC 8.4) where non-determinism is implemented by lazily generating search trees which are explored by various search strategies. Curry2Go compiles to Go (Version 1.16) and manages a queue of tasks to implement DFS and BFS or use goroutines communicating via channels to implement FS.

The first three benchmarks are typical purely functional programs. `nrev_4096` is the quadratic naive reverse algorithm applied to a list with 4096 elements, `takPeano` is a highly recursive function on naturals [Partain, 1993] applied to arguments (24,16,8) in Peano representation, and `primesHO_1000` computes the 1000th prime number by constructing an infinite list of all primes via the sieve of Eratosthenes (using higher-order functions). These benchmarks indicate that, for purely functional programs, Curry2Go is much faster than PAKCS but less efficient than KiCS2. The latter is not surprising since Haskell/GHC is highly optimized for these kinds of programs.

One might think that the less efficient behavior of Prolog-based PAKCS is the fact that Prolog also supports non-determinism. This hypothesis is refuted by the remaining non-deterministic benchmark programs. `psort_13` is the naive permutation sort applied to a list of 13 elements. `addNum_`n non-deterministically chooses a number (out of 2000) and adds it n times, and `select_`n non-deterministically selects an element in a list of length n and sums up the element and the list without the selected element. The considerable slowdown in KiCS2 with increas-

ing values for n is caused by the duplication of choices in pull-tab steps when non-deterministic expressions are shared, as discussed in Hanus and Teegen [2021]. This is avoided in Curry2Go by adding a sort of memoization for choices, as described in Böhm, Hanus, and Teegen [2021]; Hanus and Teegen [2021].

Apart from the fact that the fair search strategy of Curry2Go is the only operationally complete strategy (e.g., it is able to compute a value for `idND 0`), there are also other interesting differences between the search strategies. For instance, KiCS2 shows some overhead of BFS compared to DFS (possibly due to the additional structures used to implement a breadth-first tree search), whereas there is almost no overhead in Curry2Go (since the difference between BFS and DFS is just a different schedule of tasks). Moreover, the fair search (FS) strategy is sometimes faster than BFS and DFS thanks to the use of goroutines possibly scheduled on different processors. This is also visible in the last two lines of Table 1 which show the time to sort

```
[primes!!303, primes!!302, primes!!301, primes!!300]
```

with the deterministic insertion sort (`isort`) and the non-deterministic permutation sort (`psort`) algorithm, respectively, where `primes` defines the infinite list of all prime numbers. Due to backtracking, identical computations might be repeated if they occur in different non-deterministic branches. Thus, `primes` is re-evaluated by PAKCS several times when the list is passed to the non-deterministic operation `psort`. This is not the case in implementations which represent choices in a graph structure so that the results of deterministic computations are shared across non-deterministic evaluations [Braßel and Huch, 2007]. In Curry2Go, where non-deterministic branches are evaluated by goroutines, it could be even better to use a non-deterministic algorithm since it might map evaluations of common subexpressions to different computation nodes, as shown by the results for `psort_primes4`. This is also demonstrated with the following benchmark, where the timings for `psort_primes4` increased to a list of eight prime numbers and executed with different numbers of processors (by setting the Go variable `GOMAXPROCS`) are shown:

# processors	1	2	4	8
`psort_primes8`	6.57	3.41	1.99	1.55

Hence, the presence of multiple processors can be exploited in a non-deterministic program without requiring specific user annotations.

4 Conclusions

We have compared the implementation of different search strategies for non-deterministic programming, like depth-first, breadth-first and fair (concurrent) search. Due to memory restrictions in early years, depth-first search implemented by backtracking was introduced and is still used in the logic programming language Prolog. Backtracking causes a gap between the theory and practice of logic programming and complicates teaching and the practical use of logic programming techniques. By using recent implementation techniques developed in functional logic programming, operationally complete strategies can compete with backtracking and can even be faster on multi-processor architectures. Hence, logic programming must not be tight to backtracking: with modern implementation technologies, one can use better strategies that avoid the classical drawbacks of backtracking, namely the operational incompleteness of search. This closes the gap between theory and practice of logic programming and could lead to a higher, really declarative programming style.

References

Antoy, S. (2011). "On the correctness of pull-tabbing". In: *Theory and Practice of Logic Programming* 11.4-5, pp. 713–730. DOI: 10.1017/S1471068411000263.

Antoy, S. and M. Hanus (2000). "Compiling multi-paradigm declarative programs into Prolog". In: *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*. Springer LNCS 1794, pp. 171–185. DOI: 10.1007/10720084_12.

– (2010). "Functional logic programming". In: *Communications of the ACM* 53.4, pp. 74–85. DOI: 10.1145/1721654.1721675.

Antoy, S., M. Hanus, et al. (2020). "ICurry". In: *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*. Springer LNCS 12057, pp. 286–307. DOI: 10.1007/978-3-030-46714-2_18.

Antoy, S. and A. Jost (2016). "A new functional-logic compiler for Curry: Sprite". In: *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*. Springer LNCS 10184, pp. 97–113. DOI: 10.1007/978-3-319-63139-4_6.

Backus, J. (1978). "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". In: *Comm. of the ACM* 21.8, pp. 613–641.

Böhm, J., M. Hanus, and F. Teegen (2021). "From non-determinism to goroutines: a fair implementation of Curry in Go". In: *Proc. of the 23rd International Sympo-*

sium on Principles and Practice of Declarative Programming (PPDP 2021). ACM Press.

Braßel, B., M. Hanus, et al. (2011). "KiCS2: a new compiler from Curry to Haskell". In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. Springer LNCS 6816, pp. 1–18. DOI: 10.1007/978-3-642-22531-4_1.

Braßel, B. and F. Huch (2007). "On a tighter integration of functional and logic programming". In: *Proc. APLAS 2007*. Springer LNCS 4807, pp. 122–138. DOI: 10.1007/978-3-540-76637-7_9.

Cohen, J. (1988). "A view of the origins and development of Prolog". In: *Communications of the ACM* 31.1, pp. 26–36. DOI: 10.1145/35043.35045.

Hanus, M., S. Antoy, et al. (2020). *PAKCS: the Portland Aachen Kiel Curry System*. Available at http://www.informatik.uni-kiel.de/~pakcs/.

Hanus, M. and F. Teegen (2021). "Memoized pull-tabbing for functional logic programming". In: *Proc. of the 28th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2020)*. Springer LNCS 12560, pp. 57–73. DOI: 10.1007/978-3-030-75333-7_4.

Hanus (ed.), M. (2016). *Curry: an integrated functional logic language (vers. 0.9.0)*. Available at http://www.curry-lang.org.

Hoare, C.A.R. (1978). "Communicating sequential processes". In: *Communications of the ACM* 21.8, pp. 666–677. DOI: 10.1145/359576.359585.

Lloyd, J.W. (1987). *Foundations of logic programming*. Springer, second, extended edition.

López-Fraguas, F. and J. Sánchez-Hernández (1999). "TOY: a multiparadigm declarative system". In: *Proc. of RTA'99*. Springer LNCS 1631, pp. 244–247.

Partain, W. (1993). "The nofib benchmark suite of Haskell programs". In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer, pp. 195–202.

Robinson, J.A. (1965). "A machine-oriented logic based on the resolution principle". In: *Journal of the ACM* 12.1, pp. 23–41.

Wadler, P. (1997). "How to declare an imperative". In: *ACM Computing Surveys* 29.3, pp. 240–263.

Wittorf, M.A. (2018). "Generic translation of Curry programs into imperative programs (in German)". MA thesis. Kiel University.

Meta-Operatoren in

Christian Heinlein

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

Abstract. Meta-Operatoren sind spezielle Operatoren in der Programmiersprache MOSTflexiPL, deren Anwendungen über gewöhnliche Operatoranwendungen „gestülpt“ werden können, um diese in sehr flexibler Weise zu parametrisieren. Abhängig von Laufzeitbedingungen, können so aus einem einzigen Ausdruck im Quelltext zur Laufzeit unterschiedliche konkrete Ausdrücke entstehen. Dies ist insbesondere zur Verarbeitung und Weitergabe optionaler, alternativer und wiederholter Syntaxteile eines Operators nützlich.

1 Einleitung

MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language) [1] ist eine Programmiersprache, deren Syntax vom Anwender nahezu beliebig erweitert und angepasst werden kann. Aufbauend auf einer kleinen Menge vordefinierter Grundoperatoren (z. B. für Arithmetik, Logik und Ablaufsteuerung), 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. `-•` für Vorzeichenwechsel, `•+•` für Addition oder `•!` für Fakultät; • bezeichnet jeweils einen Operanden) auch Mixfix-Operatoren (z. B. `•[•]` für den Zugriff auf Elemente eines Felds), Steueranweisungen (z. B. `if•then•else•end`), Typkonstruktoren (z. B. `List•`) und Deklarationsformen (z. B. `•:•`) ab.

2 Beispiele

Beispielsweise definiert

```
(x:int) "²" -> (int = x * x)
```

einen neuen Operator $\bullet^2$, der das Quadrat seines Operanden berechnet und anschließend in Ausdrücken wie z. B. 10^2 oder $(2+3)^2$ verwendet werden kann.

Die Signatur (links vom Pfeil) eines solchen Operators ist allgemein eine Folge von Namen und Parameterdeklarationen. Ein Name ist entweder eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt (z. B. `x` oder `abc123`) oder eine Folge beliebiger Zeichen innerhalb von Anführungszeichen (die nur bei der Definition des

Operators, aber nicht bei seiner Verwendung angegeben werden). Eine Parameterdeklaration ist von der Gestalt `(name:type)` mit einem Namen *name* wie gerade definiert und einem Typ *type*. Rechts vom Pfeil steht etwas der Gestalt `(type=impl)` mit einem Typ *type* (dem Resultattyp des Operators) und einem Ausdruck *impl*, der die Implementierung des Operators (wie in funktionalen Sprachen) darstellt.

Noch allgemeiner kann die Signatur eines Operators unter Verwendung der bekannten EBNF-Symbole auch optionale, alternative und wiederholte Teile enthalten, zum Beispiel:

```
sum of (x:int) { and (y:int) } -> (int = ......);

calc [minus] (x:int) { (plus|minus) (y:int) } -> (int = ......);

dnf [not] (a:bool) { and [not] (b:bool) }
{ or [not] (c:bool) { and [not] (d:bool) } }
-> (bool = ......);

print { [(T:type)] (x:T) } -> (bool = ......)
```

Mögliche Verwendungen der so definierten Operatoren wären dann zum Beispiel:

```
sum of 1;
sum of 1 and 2;
sum of 1 and 2 and 3;

calc 1 plus 2;
calc minus 1 plus 2 minus 3;

dnf not a or b and not c;
dnf a and b or not c or d and e and not f;

print 1 2.5 "hallo"
```

Weitere Beispiele für Operatoren mit optionalen, alternativen und wiederholten Teilen sind Steueranweisungen wie z. B. `if•then•{elseif•then•}[else•]end` und `try•{catch•then•}[finally•]end`.

3 Probleme

Da solche Operatoren auf (zum Teil unendlich viele) unterschiedliche Arten angewandt werden können, besteht für ihre Implementierung das Problem herauszufinden, wie genau der jeweilige Operator jeweils angewandt wurde, d. h. ob ein optionaler Teil bei einer Anwendung vorhanden ist oder nicht, welche von mehreren Alternativen verwendet wurde und wie oft und auf welche Arten ein wiederholter Teil angegeben wurde. Je nach konkreter Anwendung eines Operators, kann ein Parameter innerhalb eckiger und runder Klammern einen Wert besitzen oder nicht, während Parameter innerhalb geschweifter Klammern sogar beliebig viele Werte besitzen können, die – wie das Bei-

spiel `print` zeigt – sogar unterschiedliche Typen besitzen können, sodass eine Zusammenfassung aller Wert in einem Feld o. ä. im allgemeinen nicht möglich ist.

Ein weiteres Problem besteht darin, solche Teile einer Operatoranwendung entweder unverändert oder geeignet angepasst an andere Operatoren weiterzugeben. Beispielsweise ist es wünschenswert, dass ein Operator `avg of•{and•}` zur Berechnung des Durchschnitts beliebig vieler Werte diese Werte an den Operator `sum of•{and•}` weitergeben kann, um auf einfache Art und Weise ihre Summe zu berechnen. Ebenso könnte ein Operator `cnf[not]•{or[not]•}{and[not]•{or[not]•}}` (conjunctive normal form) unter Ausnutzung der Regeln von De Morgan auf einfache Art und Weise unter Verwendung des Operators `dnf[not]•{and[not]•}{or[not]•{and[not]•}}` (disjunctive normal form) implementiert werden, wenn er die Werte seiner Parameter sowie das Vorhanden- oder Nichtvorhandensein der Namen `not` geeignet an diesen Operator weitergeben könnte.

4 Lösung durch Meta-Operatoren

Eine mögliche Lösung für die zuvor beschriebenen Probleme sind Meta-Operatoren, deren Bedeutung und Verwendung am besten anhand von Beispielen erläutert werden kann:

```
sum of (x:int) { and (y:int) } -> (int = x ^{ + y })
```

Hier wird ausgenutzt, dass es zu jeder Wiederholungsklammer `{...}` in der Signatur eines Operators einen korrespondierenden Meta-Operator **`^{○}`** gibt, dessen Anwendungen über gewöhnliche Operatoranwendungen „gestülpt" werden können. (Um gewöhnliche und Meta-Operatoren besser unterscheiden zu können, werden die Namen von Meta-Operatoren jeweils fett gedruckt.) Das Symbol ○ kennzeichnet im Gegensatz zu • keinen gewöhnlichen Operanden, sondern einen Meta-Operanden, der aus beliebig vielen Namen und (gewöhnlichen) Operanden der „darunterliegenden" Operatoranwendungen (sowie weiterer Meta-Operator-Anwendungen, für deren Meta-Operanden dasselbe gilt) bestehen kann. Zur Laufzeit (!) wird so eine Meta-Operator-Anwendung dann durch beliebig viele Aneinanderreihungen ihres Meta-Operanden ersetzt, sodass aus dem Ausdruck `x` **`^{`** `+ y` **`}`** dann entweder `x` oder $x + y_1$ oder $x + y_1 + y_2$ usw. werden kann. Die Indizes bei `y` sollen hierbei anzeigen, dass jedes `y` einen anderen Wert besitzen kann, ebenso wie die Konstante `y` in C++ [2] in jedem Durchlauf der Schleife

```
for (const int y : { 1, 2, 3 }) ......
```

einen anderen Wert besitzt. Die konkreten Werte für $y_1, y_2, \ldots$ stammen hierbei aus der konkreten Operatoranwendung, sodass aus den Anwendungen `sum of 1` bzw. `sum of 1 and 2` bzw. `sum of 1 and 2 and 3` zum Beispiel die Ausdrücke `1` bzw. `1 + 2` bzw. `1 + 2 + 3` in der Implementierung des Operators `sum` werden.

Da MOSTflexiPL eine statisch typisierte Sprache ist, muss der Compiler zur Übersetzungszeit überprüfen, ob jeder Ausdruck, der auf diese Weise zur Laufzeit entstehen

kann, sowohl syntaktisch als auch semantisch korrekt ist, was im obigen Beispiel offensichtlich ist, im allgemeinen aber eine nicht triviale Aufgabe darstellt.

Das folgende Beispiel zeigt die Verwendung weiterer Arten von Meta-Operatoren:

```
calc [minus] (x:int) { (plus|minus) (y:int) } -> (int =
    ^[ - | ] x ^{ ^( + | - ) y }
)
```

Zu jeder Verzweigungsklammer `(...|...)` mit zwei Alternativen in der Signatur eines Operators gibt es in der Implementierung des Operators einen korrespondierenden Meta-Operator **`^(○|○)`** mit zwei Meta-Operanden (und entsprechend für Klammern mit mehr als zwei Alternativen). Eine Anwendung dieses Meta-Operators wird zur Laufzeit durch einen der beiden Meta-Operanden ersetzt, je nachdem welche Alternative bei der vorliegenden Operatoranwendung verwendet wurde.

Zu jeder Optionsklammer `[...]` in der Signatur eines Operators gibt es in der Implementierung des Operators korrespondierende Meta-Operatoren **`^[○]`** und **`^[○|○]`** mit einem bzw. zwei Meta-Operanden. Eine Anwendung des Meta-Operators **`^[○]`** wird zur Laufzeit entweder durch den Meta-Operanden oder durch nichts ersetzt, je nachdem ob der optionale Teil bei der vorliegenden Operatoranwendung vorhanden ist oder nicht. Eine Anwendung des Meta-Operators **`^[○|○]`** wird zur Laufzeit entweder durch den ersten oder den zweiten Meta-Operanden ersetzt, je nachdem ob der optionale Teil bei der vorliegenden Operatoranwendung vorhanden ist oder nicht.

Aufgrund dieser Regeln entstehen aus den Operatoranwendungen `calc 1 plus 2` bzw. `calc minus 1 plus 2 minus 3` die Ausdrücke `1 + 2` bzw. `-1 + 2 - 3` in der Implementierung des Operators `calc`. Auch hier muss der Compiler wieder überprüfen, dass jeder Ausdruck, der zur Laufzeit entstehen kann, korrekt ist.

5 Ungewöhnliche Verwendungen von Meta-Operatoren

Durch konsequente Anwendung der bis jetzt beschriebenen Regeln für Meta-Operatoren sind auch ungewöhnliche Verwendungen wie zum Beispiel folgendes möglich:

```
strange (x:int) (a: plus | times) (y:int)
                         (b: minus | div) (z:int) -> (int =
    x ^(a: + | * ) y ^(b: - | / ) z
)
```

Um die zu den beiden Verzweigungsklammern gehörenden Meta-Operatoren unterscheiden zu können, sind die Klammern mit den Namen `a` bzw. `b` benannt, die dann auch bei der Verwendung der Meta-Operatoren angegeben werden können. Abhängig von der konkreten Anwendung des Operators `strange` können in seiner Implementierung vier verschiedene Ausdrücke entstehen, deren Operatorbaum aufgrund der üblichen Vorrangregeln für arithmetische Operatoren jeweils unterschiedlich aufgebaut ist:

```
strange x plus y minus z   →  (x + y) - z
strange x plus y div z     →  x + (y / z)
strange x times y minus z  →  (x * y) - z
strange x times y div z    →  (x * y) / z
```

Die Klammern dienen hier nur zur Verdeutlichung der unterschiedlichen Operatorbäume.

6 Vergleichbare Konzepte

Meta-Operatoren haben eine gewisse Ähnlichkeit mit den Sprachmitteln für bedingte Übersetzung in C und C++ [2]: Abhängig davon, ob die Makros `a` und `b` definiert sind oder nicht, entsteht aus dem Code

```
x
    #ifdef a
    +
    #else
    *
    #endif
y
    #ifdef b
    -
    #else
    /
    #endif
z
```

zur Übersetzungszeit ebenfalls einer der in §5 genannten Ausdrücke. Die wesentlichen Unterschiede zu Meta-Operatoren sind jedoch, dass letztere erst zur Laufzeit ersetzt werden und dass der C/C++-Präprozessor keine Möglichkeit für Wiederholungen bietet, sodass Meta-Operatoren wesentlich mehr Flexibilität bieten.

Meta-Operatoren haben außerdem eine gewisse Ähnlichkeit mit Makros in Lisp [3]: Anwendungen des im folgenden definierten Makros `strange` werden abhängig von den Werten der Parameter `a` und `b` zur Laufzeit ebenfalls durch einen der vier in §5 genannten Ausdrücke (in Lisp-Syntax) ersetzt:

```
(defmacro strange (x a y b z)
    (if a
        (if b
            `(- (+ ,x ,y) ,z)
            `(+ ,x (/ ,y ,z))
        )
```

```
            (if b
                `(- (* ,x ,y) ,z)
                `(/ (* ,x ,y) ,z)
            )
        )
    )
```

Ein wesentlicher Unterschied zu Meta-Operatoren ist hier, dass in Lisp keine Überprüfungen zur Übersetzungszeit stattfinden und deshalb prinzipiell Fehler zur Laufzeit auftreten können, wenn einer der entstehenden Ausdrücke nicht korrekt ist. Außerdem kann die Abhängigkeit der resultierenden Ausdrücke von `a` und `b` in Lisp nicht unabhängig voneinander ausgedrückt werden.

7 Benutzerdefinierte Meta-Operatoren

Da die Syntax von MOSTflexiPL beliebig erweiterbar und anpassbar ist, gibt es neben den bis jetzt beschriebenen vordefinierten Meta-Operatoren auch die Möglichkeit, eigene Meta-Operatoren zu definieren. Mit einem geeignet definierten Meta-Operator **^{•=•..•:○}** – der neben einem Meta-Operanden ○ auch drei gewöhnliche Operanden • besitzt – würde dann z. B. der Ausdruck

```
sum of 0 ^{ i = 1 .. 3 : and i }
```

zur Laufzeit durch `sum of 0 and 1 and 2 and 3` ersetzt werden.

8 Zusammenfassung und Ausblick

Meta-Operatoren sind Operatoren mit speziellen Meta-Operanden, deren Anwendungen über gewöhnliche Operatoranwendungen „gestülpt“ werden können, um diese in sehr flexibler Weise zu parametrisieren. Abhängig von Laufzeitbedingungen – z. B. wie oft ein wiederholter Syntaxteil bei einer Operatoranwendung vorhanden ist oder ob ein optionaler Teil vorhanden ist oder nicht – können so aus einem einzigen Ausdruck im Quelltext zur Laufzeit mehrere (eventuell sogar unendlich viele) verschiedene Ausdrücke mit ähnlicher oder auch sehr unterschiedlicher Bedeutung entstehen. Damit können optionale, alternative und wiederholte Syntaxteile eines Operators in seiner Implementierung sehr einfach verarbeitet oder an andere Operatoren weitergegeben werden.

Referenzen

[1] C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: J. Edwards (ed.): *Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)* (Tucson, AZ, October 2012), 159–178.

[2] B. Stroustrup: *The C++ Programming Language* (Fourth Edition). Addison-Wesley, Upper Saddle River, NJ, 2013.

[3] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

Effectful Effects and Contextual Effect Polymorphism

Samuel Pilz
Compilers and Languages Group
TU Wien
samuel.pilz@tuwien.ac.at

Abstract

Effect systems enable programmers to express a wide range of control flow patterns, typically implemented by extending the type signatures with effects. We illustrate this by reviewing two recently proposed effect systems, namely the languages `Effekt` and `Olaf` that strike different trade-offs between expressiveness and ease of use: `Effekt` [Brachthäuser, Schuster, and Ostermann, 2020] simplifies the effect system by introducing the notion of contextual effect polymorphism. `Olaf` [Zhang, Salvaneschi, and Myers, 2020] introduces effectful effects, which allow effect handlers to raise effects of their own, adding the possibility of bidirectional control flow. More recent versions of `Effekt` have been extended by the powerful feature of bidirectional control flow via effectful effects. Inspired by `Olaf`, we extend `Effekt` further by integrating higher order effects, adding more expressiveness for advanced use-cases, while maintaining ease of use for simpler applications. Finally, we discuss an application of the proposed approach for implementing complex asynchronous computations.

References

Brachthäuser, Jonathan Immanuel, Philipp Schuster, and Klaus Ostermann (2020). "Effects as capabilities: effect handlers and lightweight effect polymorphism". In: *Proceedings of the ACM on Programming Languages* OOPSLA, 126:1–126:30.

Zhang, Yizhou, Guido Salvaneschi, and Andrew C Myers (2020). "Handling bidirectional control flow". In: *Proceedings of the ACM on Programming Languages* OOPSLA, 139:1–139:30.

Making a Monadic Curry

The Quest for a Complete and Fast(er) Compiler Implementation

Kai-Oliver Prott
kpr@informatik.uni-kiel.de
CAU Kiel

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

Abstract
We present a monadic model for the functional-logic programming language *Curry* that also includes any recent extensions to the language like weak encapsulation of non-determinism via set functions. We take a look at different existing compiler implementations of Curry and discuss any shortcomings of their solution. By combining and adapting various existing solutions, we design a model for Curry that makes no semantic compromise while still being as fast or faster than other implementations. It can also be used to implement a compiler based on a monadic lifting of the source code.

1 Introduction

Functional logic languages like *Curry* [Hanus, Kuchen, and Moreno-Navarro, 1995] combine some of the most important features of functional and logic languages. In the context of Curry, the concepts of lazy and demand-driven evaluation, functions as first-class citizens as well as polymorphism are integrated with computations over free variables and non-deterministic search. Curry's syntax is mostly borrowed from the purely functional language *Haskell* [Marlow, 2010], but the extension with free variables allows more elegant formulations of common problems and functions. Consider a function that returns the last element of a single-linked list data structure for example.

```
last l | xs ++ [x] =:= l = x where x, xs free

[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

The single equation for `last` defines the last element of a list to be some free variable, such that appending it to the back of another list using `(++)` results

in the original input list. While free variables are one concept from logic programming that Curry integrates, it also provides the binary operator `(?)` for non-deterministic choice. For example, we can implement a function that non-deterministically chooses one element from a (non-empty) list and fails with no result on an empty list.

```
anyOf []     = failed
anyOf (x:xs) = x ? anyOf xs
```

While non-determinism, free variables as well as unification are features that have been part of the Curry language from the beginning, the language has since been extended with various abstractions for pattern-matching and encapsulation of non-deterministic evaluation.

However, there are multiple compilers for Curry with a varying degree of support for the "extended" language. This difference in feature support stems from the fact that those compilers each use a different approach to compile Curry and they all use a different language as the compilation target. For this paper, the three main compilers we will take a look at are *PAKCS* (Curry to Prolog) [Hanus, 2018], KiCS2 (Curry to Haskell) [Braßel et al., 2011] and curry2go (duh) [Böhm, Hanus, and Teegen, 2021] Our Goal is to create a monadic model of Curry that supports *all* of the new features while also integrating recent optimizations for the run-time evaluation of Curry-Programs.

In the next sections, we will take a look at different features of Curry. For each of them we will examine the current implementations and their deficiencies in different compilers. Using that knowlege, we can then adapt the best approach for our monadic Curry implementation.

2 Modeling Non-Determinism and Searching for Solutions

Non-determinism is often represented in a binary tree structure, where the nodes are created by the choice operator `(?)`. For this reason, both curry2go and KiCS2[1] evaluate a Curry program into a tree data type. Afterwards, one can search for concrete solutions in that tree, for example with breath-first search or other strategies. In contrast, the implementation of PAKCS relies on the backtracking evaluation provided by its prolog run-time system and does not allow for different search strategies.

Since we do want to enable different search strategies, we will also use a tree-based implementation for our model. The concrete result values will be annotated

[1]KiCS2 integrates the binary tree structure directly into the data types, but the idea is the same.

at the leaves of the tree. We will use the following data type definition, which can be annotated with other information at the node and failure constructors. These annotations will later be instantiated with various types, but for now we can ignore them.

```
data Tree n e a = Leaf a
                | Empty f
                | Node n (Tree n e a) (Tree n e a)
```

Note that `Tree` applied to just two variables is a monad and can be used in a monad transformer [Liang, Hudak, and M. Jones, 1995; M. P. Jones, 1995] stack.

In a non-determinism implementation based on a tree structure, the evaluation of a given program is based on *pull-tabbing* [Antoy, 2011]. If a function demands the evaluation of one of its arguments and the argument is non-deterministic, the non-deterministic choices in the argument are *pulled* to the top and the function is applied at each leaf of the tree. This is visualized in the following example.

```
([1] ? [2]) ++ [3]    ==>    ([1] ++ [3]) ? ([2] ++ [3])
```

Our final goal is to implement a `Curry` monad that combines a tree for non-determinism with some stateful information (`CurryState`) and free variables (handled by `CurryVal`). `Curry` will look roughly as follows.

```
newtype Curry a = Curry (StateT CurryState (Tree _ _) (CurryVal a))
```

This monadic model can be used to implement a compiler from Curry to Haskell in the future together with a non-strict monadic lifting [Wadler, 1990]. One thing to note is that in our lifting not just the functions, but also data types are lifted to allow for lazy and deep non-determinism nested in data structures. For example, the following code shows Haskell's list data type and our lifted variant in GADT syntax.

```
data [] a where          data ListC a where
  []  :: [a]               NilC  :: ListC a
  (:) :: a -> [a] -> [a]   ConsC :: Curry a -> Curry (ListC a) -> ListC a
```

3 Free Variables and Unification

We have already seen the function `last` as an example for using free variables and unification. In general, this concept allows for computation of functions with incomplete information and has also been used in the context of web programming [Hanus, 2021]. Recently, free variables have been restricted on the

type-level to only be possible for constructor-based types and primitive data types like `Int` [Hanus and Teegen, 2019].

There are three different approaches used by the current Curry Compilers to implement this feature.

1. The KiCS2 Compiler uses generator functions to represent free variables. That is, a free variable of type τ is translated to a 0-ary function that enumerates all possible constructors and terms of type τ non-deterministically. This concept is often called narrowing in the context of logic programming.

   As a consequence, *strict* unification of a free variable with some value can be implemented as a simple check that the values are structurally equal. This can be slow for data types with a lot of constructors like `Int`. Thus, the implementation for `Int` uses a binary representation to optimize unification. If the possible set of values for a free variable is infinite, computations/unifications with value generators might not even terminate in some cases.

2. Since PAKCS uses Prolog as its back-end for compilation, free variables (and unification to some extent) are handled by Prolog's run-time system. In Prolog, free variables are treated as run-time objects where unification is used to solve term equations. While this is a very efficient way to implement free variables that also avoids the infinite search space problem of unification whith value generators, it suffers from a small problem with the instantiation of unconstrained variables. Consider the following code snippet.

   ```
   let x free in ensureNotFree x :: Bool
   ```

   The variable `x` is free in the function call to `ensureNotFree` and, thus, would have to be instantiated. However, PAKCS uses an untyped approach and in consequence does not know with which terms to instantiate the variable. Instead, the computation is suspended until the variable is instantiated, because a shared ocurrence is instantiated for example. In the small example above, this will never happen and the evaluation will not resume.

3. Although the curry2go compiler is similar to PAKCS in that it uses an untyped approach for its implementation, it cannot rely on Go's run-time system to handle free variables. Instead, the run-time system of the curry2go compiler represents free variables explicitly. When a free variable is demanded by some case-expression, the variable is narrowed similarly to the generator functions of KICS2. This is only possible, because in a case expression we at least know the relevant outermost constructor that are matched on.

   Since variables are represented explicitly, unification can essentially just bind the free variable to a constructor term to avoid the slow or non-terminating

aspect of the KICS2 implementation. However, due to the untyped-ness approach of the implementation in curry2go, it suffers from the same problem of instantiating unconstrained variables that PAKCS has as well. The function `ensureNotFree` is missing the type information to instantiate its argument.

Our Implementation For our implementation we want to adapt the curry2go approach to Haskell. Coincidentally, we already developed such a solution for a slightly different context in Teegen, Prott, and Bunkenburg, 2021. For the rest of this paper, we only need to know that variables in this approach will be instantiated whenever we use `(»=)` within the `Curry` monad. The data type we used in our `CurryMonad` definition for the explicit variable representation looks as follows.

```
data CurryVal a = Val a
                | Narrowable a => Var ID
```

In this definition, a variable is represented by some kind of unique identifier. We also ensure that each variable can be instantiated using narrowing by adding a corresponding type class constraint on the variable constructor. Any bindings (ID $\mapsto$ value) for these variables will be stored in an untyped heap that is part of our `CurryState`.

4 Functional Patterns

Functional patterns Antoy and Hanus, 2005 will not be discussed any further here, because they are implemented correctly for each of the three compilers (at least as far as we know :). They only require a *lazy* variant of unification that is relatively easy to implement.

5 Performance and Memoized Pull-Tabbing

When we look at all Curry extensions and their implementations, it might look like the KiCS2 compiler has everything we want. However, KiCS2 suffers from bad performance in some cases. To see this, let us take a brad look at how the run-time system evaluation ot the compiler works.

To recap, the implementation of KiCS2 is based on pull-tabbing where the non-determinism is represented as a binary tree. To correctly model call-time-choice (e.g. non-strict non-determinism) in KiCS2, choices are labeled with unique identifiers at run-time. When traversing the choice-graph to look for solutions,

KiCS2 uses the identifiers to make consistent choices (left or right subtree) for each of them. Repeatedly looking up if we already made a decision for a given choice identifier can be quite expensive. Even worse, when creating the tree, pull-tab evaluation first moves each non-deterministic choice up to the root of the expression under evaluation and consistency of choices is only checked afterwards. As a consequence "each access to a non-deterministic expression leads to a stepwise shifting of choices towards the root. Thus, multiple accesses to a same non-deterministic expression multiplies the execution time" [Hanus and Teegen, 2020]. An extreme case where this leads to a bad performance can be seen in the following example.

```
addX :: Int -> Int
addX x = let n = anyOf [1..x]
  in sum (replicate x n)

replicate :: Int -> n -> [n]
replicate 0 _ = []
replicate x n = n : replicate (x-1) n

sum :: [Int] -> Int
sum = foldr (+) 0
```

Here, we choose some $n \in [1 \ldots x]$ and repeatedly add it onto itself, leading to x^2 number of pull-tabbing steps and graph nodes. Only x of theses nodes contain actual solutions with consistent decisions for each choice identifier, whereas the other ones are discared due to at least one inconsistent choice.

Hanus and Teegen [2020] fix this performance issue of pull-tabbing with memoization. To that end, they introduce an identifier for each non-deterministic computation branch (called Task) and additionally store the decision made for a given choice and task identifier. Pull-tab steps can now be skipped for choices where a decision has already been made for the current task identifier. This technique is implemented only in the curry2go compiler, but a similar concept has been developed for a purely functional implementation of non-determinism in Haskell by Fischer, Kiselyov, and Shan [2011].

Sharing across non-determinism There is another implementation detail that affects run-time performance. Consider a deterministic function that is expensice to compute, e.g. a list of the first one million prime numbers. If we share the result of this function, we expect the list to be constructed only once and re-used at each shared ocurrence. However, if this sharing ocurrs across non-deterministic

choices, the PAKCS compiler unfortulately duplicates the computation. Both KiCS2 and curry2go avoid this problem. The former just relies on Haskell's graph-based evaluation model for this, whereas the latter requires a small extension to the aforementioned memoized pull-tabbing [Hanus and Teegen, 2020].

6 Memoizing Results Across Alternate Realities

As discussed before, a naive implementation of pull-tabbing like in KiCS2 does not lead to a satisfying performance. Even worse, we cannot even rely on Haskell's sharing at all, because the monadic values we pass around in our implementation are secretly functions due to our use of a state monad.

Instead of copying what the KiCS2 does, we will use a slightly older approach represented by Fischer, Kiselyov, and Shan [2011] to implement non-strict non-determinism using explicit sharing. They define an explicit operator `share :: (Monad m, _) => m a -> m (m a)` that is used whenever a variable is used more than once in an expression. The result of `share` can be bound monadically to yield a *shared* value of the input. If the inner monadic context of the result is evaluated the first time using `(»=)`, the implementation will just evaluate the original argument to `share`, memoize it in the `CurryState` and continue with that result. However, if the inner context is evaluated again, the implementation just looks up the previously memoized result to avoid a repeated evaluation. Thus, in the following example `factorial 100000 :: Curry Integer` will be evaluated only once if we demand the result of `facTuple`.

```
facTuple :: Curry (ListC Integer)
facTuple = share (factorial 100000) >>= \facM ->
  return (ConsC facM (return (ConsC facM (return NilC))))
```

There is only one slight problem with the implementation of `share` given by Fischer, Kiselyov, and Shan [2011]: Deterministic values are not shared across non-determinism, because `share`d values are only memoized in the state of the current non-deterministic branch. As a consequence, a slight modification of the previous example exhibits a two times worse performance than it should have.

```
facTuple :: Curry Integer
facTuple = share (factorial 100000) >>= \facM ->
  facM ? facM
```

Changes in the `CurryState` of the left branch of `(?)` are not allowed to be reflected in the state of the right branch. This seperation of state is important, because

it would be semantically wrong for a language with call-time choice to share a non-deterministic value across non-determinism.

However, if we know that the result of a shared computation was deterministic, we are allowed to share it. Just encapsulating the shared computation to examine the tree structure would sadly be too strict in some cases, for example if one of the sub-trees is non-terminating. Thus, we need a different solution. To that end, we extend our `CurryState` with an identifier for the current non-deterministic branch we are in. Whenever a non-deterministic choice ocurrs, this branch identifier is split into two fresh, unique identifier. Now we can determine if evaluating a `Curry` expression introduced any non-determinism by comparing the branch identifier from before and after evaluating the expression.

To actually memoize deterministic values across non-determinism in the implementation of `share`, we need to use some impure `IO`. This `IO` is captured in our pure interface using `unsafePerformIO`, which we argue is safe to use in our case. On a high level, the implementation of `share` now looks as follows.

```
share :: Curry a -> Curry (Curry a)
share ma = do
  i <- freshID
  let detRef = unsafePerformIO (newIORef Nothing)
  return $ do
    maybeB <- findBindingForID i
    case maybeB of
      Just a  -> return a
      Nothing -> case unsafePerformIO (readIORef detRef) of
        Just a  -> return a
        Nothing -> do
          bID1 <- getBranchID
          a <- m a
          bID2 <- getBranchID
          if bID1 == bID2
            then unsafePerformIO (writeIORef detRef (Just a))
                   `seq` return a
            else insertBindingForID i a
                   >> return a
```

In the outer monad layer, we first generate a unique ID for the value we want to share and an `IORef` in which to save a potential deterministic result. In the inner monad layer, we check if there is either a binding for our ID in the `CurryState` or in our memory cell. Only if both of them contain no binding, we evaluate the

argument to share and compare the branch ID from before and after. If it did not change, we just insert the value into the memory cell, or otherwise into the `CurryState`.

7 Weak Encapsulation with Set-Functions

Encapsulating non-determinism is important for applications where one wants to compare different solutions to a problem and process them further. The problem with encapsulating all non-determinism in an expression `e` of a lazy language like Curry is that `e` might share a sub-expression which is defined outside of `e` and might be non-deterministic. *Strong* encapsulation requires all non-determinism that ocurrs during the evaluation to be encapsulated. In contrast, *weak* encapsulation should only capture the non-determinism originating from inside the encapsulated expression and not from outside. To that end, Antoy and Hanus [2009] introduce *set functions* as a way to weakly encapsulate non-determinism. Here, we will use a family of functions $\texttt{set}_n$ `::` $(\tau_1$ `->` $\tau_n)$ `->` τ_1 `->` τ_{n-1} `->` $[\tau_n]$. As an example for weak encapsulation, consider the following two expressions.

```
e1, e2 :: [Bool]
e1 = let xs1 = [True ? False] in set1 anyOf xs1
e2 = let xs2 = [True,  False] in set1 anyOf xs2
```

Note that the first expression passes a singleton list with a non-deterministic element to the set function of `anyOf`, whereas the second expression uses a list with two deterministic elements. Since `anyOf` introduces no non-determinism for a list of length one and the non-determinism originates on the outside of the encapsulation, the first expression evaulates to the two results `[True]` and `[False]`. For the second expression, the set-function captures the non-deterministic choice between the list elements and is thus equivalent to `[True, False]`.

There is one easy way to implement weak encapsulation that is taken by both PAKCS and curry2go. The set function first evaluates all of its arguments to (full) normal form to trigger any non-determinism ocurring outside the function to be captured. Finally, one only has to evaluate the function for each of the non-deterministic values of its arguments and capture any ocurring non-determinism. However, evaluating all arguments beforehand might make the set-function more strict than the original, un-encapsulated function. Thus, `set2 const True failed` does not produce any result in PAKCS and curry2go, although the expresion should be equal to `[True]` according to the specification of set functions.

A different approach is taken by KiCS2, where each choice and failure is annotated (at runtime) with the information, whether or not it ocurred under a set-function. To support arbitrary nesting of set-functions, each set-function needs to know which choices/failures it shoud capture and which to leave as-is for set-function on an outer nesting level. Therefore, the information at a choice is precisely the nesting level of set functions at its time of creation. This allows the set-functions in KiCS2 to look at the choice structure only after evaluating the function without strictly evaluating any arguments that are not actually required. As a consequence, set-functions in KiCS2 are less strict than in PAKCS or curry2go. Thus, we will use the same approach and adapt it to our needs.

As discussed before, each non-deterministic choice/failure happens at a certain nesting level of set functions. Since set functions offer a way to recover from a failed computation, we also have to ensure to keep the `CurryState` of a failure. Therefore, we instantiate the annotations of our previous `Tree` type as follows and use this throughout our implementation.

```
type Level = Int
type SetTree a = Tree Level (Level, CurryState) a
```

Next, we will take a look at the implementation of `set1`. Any other set function can be implemented in a similar fashion. First we start by getting the current level and increasing it by one. Afterwards we need to capture the result of applying the input function `f` to its argument(s). Here we need to ensure that the level is reset before evaluating the argument. This is done in `setLevelC`, which traverses its argument data structure and sets the level accordingly, even in the effectful/lifted components of a data type. Before encapsulating the result of `f` for the given level, we compute the ground normal form to ensure that we also trigger any deeply ocurring non-determinism in the lifted data components.

```
set1 :: _ => (Curry a -> Curry b) -> Curry a -> Curry (ListC b)
set1 f arg = do
  lvl <- currentLevel <$> get
  let newLevel = succ lvl
  modify (\s -> s { currentLevel = newLevel })
  captureWithLvl newLevel $ groundNormalForm $ f
    (setLevelC lvl arg)
```

The implementation of `captureWithLvl` is a bit too long to show, but it performs the following steps.

1. Evaluate the `Curry` argument to get its non-determinism tree. Also make sure to keep the state in case of success and failure.

2. Traverse the tree and decide:
   - ▻ On a Leaf, collect the result into a singleton list.
   - ▻ On a failure with the captured level, return an empty list.
   - ▻ On a failure with a different level, replicate the failure.
   - ▻ On a node with the captured level, recursively collect the results of the left and right branch. The results are concatenated.
   - ▻ On a node with a different level, also collect the results of the left and right branch. This time, however, create a choice node between the left and right results with the same level as the choice we encountered.

Testing this implementation seems to be promising at first. Sadly, one can craft an example where this approach produces duplicate and sometimes even incorrect solutions. A disaster! To understand what is happening, we need to look at the following example in Curry notation.

```
disasterInstantiation :: Bool -> Int
disasterInstantiation x' =
  let left  = if x then 1 else 2
      right = if x then 3 else 4
  in left ? right

disasterSet :: [Int]
disasterSet = set1 disasterInstantiation (True ? False)
```

For `disasterSet`, we would expect two results, since we pass in a choice between two values. Both results should be a list of two elements ([1,3] and [2,4]). However, our implementation additionally returns the nonsensical results [1,4] and [2,3]. Both of them would require an inconsistent choice for `x` to be made. Evaluating a set-function should apparently let some information about non-deterministic pass between the left and right branch of a captured choice.

To implement this, we adapt the implementation of `(?)`, such that it behaves differently in the context of a set function. Although it seems odd, in the context of a set function we do not create a non-deterministic node at the root of the tree with the left and right argument of `(?)`, but instead traverse through the tree of the left argument and extend it at every leaf and failure with the tree that results from the right argument. This way, we can even pass the state at every leaf and failure within the left tree to the computation of the right tree. Note that `(?)` under set functions is now a bit stricter since it has to produce (parts of) the left tree before creating the right tree. However, if the left argument of `(?)` does not produce (parts of) a result before looping forever, it is not possible

for the set function to produce any result in other implementations as well since we always examine the left sub-tree of a node before the right one. Only in an implementation with a *fair* search strategy where both sub-trees are considered simultaneously via multiple threads, this could make a difference.

8 Side Note: Why a Monadic Implementation Matters

There are various reasons why we want a monadic implementation of Curry. First of all, monadic liftings for call-by-value and call-by-name have been known for a long time [Wadler, 1990]. They have been used extensively for translations into proof-assistant languages, for example by Abel et al. [2005]. We are also currently working on an unpublished paper where we implement a monadic lifting as a GHC plugin for the compiler, such that Haskell can be augmented by any implicit effect that can be implemented as a monad. This is the main reason why we started looking into a monadic model of curry, such that we can implement a compiler for Curry on top of GHC. Part of this plugin is described in my master's thesis [Prott, 2020].

9 Conclusion and Future Work

In conclusion, each of the current Curry compilers has at least one flaw in some aspect of their implementation. A summary of these flaws is given in Table 1. In this paper, we briefly discussed these implementations and showed why they do or do not work. Afterwards, we presented how we combined and extended existing solutions for each of the problems such that we get a monadic model of Curry.

In the future, we are planning to implement an actual standalone compiler based on this implementation as well as a compiler as a GHC plugin. There are also some small performance problems in some cases due to extensive garbage collection that we still have to examine. Of course, we also plan on doing extensive benchmark comparisons with the other compilers. A preliminary test indicated that we are faster than curry2go, PAKCS and KiCS2 in most cases. We also want to adopt a different part of the curry2go compiler that we did not talk about in this paper: its implementation of a fair search for solutions.

Table 1. Features and their support in KICS2, PAKCS and curry2go

Feature	KICS2	PAKCS	curry2go
Free Variables, Unification	Search space can be huge	Instantiation of unconstrained variables impossible	
Set Functions	Semantically correct, but has a bug	Too strict, since all arguments of the set function are forced	
Functional Patterns	Supported without problems		
Memoized Pull-Tabbing	Nope	n/a	Yes
Sharing Across ND	Yes	Nope	Yes
Search Strategies	breath-first, depth-first	backtracking	breath-first, depth-first, fair

References

Abel, Andreas et al. (2005). "Verifying haskell programs using constructive type theory". en. In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell - Haskell '05*. Tallinn, Estonia: ACM Press, pp. 62–73. ISBN: 978-1-59593-071-2. DOI: 10.1145/1088348.1088355.

Antoy, Sergio (July 2011). "On the correctness of pull-tabbing". en. In: *Theory and Practice of Logic Programming* 11.4-5, pp. 713–730. ISSN: 1471-0684, 1475-3081. DOI: 10.1017/S1471068411000263.

Antoy, Sergio and Michael Hanus (2005). "Declarative Programming with Function Patterns". In: *Logic Based Program Synthesis and Transformation*. Ed. by Patricia M. Hill. Vol. 3901. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 6–22. ISBN: 978-3-540-32654-0 978-3-540-32656-4. DOI: 10.1007/11680093_2.

Antoy, Sergio and Michael Hanus (2009). "Set functions for functional logic programming". en. In: *Proceedings of the 11th ACM SIGPLAN Conference on*

Principles and Practice of Declarative Programming - PPDP '09. Coimbra, Portugal: ACM Press, p. 73. ISBN: 978-1-60558-568-0. DOI: 10.1145/1599410.1599420.

Böhm, Jonas, Michael Hanus, and Finn Teegen (2021). "From Non-determinism to Goroutines: A Fair Implementation of Curry in Go". In: *Proceedings of the 23rd Symposium on Principles and Practice of Declarative Programming*. Tallin (Estonia): ACM.

Braßel, Bernd et al. (2011). "KiCS2: A New Compiler from Curry to Haskell". In: *Functional and Constraint Logic Programming*. Ed. by Herbert Kuchen. Vol. 6816. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–18. ISBN: 978-3-642-22530-7 978-3-642-22531-4. DOI: 10.1007/978-3-642-22531-4_1.

Fischer, Sebastian, Oleg Kiselyov, and Chung-Chieh Shan (Sept. 2011). "Purely functional lazy nondeterministic programming". en. In: *Journal of Functional Programming* 21.4-5, pp. 413–465. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796811000189.

Hanus, Michael, ed. (Mar. 2018). *PAKCS: The Portland Aachen Kiel Curry System*.

Hanus, Michael (2021). "Lightweight Declarative Server-Side Web Programming". en. In: *Practical Aspects of Declarative Languages*. Vol. 12548. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 107–123. ISBN: 978-3-030-67437-3 978-3-030-67438-0. DOI: 10.1007/978-3-030-67438-0_7.

Hanus, Michael, Herbert Kuchen, and Juan José Moreno-Navarro (1995). "Curry: A Truly Functional Logic Language". In: *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*. Portland (USA), pp. 95–107.

Hanus, Michael and Finn Teegen (2019). "Adding Data to Curry". en. In: *Declarative Programming and Knowledge Management*. Ed. by Petra Hofstedt et al. Vol. 12057. Cham: Springer International Publishing, pp. 230–246. ISBN: 978-3-030-46713-5 978-3-030-46714-2. DOI: 10.1007/978-3-030-46714-2_15.

Hanus, Michael and Finn Teegen (2020). "Memoized Pull-Tabbing for Functional Logic Programming". en. In: *Functional and Constraint Logic Programming*. Ed. by Michael Hanus and Claudio Sacerdoti Coen. Vol. 12560. Bologna, Italy: Springer International Publishing, pp. 57–73. ISBN: 978-3-030-75332-0 978-3-030-75333-7. DOI: 10.1007/978-3-030-75333-7_4.

Jones, Mark P. (1995). "Functional Programming with Overloading and Higher-Order Polymorphism". In: *Advanced Functional Programming*. Ed. by Gerhard Goos et al. Vol. 925. Berlin, Heidelberg: Springer-Verlag, pp. 97–136. ISBN: 978-3-540-59451-2. DOI: 10.1007/3-540-59451-5_4.

Liang, Sheng, Paul Hudak, and Mark Jones (1995). "Monad Transformers and Modular Interpreters". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. New York, NY,

USA: Association for Computing Machinery, pp. 333–343. DOI: 10.1145/199448.199528.

Marlow, Simon, ed. (2010). *Haskell 2010 Language Report*.

Prott, Kai-Oliver (2020). "Extending the Glasgow Haskell Compiler for functional-logic Programs with Curry-Plugin". Master's Thesis. Christian-Albrechts-Universität zu Kiel.

Teegen, Finn, Kai-Oliver Prott, and Niels Bunkenburg (2021). "Haskell^{-1}: Automatic function inversion in haskell". In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*. Haskell 2021. New York, NY, USA: Association for Computing Machinery, pp. 41–55. ISBN: 978-1-4503-8615-9. DOI: 10.1145/3471874.3472982.

Wadler, Philip (1990). "Comprehending monads". en. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming - LFP '90*. Nice, France: ACM Press, pp. 61–78. ISBN: 978-0-89791-368-3. DOI: 10.1145/91556.91592.

Combinatory Differentiation: From Category Theory to High-Performance Automatic Differentiation

(Extended abstract)

Fritz Henglein
henglein@diku.dk
Department of Computer Science, University of Copenhagen (DIKU)

Abstract

Automatic differentiation (AD) is the discipline of computing derivatives for functions given by programs. It is used in gradient-based optimization, neural networks, probabilistic inference and has numerous applications in computer vision, natural language processing, computational science, bioinformatics, quantitative finance, etc. For example, backpropagation, which is used in machine learning to train neural networks, is an instance of reverse mode AD applied to scalar functions.

Intuitively, *forward-mode* AD works by replacing a real number x—in practice a floating point number—by a dual number $x + y \cdot dx$, which adds a *differential* $y \cdot dx$ to x, where dx is an indeterminate variable; in particular, a dual number is represented as the pair (x, y) at run time. The program is then executed with dual numbers instead of reals. Dual numbers support addition, scalar multiplication with a real number, multiplication and division. For library functions l on real numbers a corresponding library of derivatives l' is required such that l can be computed on dual numbers by

$$l(x + y \cdot dx) = l(x) + (l'(x) \cdot y) \cdot dx.$$

The output of a sequentially executed program f on dual number $x + dx$ then yields the derivative of f at x in the second component of the output: $f(x + dx) = f(x + 1 \cdot dx) = f(x) + (f'(x) \cdot 1) \cdot dx = f(x) + f'(x) \cdot dx$. This can be generalized to functions $f : \mathbb{R}^m \to \mathbb{R}^n$ of multiple scalar input and output variables: Forward AD computes the partial derivative $\frac{\partial f}{\partial x}$ at a given input vector v_0 for one particular scalar input variable x at a time. In particular, it computes the Jacobian $\in \mathbb{R}^{m \times n}$ of f at v_0 one column at a time. *Reverse-mode* AD computes the Jacobian one row at a time at the expense of building a trace of the

operations executed when evaluating f on v_0 and processing the trace in reverse order. If v_0 is changed, all computations, including trace construction, need to be repeated.

These traditional AD approaches and their combinations are limited to functions $f : \mathbb{R}^m \to \mathbb{R}^n$. Execution is sequential. They furthermore suffer from space explosion for functions on high-dimensional vector spaces due to using matrices to represent derivatives. In machine learning applications functions routinely operate on million- or even billion-dimensional vector spaces, however.

We show that AD can be performed for arbitrary-dimensional vector spaces[1] such that parallelism in the input program is preserved and space blow-up is avoided. Conceptually, given a program $f : V \to W$ and an input $v_0 \in V$, we compute the value and its derivative at v_0 by interpreting f to yield not only the output value, but also a compact symbolic representation of the *Fréchet derivative* of f at v_0, which is a linear map $l \in Hom(V, W)$ such that

$$f(v_0 + dv) \approx f(v) + l(dv),$$

that is such that

$$\lim_{||dv||_V \to 0} \frac{||f(v + dv) - (f(v) + l(dv))||_W}{||dv||_V} = 0$$

where $||\ldots||_U$ is the norm that comes with the Banach space U.

Fréchet derivatives generalize Jacobians; in particular, they need not be represented as matrices, but can be arbitrarily more compact. For reverse mode AD we compute the adjoint $l^{adj} \in Hom(W, V)$ of l symbolically; this corresponds to transposing the Jacobian matrix without, however, ever manifesting input or output as a potentially huge matrix. Finally, l or l^{adj} is applied according to the specific use. In deep learning, f is a scalar function, that is $W = \mathbb{R}$, and thus $l^{adj} \in Hom(\mathbb{R}, V)$. Backpropagation consists of applying l^{adj} to 1, which yields $l^{adj}(1)$, the gradient of f at v_0.

We show that almost all the computations can be performed *symbolically*, for an indeterminate input v, without expression swell. This avoids the need for repeated computation for different input vectors and amounts to *symbolic differentiation* of functions that may employ second-order functions (functional operators) on not only scalar and vector types, but also spaces such as tensor products and dependent (bi)products, which generalize higher-rank tensors. Symbolic differentiation is based on generalized differentiation rules for sequential and parallel composition as well as constant, linear and bilinear functions. They are most easily formulated for functions in *combinatory* (also called *point-free*) form.

For example, a layer of a neural network is parameterized by a weight matrix $W \in \mathbb{R}^{m \times n}$ and bias vector $b \in \mathbb{R}^m$ and takes a data vector $v \in \mathbb{R}^n$ as input, where $m, n \gg 0$ may be in the millions or billions. It can be defined in a single line by

$$g(W, b, v) = \text{map}\, h\, (W \star v + b)$$

where map applies a function to each element of a vector, $h : \mathbb{R} \to \mathbb{R}$ is an activation function such as tanh, $\star$ is matrix/vector multiplication and $+$ is vector addition. It can

[1]With sufficient structure for differentiation to make sense such as a Banach space

be evaluated using data-parallel implementations of these operations, for example in the purely second-order functional programming language Futhark, which generates high-performance GPU- and multicore code.

The Fréchet derivative of $g_v(W, b) = g(W, b, v)$ is symbolically computed to be

$$g'_v(W, b) = \Delta(\text{map}\, h'\, (W \star v + b)) \bullet ((\star v) \bullet \pi_1 + \pi_2)$$

where Δ is zip-apply, which applies a vector of functions to a corresponding vector of values; $\bullet$ is composition of linear maps; $(\star v)$ is section notation for the function $(\star v)(W) = W \star v$; $+$ is linear map addition; and π_1, π_2 project out the first, respectively second argument of a pair. Note that $\Delta(\text{map}\, h'\, (W \star v + b)) \bullet ((\star v) \bullet \pi_1 + \pi_2)$ denotes a linear map in $Hom(\mathbb{R}^{m \times n} \times \mathbb{R}^m, \mathbb{R}^m)$ for any $m, n \in \mathbb{N}$. Representing this linear map as a matrix would require a forbidding $(m \cdot n + m) \cdot m$ entries. Finally, the adjoint l^{adj} of $l = g'_v(W, b)$ can be computed symbolically from l to be

$$l^{adj} = \langle (\otimes v), id \rangle \bullet \Delta(\text{map}\, h'\, (W \star v + b))$$

where $\otimes$ is the outer product of vectors defined by $((\otimes v)(w))_{i,j} = (w \otimes v)_{i,j} = w_i \cdot v_j$ and $\langle f, g \rangle(x) = (f(x), g(x))$. Note that l^{adj} is a linear map in $Hom(\mathbb{R}^m, \mathbb{R}^{m \times n} \times \mathbb{R}^m)$, expressed as a one-liner in a combinatory language for composing linear maps using data parallel operations. This is the code that results in updates to the initial weight matrix and bias vector during backpropagation through the neural network. It is derived by symbolic differentiation and symbolic computation of adjoints, and it retains use of data-parallel operations.

The above describes some of the ongoing joint work on high-performance automatic differentiation performed by the Functional High-Performance Computing and Deep Probabilistic Programming research groups at DIKU in collaboration with partners at Utrecht University and Google Mind.

STUBBER: Compiling Source Code into Bytecode without Dependencies for Java Code Clone Detection

André Schäfer
andre.schaefer@uni-jena.de
Friedrich Schiller University Jena, Jena, Germany

Wolfram Amme
wolfram.amme@uni-jena.de
Friedrich Schiller University Jena, Jena, Germany

Thomas S. Heinze
thomas.heinze@dlr.de
German Aerospace Center, Jena, Germany

In recent years, many clone detection tools for Java have been introduced. On the one hand, many of these tools work with Java source code and can therefore be conveniently evaluated with the modern evaluation framework BigCloneEval and the benchmark BigCloneBench. On the other hand, certain clone detectors deliberately address bytecode as input and cannot be applied to Java source code. Simple compilation often does not solve the problem due to missing required dependencies, as in the case of the Java files in BigCloneBench. Therefore, we present the STUBBER tool for compiling Java source code into bytecode without dependencies. We can show that STUBBER can successfully generate bytecode for over 95% of all Java source files and 92.5% of all code clones contained in BigCloneBench. Consequently, the evaluation of bytecode-based clone detectors with BigCloneEval on BigCloneBench becomes possible and such tools can thus also be compared with source code-based clone detectors.

Verwendung von Klonerkennung zum Auffinden von Signaturen von Malware-Familien: Eine Fallstudie über FinSpy

Nils Scheidweiler
Nils.Scheidweiler@uni-jena.de
Friedrich-Schiller-Universität Jena

André Schäfer
Andre.Schaefer@uni-jena.de
Friedrich-Schiller-Universität Jena

Wolfram Amme
Wolfram.Amme@uni-jena.de
Friedrich-Schiller-Universität Jena

Thomas S. Heinze
Thomas.Heinze@dlr.de
Deutsches Zentrum für Luft- und Raumfahrt

Abstract

Bei der Entwicklung von Malware wird oftmals existierender Code wiederverwendet. Die Suche nach Code, der bekannter Malware ähnelt, kann daher für die Malwaredetektion eine vielversprechende Strategie sein. Nach einer Vorstellung verschiedener Techniken zur Malwaredetektion analysieren wir die Verwendung des Klon-Detektors StoneDetector zum Auffinden von Android-Malware. StoneDetector erzeugt aus Quellcode die Kontrollflussgraphen und wandelt diese in Dominatorbäume um. Durch die Extraktion der Pfade von den Blättern zur Wurzel eines Dominatorbaums werden Beschreibungsmengen gebildet. Mithilfe von bösartigen Samples einer Malwarefamilie und gutartigen Samples werden Beschreibungsmengen gesucht, die in den meisten bösartigen Samples, aber nicht in den Gutartigen vorkommen und diese Beschreibungsmengen als Signatur der Malwarefamilie extrahiert. Die Machbarkeit des Ansatzes wird anhand einer Fallstudie mit Android-Samples der FinSpy-Malwarefamilie gezeigt. Es werden 31 FinSpy und 20 gutartige Samples in eine Trainings- und eine Testmenge unterteilt und die Signatur mithilfe der Samples der Trainingsmenge gebildet. Für die Beurteilung wird die FinSpy Signatur in bösartigen und gutartigen Samples der Testmenge gesucht. Es kann gezeigt werden, dass mit dem Ansatz alle getesteten bösartigen und gutartigen Samples richtig klassifiziert werden.

A formal type system and type checker for Erlang

Albert Schimpf
a_schimpf12@cs.uni-kl.de
TU Kaiserslautern

Annette Bieniusa
bieniusa@cs.uni-kl.de
TU Kaiserslautern

Stefan Wehr
stefan.wehr@hs-offenburg.de
Hochschule Offenburg

Abstract

Erlang is a dynamically typed functional programming language used to built concurrent and distributed applications. The language closely implements the actor model where processes are spawned dynamically and communicate primarily by message passing. Among typical functional language features, message passing and dynamic function invocation makes it difficult to design a complete static type system for Erlang.

One such example is the behavior module `gen_statem`, which encapsulates the generic behavior of a state machine. A module can extend such a state machine behavior and implement the business logic while relying on the generic functionality of the library. The behavior allows function definitions to be the states of the state machine, which in turn means that the state machine dynamically uses function names as states. The return type of such a state function can be a tuple consisting of atoms `next_state, state_name, Data` , where the type information to type those functions is completely lost.

Current approaches to introduce typing in Erlang include success typing **dialyzer**, gradual typing[1], subtyping**marlow**, type inference based on Hindley-Milner **etc**, bidirectional typing **bidirect**, and session types for the message passing primitives **session**. All of the approaches either focus on specific features, fail to capture common use cases, cover a small sub-set of the language, or do not work with large Erlang code bases. In our talk, we will present the most challenging features of the Erlang language and its type language. We argue that the starting point for a successful typing approach is the Erlang language coupled with the OTP framework with its associated usage patterns. We discuss how their interplay requires adaptations of different advanced type features such as equi-recursive types, union types, occurrence typing, and intersection types.

[1] https://github.com/josefs/Gradualizer

Effect Systems in Haskell

A Case Study

Hannes Siebenhandl
Compilers and Languages Group
TU Wien
hannes.siebenhandl@tuwien.ac.at

Abstract
Effect systems have been introduced in order to describe computational effects a program can have, usually by tracking allowed effects in the type system. In our case study, we investigate three state-of-the-art Haskell libraries supporting algebraic effect systems, namely `polysemy`, `freer-simple` and `fused-effects`. The goal of our case study is to identify the strengths and limitations of these libraries from a usability perspective. To this end we introduce a metrics considering performance, expressiveness, and accessibility as parameters. The main findings of our case study are that none of these libraries uniformly outperforms the others, and that none of them satisfies all of our usability criteria to the desirable extent. In spite of the simplicity of our approach to evaluate the usability of a library, our findings highlight important strengths and limitations of popular effect system libraries in Haskell and identify issues of future research to further enhance their usability and power.

1 Background

Effect systems are a vibrant research topic since they have been first proposed by Lucassen and Gifford [1988]. They allow fine-grained control over the capabilities of functions and procedures, such as side effects, by tracking effects directly in the type system. *Algebraic* effect systems, as defined by Plotkin and Pretnar [2009], extend classical effect systems by adding handlers for computational effects which are representable by a chosen algebraic theory. There are mainstream programming languages which adopted limited forms of effect systems. E.g., Rytz, Odersky, and Haller [2012] showed that the checked exceptions in Java form an effect system. However, only few languages provide support for user-defined algebraic effect systems. Haskell, a lazy, purely functional programming

language with an industrial strength compiler named GHC [1] (Glorious Haskell Compiler), is flexible enough to allow library implementations, usually built on top of free monads, as introduced by Trnková et al. [1975], of algebraic effect systems without special compiler support.

2 Case Study

In our case study, we consider `polysemy`, `freer-simple`, and `fused-effects`, which are three state-of-the-art Haskell libraries supporting algebraic effect systems. In order to compare their relative merits, we introduce metrics considering

- ▻ performance,
- ▻ expressiveness,
- ▻ and accessibility

as parameters.

We measure *performance* of a library in terms of the metrics that are available from a subset of the microbenchmark-suite `effect-zoo` [2]. The baseline is provided by implementing and executing an equivalent benchmark for `mtl`. The measurement of *expressiveness* is based on the different control flow structures that can be expressed with a library, e.g., whether coroutines are supported and whether it is possible to have scoped effect handlers i. e., *higher order effects*. The number of supported constructs is obtained from the websites of the respective libraries. *Accessibility* is, in principle, a subjective metric. In order for the reader to get a grasp of our measurements, we combine number of lines of code as an objective metric while judging how intuitive we think type-error messages produced by the compiler are when an effect is not handled, an effect's context is ambiguous or a function is missing an effect constraint.

3 Findings

Performance. Table 1 summarises our findings when examining performance of the three libraries for the problems *CountDown* and *FileSizes*. The former benchmark takes as input a number and counts down until it reaches the value zero using a `State` effect, thus the efficiency of the effect system's encoding is measured, as no other operations have any impact on the run-time. *FileSizes*

[1] https://www.haskell.org/ghc/

[2] https://github.com/ocharles/effect-zoo/tree/eff-and-polysemy

Table 1. Average performance in milliseconds and lines of code (LoC) for the benchmarks **CountDown** and **FileSizes**. For the *relative* performance, we take `mtl`'s execution time as the baseline.

Performance	CountDown			FileSizes		
	time	relative	LoC	time	relative	LoC
`mtl`	0.029ms	1.00	8	0.163ms	1.00	41
`polysemy`	2.040ms	70.34	8	0.383ms	2.35	27
`fused-effects`	0.126ms	4.34	8	0.194ms	1.19	53
`freer-simple`	0.435ms	15.00	8	0.194ms	1.19	31

Table 2. Overview of the *expressiveness* of effect system libraries in Haskell.

Expressiveness	**Coroutines**	**Higher Order Effects**
`mtl`	✓	✓
`polysemy`	✗	✓
`fused-effects`	✗	✓
`freer-simple`	✓	✗

takes a number of file paths and reads their file size from available metadata. It is implemented using a custom effect and is a suitable benchmark for how many lines of code are required to add a custom effect. However, `polysemy` suffers performance-wise even when the effect system is barely used. From the micro-benchmarks, we can see that `fused-effects` outperforms its competitors, but is still notably slower than the reference implementation.

Expressiveness. The expressiveness of effect systems can be paramount for their usability, but there are often trade-offs in terms of performance and accessibility. The library `freer-simple` is restricted to first-order effects and is therefore not capable of expressing higher-order effects, such as the common *acquire-release* pattern, while its competitors provide first-class support for it. On the other hand, and contrary to `fused-effects` and `polysemy`, `freer-simple` is capable of expressing *coroutines*, i. e., yielding from its current computation and resuming it at another time. An overview of the findings is shown in Table 2, showing that no effect system library has the same capabilities as `mtl`, and none of them is strictly superior to any other library.

Accessibility. Regarding *accessibility*, we look at the error messages produced by the compiler. While type-errors are produced by GHC, library authors may

Table 3. *Accessibility* of effect system libraries in Haskell, measured by the quality of the error messages for the cases *missing an effect handler* (MEH), *missing an effect context* (MEC) and *ambiguous context* (AC). Possible values are `Helpful` (H), `Complicated` (C), `Exposes Internal Types` (EIT) and `Not Representable` (✗)

Accessibility	**MEH**	**MEC**	**AC**
`mtl`	C	C	✗
`polysemy`	H	H	H
`fused-effects`	C	C	C
`freer-simple`	C	EIT	C

identify common mistakes and provide custom type-errors to help guide a user. The authors of `polysemy` identify common mistakes and provide helpful error messages for all examined scenarios, moreover they provide a GHC source-plugin which automatically fixes trivial mistakes. However, competitors take no additional precaution to help guide users in case of mistakes, showing error messages that contain internal types, e.g. types users utilised themselves. Additionally, the number of lines of code required for writing effect systems is the lowest for `polysemy` and the highest for `fused-effects`, as can be seen in Table 1. In Table 3 we show our findings for the accessibility of the individual libraries. In conclusion, `polysemy` outperforms its competitors when it comes to *accessibility* by requiring the least amount of code, even compared to `mtl`, and providing better error messages than any other library.

4 Conclusion, Future Work

We compared three Haskell libraries that allow developers to define and use algebraic effect systems for their performance, expressiveness, and accessibility. Our findings show that no library uniformly outperforms the others, `fused-effects` is closest to the performance of the reference implementation in `mtl`, but requires the most boilerplate when defining custom effects. The performance overhead `polysemy` incurs on code that relies heavily on effects cannot be neglected, but `polysemy` requires fewer lines of code to define effects and is more accessible. The library `freer-simple` is between `fused-effects` and `polysemy` in regard to performance and accessibility, but has a unique feature in this comparison: the support of *coroutines*.

The obvious question is, can we have a library that combines the advantages of all three examined libraries? In particular, performance competitive to `mtl`, with

the expressiveness of `polysemy`/`fused-effects` and `freer-simple`, while being as accessible as `polysemy`. Being able to have coroutines as well as higher order functions is a rewarding topic for further research. Performance relies a lot on what optimisations are applied by the compiler. Alexis King [3] demonstrates why effect systems based on free monads often yield sub-par performance in her talk and proposes to use delimited continuation. Rompf, Maier, and Odersky [2009] show that this approach is promising in practice. Moreover, Alexis King provides an algebraic effect system library `eff` [4] based on delimited continuations. However, `eff` seems to misbehave in the face of higher order effects [5]. Additionally, `eff` can currently not work with off-the-shelf GHC versions and requires a fork. Working on fixing possible soundness issues in `eff` as well as helping GHC to optimise effect systems more efficiently are further research topics of our own future work.

References

Lucassen, John M and David K Gifford (1988). “Polymorphic effect systems”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 47–57.

Plotkin, Gordon and Matija Pretnar (2009). “Handlers of algebraic effects”. In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 80–94. ISBN: 978-3-642-00590-9.

Rompf, Tiark, Ingo Maier, and Martin Odersky (Aug. 2009). “Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform”. In: *SIGPLAN Not.* 44.9, pp. 317–328. ISSN: 0362-1340. DOI: 10.1145/1631687.1596596. URL: https://doi.org/10.1145/1631687.1596596.

Rytz, Lukas, Martin Odersky, and Philipp Haller (2012). “Lightweight polymorphic effects”. In: *ECOOP 2012 – Object-Oriented Programming*. Ed. by James Noble. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 258–282. ISBN: 978-3-642-31057-7.

Trnková, Věra et al. (1975). “Free algebras, input processes and free monads”. English. In: *Commentat. Math. Univ. Carol.* 16, pp. 339–351. ISSN: 0010-2628.

[3] https://www.youtube.com/watch?v=0jI-AlWEwYI

[4] https://github.com/hasura/eff

[5] https://github.com/hasura/eff/issues/12

Generalizing Java Type Unification

Adding bounded type variables to unification output

Andreas Stadelmeier
a.stadelmeier@hb.dhbw-stuttgart.de
DHBW Stuttgart

Martin Plümicke
pl@dhbw.de
DHBW Stuttgart

Abstract

The Java-TX project develops a type inference algorithm for Java. Type inference enables the user to omit type annotations while coding. These types will be later automatically inserted by our type inference algorithm.

To calculate the correct typing for a given typeless Java program, the type inference algorithm has to solve a set of type constraints. These constraints are derived from the typeless input program.

The main part of the Java-TX type inference algorithm is the **Unify** algorithm, which solves those type constraints.

This paper presents changes which add generic type variables to the output of our type inference algorithm. These changes can help generating more compact and readable type solutions and reduce the calculation time of our type inference algorithm.

1 Introduction

Java-TX (TX standing for **T**ype e**X**tended) is an extension of Java Plümicke, 2021. The predominant new features are global type inference and real function types for lambda expressions. These function types are introduced in a similar fashion as in Scala but additionally integrated into the Java target-typing as proposed in a so-called strawman approach Plümicke and Stadelmeier, 2017; Reinhold, 2009. These extensions lead to a new, more powerful overloading mechanism, which means that the type of a method declaration could be an intersection of method types. Furthermore, there is a principal type property for Java-TX methods.

One of the first global type inference algorithm given by Damas and Milner Damas and Milner, 1982 reduces the global type inference problem for a small functional programming language to the unification problem which is solved by Robinson Robinson, 1965 and Martelli and Montanari Martelli and Montanari, 1982.

Our global type inference algorithm reduces the type inference problem to a type unification problem Plümicke, 2015 which is given by: For a set of constraints $\overline{\theta_1 \lessdot \theta_2}$ a substitution is demanded, such that $\overline{\sigma(\theta_1) <: \sigma(\theta_2)}$, where <: is the Java subtyping relation and $\lessdot$ represents the constraints.

As a first approach we gave for the type system of GJ Bracha et al., 1998 the type unification algorithm Plümicke, 2004. GJ allows only invarant parameters. This means given two classes C and D with

class C<$\overline{X}$> extends D<$\overline{X}$> { ... }

for a pair C<$\overline{\theta}$> $\lessdot$ D<$\overline{\theta'}$> that θ and θ' have to unified such that $\sigma(\theta) = \sigma(\theta')$. One the one hand this restrictions simplifies the algorithm and on the other the number of results grows independantly of the nesting deepness only propotional to the number subtypes.

In Plümicke, 2009 we extended the type unification algorithm to the core type system of Java 5.0 with use-site variance type parameters introduced by so-called wildcards.

Both approaches solves inequations $a \lessdot \theta$ as well as $\theta \lessdot a$, where a is type variable and θ is no type variables, by determining all sub- and all super-types of θ, respectively, and multiplying the set of constraints. This leads to a multitude of solutions and to enourmous growing of the runtime.

At the moment our type sytems allows indeed type variables bounded by other type variables, but no type variables bounded by reference types.

In this paper we generalize the type unification algorithm such that results can contain type variables bounded by reference types. This means that $a \lessdot \theta$ are no longer resolved. The lower bounds $\theta \lessdot a$ are still resolved as Java allows no lower bounds of type variables. Scala allows lower bounds such that the result of a type unification algorithm for the Scala type system could contain loweer bounded type variables, too.

2 Motivation

Originally our **Unify** algorithm calculates every possible solution for a given set of constraints. This leads to problems when trying to set in the principal type.

Lets see the following example:

```
class Example extends Object{
    Object f;
    method(list){
        f = list.get();
        return list.get();
    }
```

```
}
```

Lets assume this yields the constraints: $p \lessdot$ `List<b>`, $r \lessdot$ `List<b>`, $b \lessdot$ `Object` with p being the type placeholder for the parameter `list` and r being the placeholder for the return type of the method `method`.

The original **Unify** algorithm returns two different possibilities for the type variable b. It can either be `Object` or `Example`. This results in the following two possible solutions:

```
    List<Object> method(List<Object> list){
        f = list.get();
        return list;
    }

    List<Example> method(List<Example> list){
        f = list.get();
        return list;
    }
```

Neither of those solutions is a principal typing. In Java-TX **Plue21_2** we make use of overloading, such that

```
class Example extends Object {
    ...
    List<Object> method(List<Object> list){ ... }

    List<Example> method(List<Example> list){ ... }
    }
}
```

is the inferred program. But what if another subtype `newExample` of `Example` is added afterwards to the program. The `method` cannot be called with `newExeample` as an argument. The compiler has to be started again, so that the type inference algorithm will infer both classes `Example` and `newExample`.

Our new **Unify** algorithm will lead to the following solution:

```
class Example extends Object{
    Object f;
    <A extends Object> List<A> method(List<A> list){
        f = list.get();
        return list;
    }
}
```

The program is principal in Java-TX and now every new sub-type of `Example` `method` is applicable.

$T <: T$	S-REFL
$\dfrac{S <: T \qquad T <: U}{S <: U}$	S-TRANS
$\dfrac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends N } \{\ldots\}}{\texttt{C<}\overline{\texttt{T}}\texttt{>} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}}$	S-CLASS

Figure 1. Suptyping rules

3 Input

The **Unify** algorithm takes two kinds of constraints as input.

Constraint A constraint consists of two types or type variables and an operator. The operator can either be a $\doteq$ (same type) or $\lessdot$ (subtype). Example: $(a \lessdot \texttt{Object})$, means that the type variable a should be a subtype of `Object`.

OrConstraint An OrConstraint consists out of multiple constraint sets. For example **OrConstraint**$(\{\ \{(a \lessdot b), (a \leqslant \texttt{Object})\}\ ,\ \{(a \lessdot b)\}\ \})$ is an Or-Constraint consisting of two constraint sets.

Additionally the **Unify** algorithm uses a subtype relation $<:$. This is defined in the rules in figure 1.

The subtype relations are derived from the class definitions of the input. The syntax of a class definition is as the following:

$$\texttt{T} ::= \texttt{X} \mid \texttt{N}$$
$$\texttt{N} ::= \texttt{C<}\overline{\texttt{T}}\texttt{>}$$
$$\texttt{L} ::= \texttt{class C<}\overline{\texttt{X}}\texttt{> extends N } \{\ldots\}$$

Here $\overline{\texttt{X}}$ depicts a vector of generic type variables X.

Notice that we do neither use wildcard types nor bounds for generic type variables for the input. Both are possible in `Java` and can also be handled by our Java-TX type inference algorithm. Although in this paper we take a look at the unification algorithm without wildcards.

4 Unify algorithm

This chapter describes the **Unify** algorithm.

Input A set of type constraints $Cons_{in}$

Output A set of type unifiers Uni or fail $Uni = \emptyset$.

Hints:

- ▻ The $\lessdot^*$ used in this chapter means, that $a \lessdot^* b$ can either be a single constraint $a \lessdot b$ or a transitive series of constraints $a \lessdot c, \ldots, m \lessdot n, n \lessdot b$.
- ▻ $[a/b]T$ means replacing all type variables b with a in the type T.

Definition 4.1 (Isolated type variable)**.** An isolated type variable does only occur in constraints together with another isolated type variable.

Definition 4.2 (Solved form)**.** A set of constraints Eq has reached solved form if it contains only the following kind of constraints:

1. $a \lessdot b$, with a and b both isolated type variables
2. $a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}$, with $a \notin \overline{\texttt{X}}$
3. $a \doteq \texttt{C<}\overline{\texttt{X}}\texttt{>}$, with $a \notin \overline{\texttt{X}}$

The unify algorithm first has to build the cartesian product of all the **OrConstraint**s and the remaining normal constraints:

$$\begin{aligned} \Omega &= \text{All } \textbf{OrConstraints} \text{ in } Cons_{in} \\ C &= Cons_{in} \backslash \Omega \\ Eq_{set} &= \Omega_1 \times \ldots \times \Omega_n \times C \quad \text{for all } \Omega_i \in \Omega \end{aligned}$$

Afterwards the following steps are repeatedly executed on Eq_{set} until the algorithm terminates:

1. Repeated application of the rules depicted in Figure 2 and 3. The end configuration of Eq is reached if for each element no rule is applicable.
2.

$$\begin{aligned} Eq_1 &= \text{Subset of pairs where both type terms are type variables} \\ Eq_2 &= Eq / Eq_1 \\ Eq_{set} &= \times \Big(\bigotimes_{(\texttt{C<}\overline{\texttt{T}}\texttt{>} \lessdot a) \in Eq_3} \{ a \doteq [\overline{\texttt{T}}/\overline{\texttt{X}}]N \mid \texttt{C<}\overline{\texttt{X}}\texttt{>} <: N \} \Big) \end{aligned}$$

$$\begin{aligned}
&(\bigotimes_{\{a \lessdot \texttt{C<}\overline{\texttt{T}}\texttt{>}\} \in Eq_2, \{a \lessdot^* b\} \in Eq_1} \{b \lessdot \texttt{C<}\overline{\texttt{T}}\texttt{>}\} \cup \{b \doteq [\overline{\texttt{T}}/\overline{\texttt{X}}]N \mid \texttt{C<}\overline{\texttt{X}}\texttt{>} <: N\}) \\
&\times \{[a \doteq N \mid (a \doteq N) \in Eq_2]\} \\
&\times \{[a \lessdot N \mid (a \lessdot N) \in Eq_2]\} \times Eq_1
\end{aligned}$$

3. Application of the following *subst* rule for every $Eq'' \in Eq_{set}$

$$\text{(subst)} \quad \frac{Eq'' \cup \{a \doteq N\}}{Eq''[a \mapsto N] \cup \{a \doteq N\}} \quad a \text{ occurs in } Eq'' \text{ but not in } N$$

   for each $a \doteq N$ in each element of $Eq' \in Eq'_{set}$.
4. a) Foreach $Eq \in Eq_{set}$ which has changed in the last step start again with the first step.
   b) Build the union Eq_{set} of all results of (a) and all $Eq' \in Eq'_{set}$ which has not changed in the last step.
5. a) Filter all constraint sets which are in solved form:
   $Eq_{solved} = \{Eq \mid Eq \in Eq_{set}, Eq \text{ is in solved form}\}$
   b) We apply the following rule to every constraint set in Eq_{solved}:

$$\frac{Eq \cup \{a \lessdot b\}}{[a/b]Eq \cup \{a \doteq b\}}$$

   c) Finally we generate the unifiers:

$$\begin{aligned}
Uni = &\{\sigma \mid Eq \in Eq_{solved},\ \sigma = \{a \mapsto T \mid (a \doteq \mathsf{T}) \in Eq\}\} \cup \\
&\{\sigma \mid Eq \in Eq_{solved},\ \sigma = \{a \mapsto (X \lhd T) \mid (a \lessdot \mathsf{T}) \in Eq\}\}
\end{aligned}$$

4.1 Unify proof

Theorem 1. (Soundness): If the **Unify** algorithm finds a solution it does not contradict any of the input constraints: $\nexists (a \lessdot b) \in Cons_{in}$ where $\sigma(a) \nleq \sigma(b)$

Proof: We show theorem 1 by going backwards over every step of the algorithm. We assume there exists a unifier $\sigma = \{a_1 \mapsto \theta_1, \ldots, a_n \mapsto \theta_n\}$ for the input constraints, which is the result of the **Unify** algorithm. This means for every constraint in the input set $(a \lessdot b) \in Cons_{in}$ and $(c \doteq d) \in Cons_{in}$ this unifier will substitute all variables in a way that all constraints are satisfied: $\sigma(a) \leqslant \sigma(b)$, $\sigma(c) = \sigma(d)$

(match) $$\frac{Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, a \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}}{Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, \texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}} \quad \texttt{C<}\overline{\texttt{Z}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$$

(adopt) $$\frac{Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot^* a, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}}{Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot^* a, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}, b \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}\}}$$

(adapt) $$\frac{Eq \cup \{\texttt{C<}\overline{\texttt{A}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{B}}\texttt{>}\}}{Eq \cup \{\texttt{D<}[\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}\texttt{>} \doteq \texttt{D<}\overline{\texttt{B}}\texttt{>}\}} \quad \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{Y}}\texttt{>}$$

(reduce1) $$\frac{Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{B}}\texttt{>}\}}{Eq \cup \{\overline{\texttt{A}} \doteq \overline{\texttt{B}}\}}$$

(reduce2) $$\frac{Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>} \doteq \texttt{D<}\overline{\texttt{B}}\texttt{>}\}}{Eq \cup \{\overline{\texttt{A}} \doteq \overline{\texttt{B}}\}}$$

(equals) $$\frac{Eq \cup \{a \lessdot b, b \lessdot c, \ldots, m \lessdot n, n \lessdot a\}}{Eq \cup \{a \doteq b, a \doteq c, \ldots\}}$$

Figure 2. Reduce and adapt rules

(erase1) $$\frac{Eq \cup \{C \lessdot D\}}{Eq} \quad C <: D$$

(erase2) $$\frac{Eq \cup \{C \doteq C\}}{Eq}$$

(swap) $$\frac{Eq \cup \{C \doteq a\}}{Eq \cup \{a \doteq C\}}$$

Figure 3. Erase and swap rules

We now look at each step of the **Unify** algorithm which transforms the input set of constraints Eq to a set Eq'. If we assume the unifier σ is correct for the set Eq', then we can show that it will also be correct for the constraints Eq.

Step 5 c) The last step of the algorithm transforms a set of constraints Eq in solved form No changes to the constraint set are applied here.

Step 5 b) A unifier which is correct for $a \doteq b$ is also correct for $a \lessdot b$. The transformation $Eq' = [a/b]Eq$ does not change this. The type variable b appears only in constraints of the form $a \lessdot b$, with both sides being type variables. There cannot be two constraints $a \lessdot b, b \lessdot a$ in Eq. This would have been removed by the `equals` rule. So the only two other possible combinations for constraints containing b would be $b \lessdot c$ and $c \lessdot b$.
$\rhd$ If $\{a \lessdot b, b \lessdot c\} \in Eq$ then $\{a \doteq b, a \lessdot c\} \in Eq'$.
$\rhd$ If $\{a \lessdot b, c \lessdot b\} \in Eq$ then $\{a \doteq b, c \lessdot b\} \in Eq'$.
In both cases a correct unifier for Eq' would also be correct for Eq.

Step 5 a) We do not alter the constraint set which lateron lead to the unifier.

Step 4 The constraint sets are not altered here.

Step 3 An unifier σ that is correct for a constraint set $Eq[a \to \theta] \cup (a \doteq \theta)$ is also correct for the set $Eq \cup (a \doteq \theta)$. From the constraint $(a \doteq \theta)$ it follows that $\sigma(a) = \theta$. This means that $\sigma(Eq) = \sigma(Eq[a \to \theta])$, because every occurence of a in Eq will be replaced by θ anyways when using the unifier σ.

Step 2 This step transforms constraints of the form $(\mathtt{C}<\overline{\mathtt{X}}> \lessdot a)$ and $(a \lessdot \mathtt{C}<\overline{\mathtt{X}}>)$ into sets of constraints and builds the cartesian product with the remaining constraints. We can show that if there is a resulting set of constraints which has σ as its correct unifier then σ also has to be a correct unifier for the constraints before this transformation.
We look at each transformation done in step 2:
$\{\mathtt{C}<\overline{\boldsymbol{T}}> \lessdot a\} \in Eq \to \{a \doteq [\overline{\boldsymbol{T}}/\overline{\boldsymbol{X}}]N\} \in Eq'$**:** If $\mathtt{C}<\overline{\mathtt{X}}> <: N$ and σ is correct for $(a \doteq [\overline{\mathtt{T}}/\overline{\mathtt{X}}]N)$ then σ is also correct for $(\mathtt{C}<\overline{\mathtt{T}}> \lessdot a))$. When substituting a for $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]N$ we get $(\mathtt{C}<\overline{\mathtt{T}}> \lessdot [\overline{\mathtt{T}}/\overline{\mathtt{X}}]N)$, which is correct because $\mathtt{C}<\overline{\mathtt{X}}> <: \mathtt{C}<\overline{\mathtt{Y}}>$ (see `S-CLASS` rule).
$\{a \lessdot \mathtt{C}<\overline{\mathtt{T}}>,\ a \lessdot^* b\} \in Eq \to \{T \lessdot b, a \lessdot \mathtt{C}<\overline{\mathtt{T}}>,\ a \lessdot^* b\} \in Eq'$ Trivial
$\{a \lessdot \mathtt{C}<\overline{\mathtt{T}}>,\ a \lessdot^* b\} \in Eq \to \{a \doteq [\overline{\boldsymbol{T}}/\overline{\boldsymbol{X}}]N, a \lessdot \mathtt{C}<\overline{\mathtt{T}}>,\ a \lessdot^* b\} \in Eq'$ This is the same as in the first transformation. Here we can also show correctness via the `S-CLASS` rule.

Step 1 erase-rules remove correct constraints from the constraint set. A unifier σ that is correct for the constraint set Eq is also correct for $Eq \cup \{\theta \doteq \theta\}$ and $Eq \cup \{\theta \lessdot \theta'\}$, when $\theta \leqslant \theta'$.

swap-rule does not change the unifier for the constraint set. $\doteq$ is a symmetric operator and parameters can be swapped freely.

match The subtype relation is transitive, so if there is a correct solution for $a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, \texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}$ then this solution would also apply for $a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}$ or $a \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}$.

adopt An unifier which is correct for $Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot^* a, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}, b \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}\}$ is also correct for $Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot^* a, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}$.

adapt If there is a σ which is a correct unifier for a set $Eq \cup \{\texttt{C<}[\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}\texttt{>} \doteq \texttt{C<}\overline{\texttt{B}}\texttt{>}\}$ then it is also a correct unifier for the set $Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>} \lessdot \texttt{C<}\overline{\texttt{B}}\texttt{>}\}$, if there is a subtype relation $\texttt{D<}\overline{\texttt{X}}\texttt{>} \leqslant^* \texttt{C<}\overline{\texttt{Y}}\texttt{>}$. To make the set $Eq \cup \{[\overline{\texttt{A}}/\overline{\texttt{X}}]\texttt{C<}\overline{\texttt{Y}}\texttt{>} \doteq \texttt{C<}\overline{\texttt{B}}\texttt{>}\}$ the unifier σ must satisfy the condition $\sigma([\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}) = \sigma(\overline{\texttt{B}})$. By substitution we get $Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>} \lessdot \texttt{C<}[\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}\texttt{>}\}$ which is correct under the `S-CLASS` rule.

reduce The `reduce1` and `reduce2` rules are obviously correct under the FJ typing rules.

OrConstraints If σ is a correct unifier for one of the constraint sets in Eq_{set} then it is also a correct unifier for the input set $Cons_{in}$. When building the cartesian product of the **OrConstraints** every possible combination for $Cons_{in}$ is build. No constraint is altered, deleted or modified during this step.

□

Theorem 2. (Completeness): The **Unify** algorithm calculates a general unifier for the input set of constraints ($Cons_{in}$). A unifier σ is a general unifier for $Cons_{in}$ if it unifies $Cons_{in}$ and for every other unifier ω there is a substitution λ so that $\omega(x) = \lambda(\sigma(x))$.

Proof:

We look at every step of the algorithm, which alters the set of constraints Eq, while assuming that there is at least one possible principal type solution σ for the input. We will show that the principal type is among them by proofing for every step of the algorithm that the principal type is never excluded.

Step 1: The first step applies the three rules from figure 3. **erase-rules:** The erase2 rule from figure 3 removes a $\{C \doteq D\}$ constraint from the constraint set. The erase1 rule removes a $\{C \doteq C\}$ constraint, but only if the two types C and D satisfy the constraint. Both rules do not change the set of possible solutions for the given constraint set.

swap-rule: $\doteq$ is a symmetric operator and parameters can be swapped freely. This operation does not change the meaning of the constraint set.

match-rule: If there is a solution for $a \lessdot$ C<$\overline{X}$>, $a \lessdot$ D<$\overline{Y}$>, this is also a solution for $a \lessdot$ C<$\overline{X}$>, C<$\overline{X}$> $\lessdot$ D<$\overline{Y}$>. A correct unifier σ has to find a type for a, which complies with $a \lessdot$ C<$\overline{X}$> and $a \lessdot$ D<$\overline{Y}$>. Due to the subtyping relation being transitive this means that $\sigma(a) \lessdot$ C<$\overline{X}$> $\lessdot$ D<$\overline{Y}$>.

adopt-rule: Subtyping in FJ is transitive, which allows us to apply the adopt rule without excluding any possible unifier.

adapt-rule: Every solution which is correct for the constraints $Eq \cup \{$C<$[\overline{A}/\overline{X}]\overline{Y}$> $\doteq$ C<$\overline{B}$>$\}$ is also a correct solution for the set $Eq \cup \{$D<$\overline{A}$> $\lessdot$ C<$\overline{B}$>$\}$. According to the S-CLASS rule there can only be a possible solution for C<$[\overline{A}/\overline{X}]\overline{Y}$> $\doteq$ C<$\overline{B}$> if $\overline{B} = [\overline{A}/\overline{X}]\overline{Y}$. Therefore this transformation does not remove any possible solution from the constraint set.

reduce-rule: For a constraint D<$\overline{A}$> $\lessdot$ D<$\overline{A}$> the FJ subtyping rule S-REFL ($\triangle \vdash T <: T$) is the only one which applies. According to this rule the transformation to $\overline{A} \doteq \overline{B}$ is correct. Only D gets removed, which is not a type variable. Therefore this step does not remove a possible solution. This applies for both reduce rules **reduce1** and **reduce2**.

equals-rule: This rule removes a circle in the constraints. This does not remove a general unifier.

Step 2: The second step of the algorithm eliminates $\lessdot$-constraints by replacing them with $\doteq$-constraints. For each (C<$\overline{X}$> $\lessdot a$) constraint the algorithm builds a set with every possible supertype of C<$\overline{X}$>. So if there is a correct unifier σ for the constraints before this conversion there will be at least one set of constraints for which σ is a correct unifier.

Step 3: In the third step the **substitution**-rule is applied. If there is a constraint $a \doteq N$ then there is no other way to fulfill the constraint set than replacing a with N. This does not remove a possible solution.

Step 4: None of the constraints get modified.

Step 5 a): The removed sets do not have a possible unifier, therefore no possible solution is omitted in this step.

Proof: In step 5.a all constraint sets that have a unifier are in solved form. All other possibilities are eliminated in steps 1-4. There are 8 different variations of constraints:
$(a \doteq a), (a \doteq C), (C \doteq a), (C \doteq C), (a \lessdot a), (a \lessdot C), (C \lessdot a), (C \lessdot C)$
After step 1 there are no $(C \doteq C)$, $(C \lessdot C)$ and $(C \doteq a)$ constraints anymore, as long as the constraint set has a correct unifier. Because a constraint set that has a correct unifier cannot contain constraints of the form $N_1 \doteq N_2$ with $N_1 \neq N_2$ and $(N_1 \lessdot N_2)$ with $(N_1 \not\leq: N_2)$. By removing $(N \doteq N)$ and $(C \lessdot D)$ with $(C <: D)$

constraints no constraints of the form $(C \doteq C)$ and $(C \lessdot D)$ remain in a constraint set that has a correct unifier after step 1.

After step 2 there are no more $(C \lessdot a)$ constraints.

After step 3 there are no $(a \doteq C)$ anymore.

We only reach step 5 if the constraint set is not changed by the substitution (step 3).

If the constraint set has a correct unifier only $(a \lessdot a)$, $(a \doteq a)$, $a \lessdot C$ and $(a \doteq C)$ constraints are left at this point. The type variables in the $(a \lessdot a)$ and $(a \doteq a)$ constraints have to be independent type variables. If a type variable c is inside a $(c \doteq C)$ constraint it is not an independent type variable. But this variable c cannot be inside a $(a \doteq a)$ or $(a \lessdot a)$ constraint, because otherwise step 3 would have replaced it in there.

So this step only excludes constraint sets which do not have a correct unifier.

Step 5 b): If the algorithm advances to this step we further only work on constraint sets in solved form. This means there are only two kinds of constraints left. $(a \doteq \mathsf{T})$, $(a \lessdot \mathsf{T})$, $(a \doteq b)$ and $(a \lessdot b)$ with a and b as type variables.

The FGJ language does not allow subtype constraints for generic types. A constraint like $(a \lessdot b)$ in a solution could be inserted as the typing shown in the example below. But this is not allowed by the syntax of FGJ. That is why we can treat this constraint as $(a \doteq b)$.

Example: This would be a valid Java program but is not allowed in FGJ:

```
class Example {
  <A extends Object, B extends A> A id(B a){
    return a;
  }
}
```

By replacing all $(a \lessdot b)$ constraints with $(a \doteq b)$ we do not remove a principal type solution.

Step 6: In the last step all the constraint sets, which are in solved form, are converted to unifiers.

We see that only a constraint set which has no unifier does not reach solved form. We showed that in every step of the **Unify** algorithm we never exclude a possible unifier. Also we showed that after we reach step 5 only constraint sets with a correct unifier are in solved form. By removing all constraint sets which are not in solved form the algorithm does not remove a possible correct unifier.

If we assume that there is a possible principal type solution σ for the input set $Cons_{in}$ and the **Unify** algorithm does not exclude any of the possible unifiers, then the result **Unify** contains the principal type solution. □

Theorem 3. (Termination): The **Unify** algorithm terminates on every finite input set.

The **Unify** algorithm gets called with a set of input constraints. After resolving the **OrConstraints** we end up with multiple *Eq* sets. Afterwards the algorithm iterates over each of those sets (see Chapter 4). We will show that **Unify** terminates on each of those sets by showing, that each step of the algorithm removes at least one type variable until the finishing state is reached. The finishing state for a constraint set is reached when step 3 is not able to substitute a type variable. This is checked by step 4 of the algorithm. Then the *Eq* set is either in solved form or determined to be unsolvable.

Proof: The **Unify** algorithm reduces the amount of type variables with every iteration. No step adds a new type variable to the constraint set. Additionally we have to show that the first step of the algorithm also terminates on every finite input set.

Step 1 Step 1 of the algorithm always terminates. *Proof:* Every rule either removes a $\lessdot$ constraint or reduces a $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ to $\overline{\texttt{X}}$ inside a constraint. None of the rules add a new $\lessdot$ constraint or a $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ type to the constraint set. Step 1 has to come to a stop once there are no more $\lessdot$ constraints or $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ types to reduce.

The rule **match** seems to generate a new $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ constraint, but the $\texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}$ constraint added by `match` will be changed immidiatly into a $\doteq$ constraint by the `adapt` rule. Afterwards the `reduce1` rule will remove this freshly added $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ type. So effectively a $\doteq$ constraint is removed by this rule in combination with `adapt` and `reduce1`.

The **adopt** rule seems to generate a new $\lessdot$ constraint. But the `adopt` rule triggers two other rules. The `match` and the `adapt` rule.

1. We start with the `adopt` rule:
$$\frac{Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot^* a, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}}{Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot^* a, b \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}}$$
2. We can now apply the `match` rule to the two resulting $(b \lessdot \ldots)$-constraints. If this is not possible due to type `C` not being a subtype of `D` or vice versa, then the *Eq* set has no possible solution and **Unify** would terminate as fail $Uni = \emptyset$:
$$\frac{Eq \cup \{b \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}}{Eq \cup \{b \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, \texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}} \quad \texttt{C<}\overline{\texttt{Z}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$$
3. The constraint added by the `match` rule fits the `adapt` rule, which we apply in the next step: $$\frac{Eq \cup \{\texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}}{Eq \cup \{\texttt{D<}[\overline{\texttt{X}}/\overline{\texttt{Z}}]\overline{\texttt{N}}\texttt{>} \doteq \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}} \quad \texttt{C<}\overline{\texttt{Z}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$$

In the end we have the conversion:

$$\frac{Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot^* a, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}}{Eq \cup \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, b \lessdot^* a, b \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}, \texttt{D<}[\overline{\texttt{X}}/\overline{\texttt{Z}}]\overline{\texttt{N}}\texttt{>} \doteq \texttt{D<}\overline{\texttt{Y}}\texttt{>}\}}$$

We can see now, that only a $\doteq$ constraint is added. The `adopt` alone adds a $\lessdot$ constraint, but due to the fact that it is always used together with `match` and `adapt` it effectively just adds a $\doteq$ constraint.

Step 2 This step does not add new type variables to the constraint set.

Step 3 The third step of the **Unify** algorithm removes at least one type variable from the constraint set or otherwise does not alter Eq at all. If Eq is not altered the algorithm terminates in the next step. The type variable is not completely removed but stays inside Eq only in one $a \doteq N$ constraint. All other occurences are replaced by N. The `subst` step can therefore only be executed once per type variable.

We see that with each iteration over the steps 1-3 at least one type variable is removed from the constraint set. Due to the fact that there is never added a fresh type variable during the **Unify** algorithm, the algorithm will terminate for any given finite set of constraints. □

5 Examples

Lets look at some examples of different compositions of constraints and how our unify algorithm will transform them.

match-rule Example:

The match-rule takes two constraints of the form $a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, a \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}$ and checks which one of the two types `C` and `D` is a subtype of the other. We only leave the constraint, which is more restrictive. If `C` and `D` do not have a subtype relation, the match-rule won't apply. This will lead to the constraint set never reaching solved form. We cannot remove the $a \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}$ constraint without matching the $\overline{\texttt{Y}}$ and $\overline{\texttt{X}}$. We do this by generating the constraint $\texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{Y}}\texttt{>}$. This constraint will then be processed by the adapt and the reduce rule.

Example: $Eq = \{a \lessdot \texttt{C<b,String>}, a \lessdot \texttt{D<b,b>}\}$ with $\texttt{C<A,B>} <: \texttt{D<B,B>}$.

$$\cfrac{\cfrac{\cfrac{\{a \lessdot \texttt{C<b,String>}, a \lessdot \texttt{D<b,b>}\}}{a \lessdot \texttt{C<b,String>}, \texttt{C<b,String>} \lessdot \texttt{D<b,b>}}\ \text{match}}{a \lessdot \texttt{C<b,String>}, \texttt{D<String,String>} \doteq \texttt{D<b,b>}}\ \text{adapt}}{a \lessdot \texttt{C<b,String>}, b \doteq \texttt{String}, b \doteq \texttt{String}}\ \text{reduce2}$$

adopt-rule Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{a \lessdot \texttt{C<String>}, b \lessdot a, b \lessdot \texttt{C<c>}\}}{\{a \lessdot \texttt{C<String>}, b \lessdot a, b \lessdot \texttt{C<String>}, b \lessdot \texttt{C<c>}\}}\ \text{adopt}}{\{a \lessdot \texttt{C<String>}, b \lessdot a, b \lessdot \texttt{C<String>}, \texttt{C<String>} \lessdot \texttt{C<c>}\}}\ \text{match}}{\{a \lessdot \texttt{C<String>}, b \lessdot a, b \lessdot \texttt{C<String>}, \texttt{String} \doteq c\}}\ \text{reduce2}}{\text{(solved form reached)}}\ \text{-}$$

Step 2 Example: Input: $\{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, a \lessdot b\}$ Step 2: $Eq_1 = \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, a \lessdot b, b \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}\}$ and $Eq_2 = \{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, a \lessdot b, \texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot b\}$

When the the set Eq contains $\{a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}, a \lessdot b\}$ then we have two possibilities. The type variable b can either be a subtype of $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ or a supertype. Both could be a possible solution.

We could add two constraints sets, one containing $b \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}$ and the other $\texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot b$. The second constraint would be changed by step 2 into $b \doteq T_1, \ldots, b \doteq T_n$ with T being the supertypes of $b \doteq \texttt{C<}\overline{\texttt{X}}\texttt{>}$. Therefore we can directly add all of the supertypes as a set.

Use case Example: We have the following input program:

```
class Example2 extends Object{
    Object lastId;
    id(x){
        lastId = x;
        return x.get();
    }
}
```

This is a Java program, where the parameter type for `x` and the return type for `id` has been left out. We want to use our **Unify** algorithm to find a correct typing for this program. We generate the following constraints:
$Cons = \{x \lessdot \texttt{Object}, b \lessdot id, x \lessdot \texttt{List<b>}\}$, with x being the type placeholder for the type of the parameter `x` and id being the placeholder for the return type of `id`.

During the **Unify** algorithm the `match` rule gets applied resulting in $Eq = \{x \lessdot \texttt{List<b>}, b \lessdot id, \texttt{Object} \lessdot \texttt{List<b>}\}$. After applying `adapt` and `reduce1` we end up with:
$Eq = \{x \lessdot \texttt{List<b>}, b \lessdot id\}$

The unused variable b is now also inserted in the input program as a generic variable:

```
class Example2 extends Object{
    Object lastId;
    <B, X extends List<B>> B id(List<B> x){
```

```
        lastId = x;
        return x.get();
    }
}
```

6 Summary

We chenged our **Unify** algorithm to emit generic type variables in the form of $a \lessdot \texttt{List<String>}$ constraints. We can now use this to insert a generic type variable instead of an overloaded method. The constraint shown above for example could lead to the following method typing:

```
<A extends List<String>> void methode(A a){
    a.add("Example");
}
```

The **Unify** algorithm presented in this paper is the type inference algorithm from our Java-TX project Plümicke, 2007, but with the following changes:

- ▷ At first we stripped the algorithm of the handling of wildcard types. This made it easier to show Soundness and Completeness for the following changes.
- ▷ We changed the handling of $a \lessdot T$ constraints in step 2 of the **Unify** algorithm.
- ▷ We introduced two new rules in step 1 of the **Unify** algorithm. The adopt and the match rule.

For this new algorithm we could show Soundness and Completeness and we showed in this paper, that the algorithm terminates for every finite input.

7 Outlook

In this paper we presented a generalization of the Java type unification algorithm for the type system of GJ, respectively, FGJ. In comparision to the type system of Java, respectively, Java-TX the main restriction is that type arguments are invariant. In contrast Java/Java-TX allows use-site variance by wildcards. The type unification with wildcards grows the results enourmously. Let us consider the following example

Example 7.1. In this example we use the standard Java types Number, Integer, Stack, Vector, AbstractList, and List. It holds $\texttt{Integer} <: \texttt{Number}$ and $\texttt{Stack<a>} <: \texttt{Vector<a>} <: \texttt{AbstractList<a>} <: \texttt{List<a>}$.

As a start configuration we use

$\{(a \lessdot \texttt{List<? extends List<? extends Number>>}),\ (b \lessdot a)\}$.

At the moment, first, for a all subtypes of $\texttt{Vector<? extends Vector<? extends Number>>}$ determined. For each subtype an **OrConstraint** is built:

$\{(a \doteq \texttt{List<? extends List<? extends Number>>}), (b \lessdot a)\} \parallel$
$\{(a \doteq \texttt{List<? extends List<Number>>}), (b \lessdot a)\} \parallel$
$\{(a \doteq \texttt{List<? extends List<? extends Integer>>}), (b \lessdot a)\} \parallel$
$\vdots$

In this framework Number have two subtypes and List four subtypes. As in the argument position the type and the extends-type are subtypes respectively, the number of subtypes is given as:

$$4 \times 8 \times 4 = 64$$

For each element of the **OrConstraint** the algorithm is continued. In the next step a is instantiated, respectively. Then, for b all subtypes are determined. It is obvious that the number of **OrConstraint**s grows enourmously.

As the input $\{(a \lessdot \texttt{List<? extends List<? extends Number>>}),\ (b \lessdot a)\}$ corresponds to the result of the by wildcards extended algorithm, the number of results are enourmously reduced.

At the moment we have optimized our implementation of the type unification algorithm by an order-strategy of evaluation the **OrConstraint**s Plümicke, 2018. The above example shows that the by wildcards extended generalized algorithm will reduce the number of **OrConstraint**s. We hope that a combination of the order-strategy and the generalized algorithm will reduce the runtime, significantly.

References

Bracha, Gilad et al. (1998). *GJ Specification*.

Damas, Luis and Robin Milner (1982). “Principal type-schemes for functional programs”. In: *Proc. 9th Symposium on Principles of Programming Languages*.

Martelli, A. and U. Montanari (1982). “An efficient unification algorithm”. In: *ACM Transactions on Programming Languages and Systems* 4, pp. 258–282.

Plümicke, Martin (2021). “Java–TX: The language – A Java extension with global type inference, real functions types, generated generics and heterogeneous translation of function types–”. (to appear).

– (July 2004). “Type Unification in Generic–Java”. In: *Proceedings of 18th International Workshop on Unification (UNIF’04)*. Ed. by Michael Kohlhase. Cork.

– (July 2007). “Java type unification with wildcards”. In: *Proceedings of 21th International Workshop on Unification (UNIF'07)*. Ed. by Évelyne Contejean. Paris.
– (2009). “Java type unification with wildcards”. In: *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*. Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf. Vol. 5437. Lecture Notes in Artificial Intelligence. Springer-Verlag Heidelberg, pp. 223–240.
– (2015). “More type inference in Java 8”. In: *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. Ed. by Andrei Voronkov and Irina Virbitskaite. Vol. 8974. Lecture Notes in Computer Science. Springer, pp. 248–256.
– (2018). “Optimization of the Java type unification”. In: *Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts*. Ed. by Jens Knoop, Martin Steffen, and Baltasar Trancón y Widemann. Research Report 482. ISBN 978-82-7368-447-9. Faculty of Mathematics and Natural Sciences, UNIVERSITY OF OSLO, pp. 3–12.

Plümicke, Martin and Andreas Stadelmeier (2017). “Introducing Scala-like function types into Java-TX”. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: ACM, pp. 23–34. ISBN: 978-1-4503-5340-3. DOI: 10.1145/3132190.3132203. URL: http://doi.acm.org/10.1145/3132190.3132203.

Reinhold, Mark (Dec. 2009). *Project lambda: straw-man proposal*. URL: http://cr.openjdk.java.net/~mr/lambda/straw-man.

Robinson, J. A. (Jan. 1965). “A machine-oriented logic based on the resolution principle”. In: *Journal of ACM* 12(1), pp. 23–41.

Can Programming be Liberated From the Functional Style?

Für die Einführung des Plurals in die (objektorientierte) Programmierung

Friedrich Steimann
steimann@acm.org
Fernuniversität in Hagen

Abstract
Funktionen sind für die Programmierung zentral, nicht nur, weil sich logische, arithmetische und Navigationsausdrücke als Funktionen auffassen lassen, sondern auch, weil, wie von Backus [1978] in seinem provokant betitelten Vortrag betont, Funktionen geeignet sind, die Programmierung grundsätzlich von den Gegebenheiten einer Hardware loszulösen. Gleichwohl zwängt die Fokussierung auf Funktionen in einer Welt, in der auch Relationen, die keine Funktionen sind, ein wichtige Rolle spielen, der Programmierung einen Stil auf, der bisweilen umständlich anmutet, jedenfalls die Verwendung von Konstruktionen erfordert, die als idiomatisch betrachtet werden können und die somit Kandidaten für weiterreichende Abstraktionen sind. Mit diesem Beitrag soll die Leserschaft für Probleme der Fokussierung auf Funktionen sensibilisiert und einfache Lösungsansätze auf verschiedenen Ebenen vorgestellt werden, denen allesamt eines gemeinsam ist: die Einführung des Plurals in die Programmierung.

1. Motivation

Funktionen spielen in der Programmierung eine zentrale Rolle. So lassen sich in der imperativen und funktionalen Programmierung viele Ausdrücke als (partielle) Funktionen auffassen und in der objektorientierten Programmierung ordnen Felder (Instanzvariablen) und (Instanz-)Methoden einem Objekt, dem Besitzer des Feldes oder dem Empfänger des Methodenaufrufs, ein anderes zu. Ist die logische Zuordnung partiell oder eine zu mehr als einem Objekt, dann müssen allerdings spezielle Konstrukte bemüht werden: Für die Partialität wird häufig ein spezielles Objekt, das Null-Objekt (in der objektorientierten Programmierung in der Regel codiert durch Tony Hoares „billion-dollar mistake“, den Null-Zeiger,

in funktionalen Sprachen eher durch einen Zweig eines Summentyps dargestellt), eingesetzt, für mehrere Objekte dagegen ein Containerobjekt, das die Objekte „enthält“ (auf diese verweist). Alternativ wird in der objektorientierten Programmierung auch für Partialität ein spezieller, einelementiger Container (wie `Optional` in Java oder `Option` in Scala) verwendet, der auch leer sein kann (dann mit der Bedeutung von „kein Objekt“). Dieser Container ist jedoch in der Regel kein Spezialfall (kein Subtyp) eines Containers für mehrere Elemente, so dass, in Sprachen mit statischer Typprüfung, „kein oder ein Objekt“ (repräsentiert durch den Typ `Optional` o. ä.) nicht da auftreten kann, wo „mehrere Objekte“ (repräsentiert durch `List` o. ä.) erwartet wird. Tatsächlich wäre das, wie wir noch sehen werden, auch problematisch, zumindest wenn dies erlauben würde, einem Container vom Typ `Optional` über einen Alias vom Typ `List` weitere Elemente hinzuzufügen (was nach den Regeln des Subtyping in der objektorientierten Programmierung gemeinhin erlaubt wäre).

Nun verlangt aber die Einführung eines Null-Objekts zur Codierung von partiellen Funktionen immer auch eine Fallunterscheidung (zwischen den Fällen „kein Objekt“ und „ein Objekt“) und die Einführung eines Containers, egal ob für ein oder mehrere Objekte, bedeutet eine Indirektion, die sich nicht zuletzt im Typ des Funktionswerts widerspiegelt: Statt kein, ein oder mehrere Objekte des erwarteten Typs bekommt man ein Objekt des Containertyps, das die erwarteten Objekte (ggf. keines) enthält. Mit dem Übergang von Einem, oder Keinem oder Einem, zu Vielen verbunden sind daher eine ganze Reihe von Unstetigkeiten oder Brüchen, wie die folgenden Beispiele für eine objektorientierte Programmiersprache wie Java zeigen:

1. **Anzahl bestimmt den Typ** Kann man beispielsweise in Java

   ```
   Konto k = new Konto();
   ```

   schreiben, so ergibt

   ```
   List<Konto> ks = new Konto();
   ```

   einen Typfehler, obwohl es sich bei der Änderung der Deklaration hier konzeptuell lediglich um einen Wechsel der Anzahl handelt: `k` kann kein oder ein Konto aufnehmen („Einzahl“), `ks` hingegen eine beliebige Anzahl („Mehrzahl“). Auch lässt sich beispielsweise aus der Typkorrektheit von `k.aufloesen()` nicht die Korrektheit von `ks.aufloesen()` folgern: Hierfür müsste erst der Typ `List` entsprechend ergänzt werden, was aber, angesichts der Allgemeinheit von `List`, nicht angezeigt scheint. Stattdessen findet man hier eher so etwas wie `ks.stream().map(k -> k.aufloesen())`, was mit der für die Einzahl hinreichenden Form nur wenig gemeinsam hat. Ein Bruch also.

2. **Anzahl bestimmt die Regeln des Subtyping** Ist `Konto` eine Subklasse von `Object`, so ist, bei Fortbestehen obiger Deklarationen,

   ```
   Object o = k;
   ```

   in Java eine gültige Zuweisung,

   ```
   List<Object> os = ks;
   ```

   hingegen nicht. Der Grund hierfür ist, dass durch die Zuweisung mit `os` ein Alias vom Typ `List<Object>` auf ein Objekt vom Typ `List<Konto>` gebildet würde, der es erlauben würde, diesem Objekt, einem Container, Elemente des Typs `Object` (inkl. seiner Subtypen) hinzuzufügen, wodurch aber in allen Fällen, in denen es sich hierbei nicht um Konten handelt, die Typinvariante von `ks` verletzt würde. Um Kovarianz bei parametrisch polymorphen Typen zu erzielen, muss entweder Wertsemantik eingeführt (der Container wird bei der Zuweisung kopiert) oder die Modifikation des Inhalts des Containers („mutation") über den Alias eingeschränkt werden. In Java ginge letzteres über den Wildcard-Mechanismus, also etwa die Deklaration

   ```
   List<? extends Object> os = ks;
   ```

   die das Hinzufügen oder Überschreiben von Elementen der durch `ks` benannten Liste über den Alias `os` verbietet [Torgersen u. a., 2004]. Anders verhält es sich bei einem möglichen Subtyping von `Optional` und `List`: `List` zu einem Subtyp von `Optional` zu machen verbietet sich von selbst (mehrere Objekte dürfen nicht auftreten, wo kein oder ein Objekt verlangt wird), und `Optional` zu einem Subtyp von `List` zu machen würde einen Mechanismus verlangen, der verhindert, dass man über einen Alias vom Typ `List` einem Container vom Typ `Optional` Elemente hinzufügt, und zwar unabhängig davon, ob der Elementtyp ko- oder kontravariant deklariert ist.

3. **Anzahl bestimmt die Codierung von „kein Objekt"** Ob (wiederum bei Fortbestehen obiger Deklarationen) `k` kein Konto bezeichnet, wird durch den Test `k==null` geprüft; für `ks` hingegen wäre der Test eher `ks.isEmpty()` (`ks==null` hätte dann eine andere Bedeutung, beispielsweise, dass `ks` noch nicht initialisiert wurde).

4. **Anzahl bestimmt Null-Sicherheit** Bei der Verwendung von Einzahl muss beim Zugriff auf die Eigenschaften eines Objekts sichergestellt sein, dass das Ergebnis des Ausdrucks, der das Objekt liefert, nicht „kein Objekt" ist, also etwa

   ```
   if (k != null) k.aufloesen();
   ```

so dass kein Laufzeit(typ)fehler auftritt. Bei der Verwendung von Mehrzahl ist dies hingegen nicht notwendig:

```
ks.stream().forEach(k -> k.aufloesen());
```

funktioniert auch, wenn `ks.isEmpty()` zutrifft (vgl. Punkt 3).

5. **Anzahl bestimmt Form der Verkettung** Bei der Verwendung von Einzahl sind verkettete Zugriffe auf Eigenschaften von Objekten wie in

   ```
   k.besitzer.anschreiben()
   ```

   kein Problem, sieht man einmal von der mangelnden Null-Sicherheit (s. o.) ab. Bei der Verwendung von Mehrzahl hingegen sind wiederum komplexe Konstrukte notwendig, so z. B., wenn auch `besitzer` Mehrzahl ist,

   ```
   ks.stream().flatMap(k -> k.besitzer).forEach(b -> b.anschreiben());
   ```

6. **Anzahl bestimmt Art der Kapselung** Während es in der objektorientierten Programmierung üblich ist, auf Felder mittels sog. Accessoren (Getter und Setter) Zugriff zu gewähren, gilt dies für containerwertige Felder zumindest dann nicht, wenn diese Beziehungen zu mehreren Objekten umsetzten sollen: Stattdessen werden dann Methoden zum Zugriff auf einzelne Elemente der Container angeboten und wenn doch einmal alle Elemente herausgegeben werden sollen, wird für diesen Zweck ein neuer Container verwendet.

7. **Anzahl bestimmt Aliasing** Während in

   ```
   Konto backup = k; k = null; k = backup;
   ```

   die Zuweisung an `backup` ihren Zweck erfüllt, tut sie es in

   ```
   List<Konto> backup = ks; ks.clear(); ks = backup;
   ```

   nicht: `ks` ist nur ein Alias für den Container, nicht für die Konten, die er enthält.

8. **Anzahl bestimmt Aufrufsemantik** Analog verhält sich ein Aufruf von

   ```
   static void clear(Konto k) { k = null; }
   ```

   wie ein Call-by-value (d. h., der aktuelle Parameter wird nicht auf „kein Konto“ gesetzt), der von

   ```
   static void clear(List<Konto> ks) { ks.clear(); }
   ```

   hingegen logisch eher wie ein Call-by-reference (der aktuelle Parameter wird auf „keine Konten“ gesetzt).

9. **Anzahl bestimmt die Bedeutung von „unveränderlich"** Während die Annotation `final` in

   ```
   final Konto k = new Konto();
   ```

   bewirkt, dass `k` kein neues Konto zugewiesen werden kann, gilt dies für

   ```
   final List<Konto> ks = new ArrayList<>();
   ```

   nicht: `ks` können beliebig Konten hinzugefügt werden.

Nun mag man sich zurücklehnen und sagen: „Da sieht man wieder einmal, wie problembeladen die imperative Programmierung ist — in der funktionalen Programmierung hat man diese Probleme nicht!" Das stimmt jedoch nur teilweise, den nicht alle genannten Probleme haben mit Zustand oder Veränderlichkeit zu tun und der Einsatz von Monaden wie `Maybe` oder `List` kann die Brüche in der Verarbeitung von keinem, einem oder mehreren Objekten nur abmildern, nicht aber beseitigen. Für die objektorientierte Programmierung hingegen mögen die Brüche idiomatisch und hinreichend akzeptiert sein, so dass kaum jemand noch darüber stolpert; ärgerlich sind sie jedoch allemal, insbesondere, wenn man bedenkt, dass sie in der objektorientierten Modellierung (mit UML und OCL oder Alloy [Jackson, 2006] etwa) nicht auftreten [Steimann, 2015]. Nicht zuletzt (und das gilt auch für die funktionale Programmierung) mag man sich den Aufwand vor Augen halten, den ein Wechsel von Einzahl zu Mehrzahl oder umgekehrt für die Pflege eines Programms bedeutet: Selbst wenn der Compiler einem, nach der Änderung der unmittelbar betroffenen Deklarationen, alle weiteren Stellen im Programm anzeigt, die ebenfalls geändert werden müssen, so dienen die Folgeänderungen, zumindest im Fall eines Wechsels von Einzahl zu Mehrzahl, doch nur der Wiederherstellung des ursprünglichen Verhaltens des Programms für den Fall, in dem trotz deklarierter Mehrzahl zunächst nur ein Objekt vorliegt (wie im Beispiel zu Punkt 1 oben), und sollten damit eigentlich vermeidbar sein.

2. Die Einführung des Numerus in die Programmierung

Sämtliche aufgezeigten Brüche lassen sich in einem Streich beseitigen, indem man Container durch containerlose Vielzahlen ersetzt, also neben der Einzahl, dem Singular, auch eine Mehrzahl, den Plural, zulässt. Im Fall der objektorientierten Programmierung muss somit der Pointer auf einen Container mit vielen Objekten durch viele Pointer, nämlich einen pro Objekt, ersetzt werden. Da die vielen Pointer nicht zu einem Ganzen (einem Objekt) zusammengefasst sind, lassen sie sich auch nicht aliasen; stattdessen müssen bei jeder Zuweisung immer alle

Pointer kopiert werden.[1] Es verschwindet dadurch nicht nur die Indirektion durch den Container und die damit verbundene Typänderung, sondern auch alle Effekte, die mit dem Aliasing von Containern verbunden waren. Um die Behandlung von Singular und Plural vollständig aneinander anzugleichen, ist es zudem notwendig, den Zugriff auf eine Eigenschaft mehrerer Objekte (in Ausdrücken wie `ks.aufloesen()` oder `ks.besitzer.anschreiben()`) implizit zu „mappen", wobei das Mapping ebenfalls containerlos erfolgt.

Die (zunächst provisorische) Einführung von Singular und Plural mittels zweier Annotationen `singular` und `plural` als Ausprägungen einer neuen semantischen Kategorie *Numerus* (wobei, grammatikalisch nicht ganz korrekt, `plural` `singular` umfasse) bedeutet auf obige Liste von Punkten bezogen:

1. **Typ und Numerus sind orthogonale Konzepte** Die Deklarationen

   ```
   singular Konto k = new Konto();
   plural Konto ks = k;
   ```

   sind gleichermaßen wohlgetypt und numerus-korrekt; die Zuweisung `k=ks` hingegen wäre zwar ebenfalls typ-, jedoch nicht numerus-korrekt. Außerdem könnte man `k` kein weiteres Konto hinzufügen (etwa durch `k=k+new Konto()`, da der Operator + stets einen Plural liefert), `ks` hingegen schon. Zudem lässt sich aus der Typkorrektheit von `k.aufloesen()` auch die von `ks.aufloesen()` folgern; die Bedeutung des zweiten Ausdrucks wäre, die Methode `aufloesen()` auf allen Objekten, die `ks` benennt, aufzurufen.

2. **Subtyping hängt nicht vom Numerus ab** Die Zuweisung in

   ```
   plural Object os = ks;
   ```

   ist kein Problem, weil hierbei kein Alias auf einen Container entsteht; stattdessen werden alle Pointer von `ks` in `os` kopiert. `os` können damit beliebige Objekte hinzugefügt werden, ohne dass dadurch die Typinvariante von `ks` verletzt würde.

3. **Codierung von „kein Objekt" ist universell** Die Abwesenheit eines Objekts eines Typs `T` kann universell durch einen Konstruktor `no T` (das Pendant zu `new T()`) dargestellt werden. Die Abfrage auf „kein Objekt" würde zweckmäßigerweise mit einem „Laufzeitanzahltest" (analog zum Laufzeittyptest) erfolgen, beispielsweise für einen Ausdruck `e` durch `|e|==0`.

4. **Null-Sicherheit ist universell gegeben** Durch das implizite Mapping beim Zugriff auf die Eigenschaften keines, eines oder mehrere Objekte kann es

[1] Dem dadurch entstehenden Mehraufwand kann durch die übliche Compiler-Magie begegnet werden, so z. B. durch Copy-on-write.

weder bei einem Singular noch bei einem Plural zu einem Null-Zeiger-Fehler kommen; eine explizite Fallunterscheidung ist dazu nicht nötig (kann aber mittels `|e|` jederzeit vorgenommen werden).

5. **Form der Verkettung ist universell** Die Bedeutung einer Verkettung wie `k.besitzer.anschreiben()` oder `ks.besitzer.anschreiben()` ist einheitlich (und entspricht im wesentlichen dem Flat mapping).
6. **Kapselung ist einheitlich** Da es keine Container zur Codierung von Mehrzahlen braucht, braucht es auch keine Unterscheidung der Kapselung beim Feldzugriff — es gelten vielmehr die Regeln für den Singular auch für den Plural.
7. **Aliasing ist einheitlich** Die Zeilen

   ```
   singular Konto backup = k; k = no Konto; k = backup;
   ```

   und

   ```
   plural Konto backup = ks; ks = no Konto; ks = backup;
   ```

   verhalten sich einheitlich.
8. **Aufrufsemantik ist einheitlich** Gleiches gilt für die Aufrufe von

   ```
   static void clear(singular Konto k) { k = no Konto; }
   ```

   und

   ```
   static void clear(plural Konto ks) { ks = no Konto; }
   ```

   Beide zeigen das logische Verhalten von Call-by-value. Da `singular` von `plural` subsumiert wird, kann die erste Methode zudem entfallen.
9. **Bedeutung von „unveränderlich“ ist einheitlich** Die Annotationen `final` in

   ```
   final singular Konto k = new Konto();
   final plural Konto ks = k;
   ```

   bewirken beide dasselbe: `k` und `ks` stehen fortan immer für dasselbe Konto.

Wie in diesen Beispielen bereits angedeutet wird für die Deklaration und statische Überprüfung des Numerus ein eigenes System benötigt, das, zur Abgrenzung vom Typsystem, *Numerus-System* genannt werden soll.[2] Wie üblich besteht dies aus Annotationen von Programmelementen und Inferenzregeln. Eine vollständige Ausarbeitung findet sich bei Steimann [2021a]; hier wird dagegen nur ein grober Überblick vermittelt.

[2] Existierende statische Prüfungen der Anzahl, so z. B. „genau eines“ (entsprechend „not null“), aber auch anderer Zahlen wie in Cω [Bierman, Meijer und Schulte, 2005] oder JavaFX [Topley, 2010], sind in der Literatur in der

2.1. Numerus-Deklarationen

Wie bereits in den obigen Beispielen angedeutet bedingt die Einführung eines Numerus in die Programmierung die Annotation von all jenen deklarierten Programmelementen, die zur Laufzeit für Objekte stehen. Wie bei Typen auch können solche Annotationen im Programm explizit angegeben oder inferiert werden. Anstelle der oben provisorisch verwendeten Annotationen `singular` und `plural` bieten sich hierfür `!` für genau eines, `?` für eines oder keines und `*` für beliebig viele an, wie sie auch von anderen Autoren im ähnlichen Zusammenhang schon verwendet wurden (so z. B. von Bierman, Meijer und Schulte [2005]). In objektorientierten Programmiersprachen wie etwa Java können solche Annotation dann überall da auftreten, wo ein Referenztyp den Typ von Objekten festlegt, also in Variablen-, Feld-, Parameter und Rückgabewertdeklarationen.

Methoden im Plural Wenn man schon einmal dabei ist, einen Plural einzuführen, dann sollten auch Methoden selbst als Singular (`!`) oder Plural (`*`) ausgezeichnet werden können [Steimann, 2021a]: Eine Methode im Plural entspräche dann der Pluralform des Prädikats eines Satzes und würde festlegen, dass das Subjekt, im Methodenrumpf von Methoden im Singular durch `this` repräsentiert, der Plural `these` sein muss (die grammatikalisch erforderliche Numerus-Kongruenz von Subjekt und Prädikat[3]), wobei `these` dann für die Gesamtheit der Empfänger eines Methodenaufrufs (ggf. auch kein Objekt) steht. Beim Aufruf einer solchen Pluralmethode würde das (bei Singularmethoden notwendige) Mapping über die einzelnen Objekte des Empfängers entfallen und der Methodenaufruf statisch gebunden. Pluralmethoden verallgemeinern somit statische Methoden [Steimann, 2021a]).

2.2. Statische Numerus-Prüfung

Sind alle Programmelemente numerus-annotiert, dann lässt sich die Numerus-Korrektheit eines Programms analog zu seiner Typkorrektheit bestimmen. Beispielhaft (und stark vereinfacht) sei hier eine Numerus-Regel für eine syntakti-

Regel in Typsysteme integriert und tatsächlich sind die Mittel, mit denen Numerus-Korrektheit eines Programms statisch durchgesetzt werden kann, genau dieselben wie bei der Typkorrektheit. Dennoch unterscheiden sich, wie die Beispiele gezeigt haben sollten, Typ und Numerus in der Programmierung genauso wie Numerus und Genus in der deutschen Sprache und der Numerus kann weitgehend unabhängig vom Typ behandelt [Steimann, 2021a] und sogar in typlosen Sprachen gewinnbringend eingesetzt werden [Steimann, 2021b].

[3] Der letzte Satz des Abstracts beinhaltet übrigens einen Numerus-Fehler, der sich nur durch eine größere Umstellung beseitigen lässt. Wer erkennt ihn?

sche Form `y = x.f;` angegeben:

$$\frac{\mathcal{N}(y) \geqslant \mathcal{N}(x) \cdot \mathcal{N}(f)}{\mathcal{N} \vdash \boxed{\texttt{y = x.f;}}\ \text{OK}}$$

wobei hier $\mathcal{N}$ eine Numerus-Umgebung, die Relation $\cdot$ durch die Tabelle

·	!	?	*
!	!	?	*
?	?	?	*
*	*	*	*

gegeben und $* > ? > !$ sei. Eine vollständige Ausarbeitung für eine objektorientierte Kernsprache mit statischer Typ- und Numerus-Prüfung findet sich bei Steimann [2021a].

2.3. Warum die Beschränkung auf Objekte?

Natürlichsprachlich ist der Plural den sog. Zählbaren vorbehalten — Nomen dürfen nur im Plural stehen, wenn sie Zählbares bezeichnen. Zählbarkeit hängt im allgemeinen am Besitz einer Identität und ist somit Individuen oder, im Jargon der Programmierung, *Objekten* vorbehalten, also solchen, von denen mehrere, selbst wenn sie sich gleichen, niemals dieselben sind. Das vornehmstes Merkmal von Individuen ist, dass sie nicht gleichzeitig an mehreren Orten sein können. Für Werte gilt das hingegen nicht: `true` etwa ist überall, wo etwas wahr ist, und `2` ist überall, wo es von etwas zwei gibt oder etwas mit zwei gemessen wird [Russell, 1912; Steimann, 2021c]. Solche Werte, die man in der Philosophie zur Abgrenzung von Individuen auch Universalien nennt, sind Abstraktionen, die wir nicht zählen können [MacLennan, 1982]; wir können von ihnen also auch keine Mehrzahl haben.[4] Der Numerus, wie er in dieser Arbeit vorgeschlagen ist, bleibt somit der Welt der Objekte (im Sinne der objektorientierten Programmierung, also definiert durch den Besitz einer Identität) vorbehalten; für Werte soll es keinen Plural geben (und den Fall „keinen Wert“ i. Ü. auch nicht).

Abgesehen von dieser eher sprachphilosophischen Begründung, die man als für die Programmierung wenig relevant erachten kann, muss man fragen, ob die Einführung einer Mehrzahl auch für Werte nicht unausweichlich zu APL-artigen Programmiersprachen führt, also zu Sprachen, in denen die Vorkommen mehrere Werte selbst wieder Werte sind (Vektoren etwa), die Mehrzahl also nicht ohne Container auskommt (und durch diese zur Einzahl reifiziert wird). Auch

[4] Wenn wir trotzdem einmal von zwei Zweien sprechen, dann sind damit zwei Individuen gemeint, die an unterschiedlicher Stelle stehen (auf einem Blatt Papier etwa) und somit unterschieden werden können.

würde die Integration von „kein Wert" bei Wahrheitswerten die Einführung einer ternären Logik bedingen, wie man am Beispiel von C♯ mit seinen `Nullable<T>` Werttypen sehr schön sieht. Zwei Werte, also etwa `true` und `false`, als Ergebnis der Auswertung eines logischen Ausdrucks wären wohl am ehesten im Bereich des Nichtdeterminismus zu verorten, um den es hier aber ebenfalls nicht gehen soll (mehr dazu in den einleitenden Bemerkungen von Abschnitt 3.2)

Anstatt also nun für alles auch einen Plural vorzusehen, möchte ich die Zählbarkeit als eine Verallgemeinerung der „Nullbarkeit" (nullability) von Referenz- oder Objekttypen, wie man sie aus der objektorientierten Programmierung kennt, verstanden wissen. Entsprechend verallgemeinert die statische Prüfung des Numerus die statische Prüfung auf „nicht Null" und korrigiert so Hoares Billion-dollar mistake quasi en passant.

3. Von der funktionalen zur (objekt-)relationalen Programmierung

Wie wir aus dem Mathematikunterricht wissen, sind Funktionen rechtseindeutige Relationen, die jedem Element ihres Definitionsbereichs höchstens ein Element ihres Wertebereichs zuordnen. Nicht ganz so geläufig ist der Umstand, dass man die Funktionsanwendung als einen Spezialfall der Relationskomposition [Tarski, 1941; Pratt, 1992] auffassen kann: Wenn f eine einstellige Funktion ist und R eine zweistellige Relation mit

$$R = \{(x, f(x)) \mid x \in dom(f)\}$$

dann gilt

$$\forall x \in dom(f) : \{(f(x))\} = \{(x)\} \,;\, R$$

wobei hier ; der Kompositionsoperator für Relationen und $\{(\cdot)\}$ eine einstellige und -elementige Relation sei.[5]

Soweit scheint erst einmal nichts gewonnen. Jedoch ist die Relationskomposition strikt allgemeiner als die Funktionsanwendung:

▷ Selbst wenn R nicht linkstotal ist, ist die Komposition $\{(x)\} \,;\, R$ immer noch für alle x wohldefiniert: für alle R und $x \notin dom(R)$ gilt $\{(x)\} \,;\, R = \{\}$. Dies steht im Gegensatz zur Funktionsanwendung $f(x)$, die im Falle einer nicht totalen, also partiellen Funktion für mache x undefiniert ist.

[5] Die Komposition von Relationen ist eng verwandt mit dem natürlichen Verbund, oder Natural join, wie er aus der Relationenalgebra [Codd, 1970] der Theorie der relationalen Datenbanken bekannt ist.

- ▻ Wenn R nicht rechtseindeutig ist, dann liefert die Komposition $\{(x)\}\,;R$ für manche x mehr als ein Element aus dem Wertebereich von R; dies kann definitionsgemäß durch eine Funktion nicht direkt ausgedrückt werden.
- ▻ Die Relationskomposition „mapped“ die Funktionsanwendung über eine beliebige Anzahl von Argumenten: Wenn wie oben $f \equiv R$, dann gilt

$$\forall n \geqslant 0, x_1, \ldots, x_n \in dom(f) : \{(x_1), \ldots, (x_n)\}\,; R = \{(f(x_1)), \ldots, (f(x_n))\}$$

- ▻ Wenn R hingegen keine Funktion (nicht rechtseindeutig) ist, dann wird das Mapping zu einem „flat mapping“, d. h.

$$\forall n \geqslant 0, x_1, \ldots, x_n \in dom(R) : \{(x_1), \ldots, (x_n)\}\,; R = \bigcup_{1 \leqslant i \leqslant n} \{(x_i)\}\,; R$$

Wie man leicht sieht, deckt dieser Übergang von Funktionen zu Relationen die Erweiterung der (objektorientierten) Programmierung um einen Plural, wie sie in Abschnitt 2 dargestellt wurde, gut ab: Man muss dazu lediglich alle Werte als einstellige (und flache) Relationen auffassen, wobei ein Objekt durch die einelementige Relation und kein Objekt durch die leere Relation dargestellt werden. Die Navigation zwischen Objekten, wie sie durch den Zugriff auf Felder oder Methoden von Objekten (member access) umgesetzt wird, entspricht dann der Relationskomposition, die die bisherigen Sonderfälle, kein Objekt und mehrere Objekte, ohne Sonderbehandlung (Fallunterscheidung) integriert. Auch Verkettungen wie `ks.besitzer.anschreiben()` gehen in der einheitlichen, einfachen Ausdrucksweise, die die Relationskomposition anstelle der Funktionsanwendung ermöglicht, auf.

Dementsprechend verlangt die Umsetzung des Paradigmenwechsels, den ein solcher Übergang von Funktionen zu Relationen in gewisser Weise darstellt, auf programmiersprachlicher Ebene gar nicht viel: Alle Variablen einschließlich Felder (wie etwa `besitzer` im obigen Beispiel) stehen intern für einstellige Relationen, die jedoch nach außen nicht in Erscheinung treten — der Inhalt der Variablen ist vielmehr eine Anzahl von Objekten eines bestimmten Typs (und keine Relation oder andere Container). Feldzugriffe und Methodenaufrufe, die nach wie vor nach außen die Form von Funktionsanwendungen haben, werden intern wie Relationskompositionen verarbeitet; die (einstelligen) Relationen, die sie intern liefern, sind außen wieder Anzahlen von Objekten. Das einzige, das man akzeptieren muss, ist, dass an einer Stelle, an der üblicherweise genau ein Objekt auftritt, nun fallweise auch mehrere auftreten können, ohne dass dafür eine etablierte mathematische oder programmiersprachliche Fassung herangezogen würde. Es ist in gewisser Weise so, als ob man die Mathematik in die

Zeit vor Cantor und seinen Mengen zurückversetzen würde: Damals waren viele einfach viele — ein Plural eben.

3.1. Ein „nummeriertes" Lambda-Kalkül?

Angesichts der Tatsache, dass sich — nach obiger Darstellung — am grundlegenden Programmiermodell durch die Einführung des Plurals nicht viel ändert (wohl aber an seiner Idiomatik!), könnte man fragen, ob man nicht einfach auch im Lambda-Kalkül Werte durch einstellige Relationen, oder Mengen, ersetzt und die Funktionsanwendung auf Mengen entsprechend „mapped". Jedoch sind Mengen selbst Singulare (lediglich ihr Inhalt ist ein Plural) und sollten damit an der Oberfläche nicht erscheinen. Außerdem sind sie, wie schon Hehner [1993] bemerkte, für den Zweck, Viele darzustellen, unnötig komplex: Mengen können Mengen als Elemente enthalten und sind somit, im Gegensatz zu Pluralen, strukturbildend. Stattdessen bietet es sich an, Viele als Strings von Termen darzustellen. Der Rest dieses Abschnitts fabuliert über eine entsprechenden Erweiterung des Lambda-Kalküls, die über ein Numerus-System und entsprechende Annotationen wo gewünscht auf das klassische Lambda-Kalkül reduziert werden können sollte.

Ausgehend vom einfachen Lambda-Kalkül mit der für Strings von Termen vorbereiteten Grammatik

$$\begin{aligned} t &::= x \mid (\lambda x.t) \mid (t \mid t) \\ v &::= (\lambda x.t) \end{aligned}$$

könnte man zunächst die Auswertung auf viele Terme ausdehnen, also $(t \mid \bar{t})$ als Term akzeptieren (wobei $\bar{t}$ für einen String von Termen stehen soll; nicht zu verwechseln mit den Argumenten einer mehrstelligen Funktion — auch hier sind alle Funktionen einstellig). Die Auswertung einer Funktionsanwendung auf mehrere Terme könnte man dann, in Anlehnung an ihre obige Betrachtung als Relationskomposition, per

$$((\lambda x.t) \mid v_1 \ \ldots \ v_n) \longrightarrow t[x/v_1] \ldots t[x/v_n]$$

(mit $v_1 \ \ldots \ v_n$ als String von Werten) regeln, was einem Mapping der Anwendung über alle Argumente entspräche, ohne dass jedoch Duplikate aus dem Ergebnis eliminiert würden (wie es bei einer Relationskomposition der Fall wäre). Syntaktisch wären damit aber auch $\bar{t}$ und $(\bar{t} \mid \bar{t})$ als Terme zuzulassen, wobei letzteres, wenn man immer nur genau eine Funktion auf Terme anwenden möchte, durch ein entsprechendes Numerus-System wieder auszuschließen wäre.

Wenn man nun weiterhin neben Funktionen auch nicht linkstotale oder nicht rechtseindeutige Relationen abdecken möchte, könnte man zudem $(\lambda x.\bar{t})$ als Term zulassen und die obige Auswerteregel zu

$$((\lambda x.t_1 \ \ldots \ t_n) \mid v_1 \ \ldots \ v_m) \longrightarrow t_1[x/v_1] \ldots t_1[x/v_m] \ldots t_n[x/v_1] \ldots t_n[x/v_m]$$

verallgemeinern. Man beachte, dass sowohl n als auch m gleich 0 sein können; in beiden Fällen würde zu keinem Term (einer Normalform!) ausgewertet.

3.2. Objektrelationale Programmierung

Die oben skizzierte Erweiterung des Lambda-Kalküls steht in einem gewissen Widerspruch zur in Abschnitt 2 gemachten Beschränkung der Anwendung des Numerus auf Objekte und dem damit einhergehenden Ausschluss von vielen Werten anstelle eines einzelnen. Tatsächlich soll die Erweiterung auf Viele hier auch nicht der Einführung eines Nichtdeterminismus in die (funktionale) Programmierung (wie etwa bei Søndergaard und Sestoft [1992], Hehner [1993], Morris und Bunkenburg [2001] oder Antoy und Hanus [2010] zu finden) dienen, sondern die Navigation von einer Anzahl von Objekten zu einer Anzahl von Objekten ermöglichen, wie sie in der Datenmodellierung und darauf aufbauenden Programmen ständig vorkommt. Eine solche Navigation wird durch den Join-Operator der Relationenalgebra [Codd, 1970], der im wesentlichen der Relationskomposition entspricht, abgedeckt.

Nun mündet die Einführung des Numerus in Abschnitt 2 im wesentlichen in der Einführung von gerichteten zweistelligen Beziehungen mit Numerus-Annotationen, die angeben, ob es sich bei der Beziehung um eine (partielle) Funktion (Singular bzw. `!` oder `?`) oder um eine Relation (Plural bzw. $*$) handelt. Solche Numerus-Annotationen sind u. a. aus der relationalen Datenmodellierung bekannt und werden dort „Mapping constraints“ [Liddle, Embley und Woodfield, 1993] genannt[6]. Mögliche Mapping constraints sind für zweistellige Relationen 1:1, 1:N, N:1 und M:N sowie für dreistellige Relationen 1:1:1, 1:1:N usw., wobei eine 1 an einer Stelle einer Relation bedeutet, dass jede einzelne Kombination von Werten an den jeweils anderen Stellen der Relation höchstens einmal in der Relation vorkommen darf, die Relation also eine (partielle) Funktion der anderen Stellen auf die Stelle mit der 1 darstellt. Ein N oder M an einer Stelle bedeutet hingegen keine Einschränkung.

Nun sind Relationen in der relationalen Datenmodellierung nicht nur grundsätzlich ungerichtet, sondern können auch mehrstellig sein. Während sich drei-

[6] nicht zu verwechseln mit den sog. Kardinalitäten oder Multiplizitäten, wie man sie beispielsweise aus UML kennt

stellige gerichtete Relationen in der objektorientierten Programmierung noch durch Dictionaries haltende Felder darstellen lassen (wobei das als Schlüssel ins Dictionary verwendete Objekt den Zugriff vom Besitzer auf das referenzierte Objekt qualifiziert [Rumbaugh, 1987]), erfordert die Pflege ungerichteter Beziehungen erheblichen Programmieraufwand (s. z. B. bei Noble [2000]). Dabei sind Mapping constraints, die beim Update von Funktionen eingehalten werden müssen, noch gar nicht berücksichtigt.

Da nun aber, wie oben bereits bemerkt, Relationen Funktionen verallgemeinern, kann man die herkömmliche objektorientierte Programmierung mit ihren (als Funktionen aufgefassten) Feldern zur Darstellung von Beziehungen zwischen Objekten als Spezialfall einer objektrelationalen Programmierung verstehen, in der Beziehungen als (globale) Relationen aufgefasst werden, die sich mittels Relationskomposition navigieren lassen. Um den objektorientierten Charakter und insbesondere das Wesen der Navigation von Objekten zu Objekten beizubehalten, kann man in einer solchen objektrelationalen Programmierung verlangen, dass n-stellige Relationen immer von $n - 1$ Ein- oder Mehrzahlen von Objekten ausgehend navigiert werden müssen, man also etwa bei einer dreistelligen Relation für eine Navigation von der ersten zur dritten Stelle immer auch die Objekte der zweiten Stelle angeben muss (wie dies etwa bei einem Dictionary-Zugriff auch der Fall ist; s. o.). Will man so im Beispiel einer dreistelligen Relation R_3 von x_1 und x_2 zu x_3 navigieren, dann entspräche dies genau dem Ausdruck $x_2 \,;\, (x_1 \,;\, R_3) = x_3$, wobei x_1 und x_2 Variablen sind, die jeweils kein, ein oder mehrere Objekte enthalten, und das Ergebnis der zweifachen Komposition dem (berechneten) Wert von x_3 (wiederum kein, ein oder mehrere Objekte) entspricht. Zentral ist also auch hier die Einführung des Plurals für Ausdrücke (wie Variablen), die zu Objekten auswerten; ein statisches Numerus-System kann zudem hergeben, dass die Variable x_3 den statischen Numerus Singular nur dann haben kann, wenn auch x_1 und x_2 statisch Singulare sind und das Mapping constraint der Relation R_3 zudem eine 1 an der dritten Stelle anführt (also etwa M:N:1).

Während das Navigieren in einer solchen Konstellation leicht darzustellen ist (tatsächlich ist die Relationskopmposition ausreichend, um eine Gleichwertigkeit der objektrelationalen Programmierung zur Relationenalgebra als Abfragesprache herzustellen), verhält es sich mit dem Updating der Relationen schwieriger, da hier die Einhaltung der Mapping constraints statisch garantiert werden muss. Die vollständige Ausarbeitung einer Lösung ist vom Autor (Steimann [2021b]) erhältlich.

4. Fazit

Die Beschränkung der in Programmiersprachen ausdrückbaren Objektbeziehungen auf Funktionen ist eben genau das: eine Beschränkung. Will man Beziehungen zu keinem oder mehreren Objekten ausdrücken, bedarf es umständlicher Konstrukte, die bei einer relationalen Betrachtung unnötig sind. In klassischen relationalen Sprachen wie Prolog [Emden und Kowalski, 1976] haben Variablen in der Regel „singular semantics" [Søndergaard und Sestoft, 1992], d. h., sie halten von mehreren zu einer Lösung gehörenden Werten jederzeit höchstens einen, aber schon die constraint-logische Programmierung weicht davon ab. Bei den Darstellungen in den Abschnitten 2 und 3.2 haben Variablen „plural semantics"; anders als Prolog dienen sie jedoch nicht der Ableitung von Relationen (wie es auch bei Datalog der Fall ist), sondern der Navigation in Objektgraphen, wie sie für die objektorientierten Programmierung typisch ist.

Literaturverzeichnis

Antoy, Sergio und Michael Hanus (2010). „‚Functional logic programming'". In: *Commun. ACM* 53.4, S. 74–85. DOI: 10.1145/1721654.1721675.

Backus, John W. (1978). „‚Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs'". In: *Commun. ACM* 21.8, S. 613–641. DOI: 10.1145/359576.359579.

Bierman, Gavin M., Erik Meijer und Wolfram Schulte (2005). „‚The Essence of Data Access in Cω'". In: *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Proceedings*. Hrsg. von Andrew P. Black. Bd. 3586. Lecture Notes in Computer Science. Springer, S. 287–311. DOI: 10.1007/11531142_13.

Codd, Edgar F. (1970). „‚A Relational Model of Data for Large Shared Data Banks'". In: *Commun. ACM* 13.6, S. 377–387. DOI: 10.1145/362384.362685.

Emden, Maarten H. van und Robert A. Kowalski (1976). „‚The Semantics of Predicate Logic as a Programming Language'". In: *J. ACM* 23.4, S. 733–742. DOI: 10.1145/321978.321991.

Hehner, Eric C.R. (1993). A Practical Theory of Programming. New York: Springer-Verlag. DOI: 10.1007/978-1-4419-8596-5.

Jackson, Daniel (2006). Software Abstractions - Logic, Language, and Analysis. MIT Press. ISBN: 978-0-262-10114-1. URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928.

Liddle, Stephen W., David W. Embley und Scott N. Woodfield (1993). „‚Cardinality Constraints in Semantic Data Models'". In: *Data Knowl. Eng.* 11.3, S. 235–270. DOI: 10.1016/0169-023X(93)90024-J.

MacLennan, Bruce J. (1982). „‚Values and Objects in Programming Languages'". In: *ACM SIGPLAN Notices* 17.12, S. 70–79. DOI: 10.1145/988164.988172.

Morris, Joseph M. und Alexander Bunkenburg (2001). „‚A theory of bunches'". In: *Acta Informatica* 37.8, S. 541–561. DOI: 10.1007/PL00013316.

Noble, James (2000). „‚Basic Relationship Patterns'". In: *Pattern Languages of Program Design*. Hrsg. von Neil Harrison, Brian Foote und Hans Rohnert. Bd. 4. Addison-Wesley. Kap. 6, S. 73–94.

Pratt, Vaughan R. (1992). „‚Origins of the Calculus of Binary Relations'". In: *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*. IEEE Computer Society, S. 248–254. DOI: 10.1109/LICS.1992.185537.

Rumbaugh, James E. (1987). „‚Relations as Semantic Constructs in an Object-Oriented Language'". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*. Hrsg. von Norman K. Meyrowitz. ACM, S. 466–481. DOI: 10.1145/38765.38850.

Russell, Bertrand A. W. (Juni 1912). „‚I.—On the Relations of Universals and Particulars'". In: *Proceedings of the Aristotelian Society* 12.1, S. 1–24. DOI: 10.1093/aristotelian/12.1.1.

Søndergaard, Harald und Peter Sestoft (1992). „‚Non-Determinism in Functional Languages'". In: *Comput. J.* 35.5, S. 514–523. DOI: 10.1093/comjnl/35.5.514.

Steimann, Friedrich (2015). „‚None, One, Many - What's the Difference, Anyhow?'" In: *1st Summit on Advances in Programming Languages, SNAPL 2015*. Hrsg. von Thomas Ball u. a. Bd. 32. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, S. 294–308. DOI: 10.4230/LIPIcs.SNAPL.2015.294.

– (2021a). „‚Containerless Plurals: Separating Number from Type in Object-Oriented Programming'". in Begutachtung.

– (2021b). „‚Objektrelationale Programmierung mit Abbildungsbedingungen'". unveröffentlichtes Manuscript.

– (2021c). „‚The Kingdoms of Objects and Values'". In: *Onward! Essays*. Hrsg. von Elisa Baniassad. im Druck.

Tarski, Alfred (1941). „‚On the Calculus of Relations'". In: *J. Symb. Log.* 6.3, S. 73–89. DOI: 10.2307/2268577.

Topley, Kim (2010). JavaFX™ Developer's Guide. Addison-Wesley Professional. ISBN: 9780321648983.

Torgersen, Mads u. a. (2004). „‚Adding Wildcards to the Java Programming Language'". In: *J. Object Technol.* 3.11, S. 97–116. DOI: 10.5381/jot.2004.3.11.a5.

Constraint-Logische Objektorientierte Programmierung mit Muli

Hendrik Winkelmann
hendrik.winkelmann@wi.uni-muenster.de
WWU Münster

Abstract
Muli ist eine Programmiersprache, welche Elemente des constraint-logischen Paradigmas mit imperativer objektorientierter Programmierung verbindet. Somit repräsentiert Muli das (vergleichsweise) neue *Constraint-Logic Object-Oriented Programming* (CLOOP) Paradigma. Hierfür wird Java 8 um das Deklarieren von *freien* Variablen, Constraints und um nicht-deterministische Ausführung erweitert. Zur Realisierung dieser Funktionalitäten verwendet Muli eine eigens-implementierte logische virtuelle Maschine. Für diese virtuelle Maschine wurden darüber hinaus *free arrays*, also Arrays mit einer freien Länge und freien Elementen, sowie *free objects*, also Objekte mit einem freien Typen, implementiert. Durch die Integration typisch objektorientierter Funktionalitäten wie dynamischem Polymorphismus mit nicht-deterministischer Ausführung stellen sich interessante Anwendungsfälle, jedoch auch (technische) Limitationen von Muli heraus. Um diese Limitationen zu überwinden, bedient sich das Mulib-Projekt eines neuen Ansatzes mit dem die aus Muli bekannten Konzepte auf einer gewöhnlichen Java virtuellen Maschine wie HotSpot ausführbar gemacht werden. Hierzu transformiert Mulib Java Bytecode in ein Format, in welchem nicht-deterministische Ausführung ermöglicht ist.

1 Muli – Ein Überblick

Muli[1] [Dageförde, 2020] ist eine Programmiersprache, welche Java 8 um freie Variablen, Constraints und nicht-deterministische Suche in gekapselten Suchregionen erweitert. Verglichen mit anderen logischen Programmiersprachen ermöglicht Muli *imperative* logische Programmierung mit einer Semantik [Dageförde und Kuchen, 2019] die beispielsweise Java-Programmierern verständlicher

[1] https://github.com/wwu-pi/muli

ist als rein deklarative Optionen. Im Folgenden werden die fundamentalen Funktionalitäten von Muli zusammengefasst, die in einer Erweiterung der Java virtuellen Maschine (JVM) hin zu einer logischen JVM implementiert sind.

1.1 Gekapselte Suche und Constraint Solving

In Muli können Entwickler Variablen in einer Suchregion als *free* deklarieren, z.B. `int i free`. Dieser Variable `i` ist kein konstanter Wert zugewiesen. Stattdessen kann `i` jeden Integer-Wert zwischen -2^{31} und $2^{31}-1$ annehmen. Mittels Constraints kann dieser Wertebereich eingeschränkt werden. Über die Verwendung von beispielsweise Vergleichsoperationen in einer Konftrollfluss-Anweisung (bspw. `if (i < 0) {...} else {...}`, wobei `i < 0` der Constraint ist) wird eine *Choice* erstellt, für die das Programm nicht-deterministisch ausgeführt wird. In der `Choice` werden Optionen hinterlegt, welche wiederum Constraints beinhalten. Wenn eine Option ausgewählt wird, bedeutet dies, dass der Rest des Programms unter der Annahme der jeweiligen Constraints ausgeführt wird. Lediglich innerhalb einer Suchregion wird Code nicht-deterministisch ausgeführt. Dies sei am folgenden Beispiel des *Taxicab number* Problems, dargestellt als Muli-Suchregion [Dageförde und Kuchen, 2020; Dageförde, 2020] erläutert:

```
1  public class HardyRamanujan {
2      public static void main(String[] args) {
3          Stream<Solution<Object>> solutions = Muli.muli(Taxicab::solve);
4          solutions.limit(1).forEach(s -> System.out.println(s.value)); }
5
6      private static int solve() {
7          int a free, b free, c free, d free, e free;
8          positiveDomain(a, b, c, d, e);
9          if (a != c && a != d &&
10             cube(a) + cube(b) == e &&
11             cube(c) + cube(d) == e) {
12             return e;
13         }
14         throw Muli.fail(); }
15
16     private static int cube(int n) {
17         return n*n*n; }
18     void positiveDomain(int... vars) {
19         for (int v : vars)
20             if (v <= 0) throw Muli.fail(); }
21 }
```

In der `main`-Methode wird eine Suchregion für die nicht-deterministische Ausführung als parameterlose Lambdafunktion, hier `Taxicab::solve`, übergeben und ein `Stream` an möglichen Lösungen wird generiert (Zeile 3).

In dieser Suchregion werden zuerst die Variablen `a`, `b`, `c`, `d` und `e` als freie Variablen deklariert (Zeile 7). Mittels des Aufrufs `positivieDomain(a, b, c, d, e)` wird dafür gesorgt, dass die Variablen nur positive Werte repräsentieren dürfen. Hierfür wird in Zeile 20 mittels `if (v <= 0)` eine `Choice` mit zwei Optionen, die jeweils einen Constraint beinhalten, erstellt: Entweder der Wert der Variable ist `<= 0`, oder der Wert der Variable ist `> 0`. Zuerst wird beispielsweise der Fall `v <= 0` untersucht. In diesem Fall wird `throw Muli.fail()` ausgeführt (Zeile 20). Diese spezielle `RuntimeException` signalisiert Muli, dass die gewählte Option nunmehr ungültig ist. Die logische JVM setzt den Stand des Programms zurück auf den Status der vorlag, bevor eine Entscheidung darüber getroffen wurde, welche Option der `Choice` ausgewertet werden soll. Dieser Vorgang wird als *Backtracking* bezeichnet. Daraufhin kann die logische JVM sich für die Option `v > 0` entscheiden. Als Konsequenz hierauf wird jede weitere Operation im verbleibenden Programm unter der Restriktion ausgeführt, dass `v > 0`, also beispielsweise `a > 0`.

Nachdem nun also bestimmt wurde, dass `a > 0`, `b > 0`, `c > 0`, `d > 0` und `e > 0`, wird als neuer Constraint hinzugefügt, dass `e=a³+b³=c³+d³`, `a≠c` und `a≠d` (Zeilen 9–11). Das Ergebnis `e` wird zurückgegeben (Zeile 12), wodurch die Suchregion verlassen wird. Bei dem Extrahieren des Rückgabewertes aus der Suchregion kann `e` nun ein Wert zugewiesen werden, der die Restriktionen erfüllt. Beispielsweise wird durch die Anweisung in Zeile 4 somit der Wert `1729` ausgegeben.

Für die Auswertung der Constraints nutzt Muli Constraint-Solver wie JaCoP [Kuchcinski, 2003] und Z3 [Moura und Bjørner, 2008]. Constraints werden durch die Wahl einer `Choice`-Option auf einen Constraint-Stack gelegt. Sollte der Constraint-Solver feststellen, dass ein Constraint, gegeben des derzeitigen Constraint-Stacks, nicht erfüllbar ist, so wird die entsprechende `Choice`-Option als ungültig markiert.

Die Wahl der nächsten `Choice`-Option wird hierbei durch einen Suchalgorithmus bestimmt. Muli unterstützt Tiefen- und Breitensuche sowie iterative Tiefensuche. Um für verschiedenste Suchalgorithmen auch Nebeneffekte zu berücksichtigen, nutzt Muli *Forward-* und *Backward-Trails*, durch welche zwischen `Choice`-Optionen auf verschiedenen Suchtiefen gewechselt werden kann. [Dageförde und Teegen, 2020]

1.2 Free Arrays

Neben primitiven freien Typen wie `int` und `double` ist es in Muli auch möglich, freie Arrays zu deklarieren. Freie Arrays sind Arrays, welche eine freie Länge haben sowie freie Elemente beinhalten. Ferner sind nicht-deterministische Array-Operationen möglich. So können auch Array-Zugriffe mit freien Indexen ausgeführt werden. [Winkelmann, Dageförde und Kuchen, 2021]

Im Folgenden sind einige Operationen aufgelistet, die die Funktionalität von freien Arrays verdeutlichen sollen.

```
1  public class FreeArraysDemo {
2      static int[] allDistinctArray(int length) {
3          int[] ar free;
4          if (ar.length != length) {
5              throw Mulib.fail(); }
6          for (int i = 0; i < ar.length; i++) {
7              for (int j = i + 1; j < ar.length; j++) {
8                  if (ar[i] == ar[j]) {
9                      throw Mulib.fail(); }
10             } }
11         return ar;
12     }
13     static boolean exceptionOrFail() {
14         int i free, j free;
15         int[] ar free;
16         if (ar.length != 1 || i == j) {
17             throw Mulib.fail(); }
18         return ar[i] == ar[j];
19     }
20 }
```

In der Methode `allDistinctArray(int)` wird ein freies `int`-Array `ar` erstellt (Zeile 3), welches genau die Länge des Wertes des `length`-Parameters haben muss (Zeilen 4–5). Da für jeden gültigen Index von `ar` ein freier `int`-Wert vorliegt, können diese Werte in ihrer gültigen Domäne eingeschränkt werden. In der Methode werden die Array-Elemente paarweise miteinander verglichen. Nach der Schleife in den Zeilen 6 bis 10 ist sichergestellt, dass jedes Element einen anderen, noch unbestimmten, Wert repräsentiert, da für jeden anderen Fall `throw Muli.fail()` (Zeile 9) ausgeführt wird.

Die Methode `exceptionOrFail()` wiederum verdeutlicht das implizite Erstellen von Constraints bei einem nicht-deterministischen Array-Zugriff. Zwei freie `int`-Variablen `i` und `j` sowie ein freies Array `ar` werden deklariert (Zeilen 14

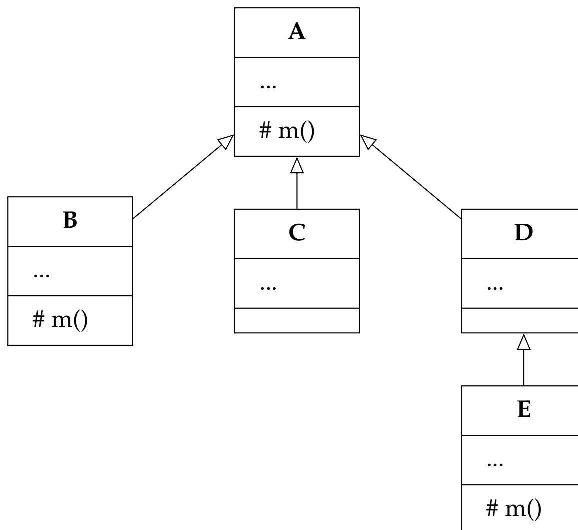
und 15). Es wird erzwungen, dass `i` und `j` jeweils unterschiedliche Werte haben, und das `ar` genau einen Wert beinhalten darf (Zeilen 16 und 17). Das Ausführen von `ar[i] == ar[j]` kann nun zwei Ergebnisse haben: Entweder, eine `ArrayIndexOutOfBoundsException` wird geworfen, da sowohl `i` als auch `j` nicht in ihrem Wertebereich eingeschränkt wurden. Die zweite Option ist, dass die Ausführung als ungültig erkannt wird: Wenn keine `ArrayIndexOutOfBoundsException` geworfen wird, müssen sowohl `i` und `j` in einem gültigen Bereich liegen. Da `ar` allerdings nur ein Element beinhalten darf, und `i`≠`j` gilt, schlägt die Auswertung fehl: Entweder `i` **oder** `j` darf den einzig gültigen Wert, 0, annehmen.

Für die Erstellung dieser Constraints wurde auf die Funktionalität von Z3 zurückgegriffen [Moura und Bjørner, 2008]. Nützlich sind freie Arrays zum Beispiel für das Auswerten von Programmcode zur Testfallgenerierung (siehe Abschnitt 1.4).

1.3 Free Objects

Jüngst wurde es auch ermöglicht, freie Objekte in Muli zu deklarieren. Solche freien Objekte haben einen freien Typen. Die Menge an möglichen Typen wird erst zur Laufzeit durch Typ-relevante Operationen und dynamischen Polymorphismus eingeschränkt. [Dageförde, Winkelmann und Kuchen, 2021]

Auch diese Funktionalität wird im Folgenden anhand eines Code-Ausschnitts erklärt. Hierbei stellt das folgende Klassendiagramm die in dem Code-Ausschnitt verwendete Klassenhierarchie dar.




```
1  public class FreeObjectsDemo {
2      static A freeCast() {
3          A obj free;
4          return (D) obj;
5      }
6
7      static A freeInstanceof() {
8          A obj free;
9          if (obj instanceof B) {
10             return (B) obj;
11         } else {
12             return obj; }
13     }
14
15     static void virtualMethodInvocation() {
16         A obj free;
17         obj.m(); }
18 }
```

In den drei aufgelisteten Methoden wird zuerst ein freies Objekt vom Typ `A` deklariert (Zeilen 3, 8 und 16). Bei der Initialisierung des freien Objektes wird kein Konstruktor verwendet. Stattdessen werden etwaige Felder als freie Variablen deklariert und eine Menge an möglichen Typen erstellt [Dageförde, Winkelmann und Kuchen, 2021]. Im Falle von `A obj free` ist die entsprechen-

de Menge $\{A, B, C, D, E\}$. In den Methoden `freeCast()` und `freeInstanceof()` wird die weitere Programmausführung explizit von dem Typen eines Objektes abhängig gemacht. Im Falle von `freeCast` wird eine `Choice` mit zwei Optionen erstellt (für Zeile 4): Entweder ist `obj` vom Typ `D` (oder einem Subtyp von `D`), oder nicht. Im ersten Fall, wird ein Objekt zurückgegeben, welches nun die möglichen Typen $\{D, E\}$ hat. Im anderen Fall wird eine `ClassCastException` geworfen, da `obj` keinen gültigen Typen im Sinne der Anweisung `checkcast` hat [Lindholm u. a., 2015, § 6.5].

Im Unterschied hierzu gibt die `obj instanceof B`-Anweisung in der Methode `freeInstanceof()` den `boolean`-Wert `true` zurück, falls `obj` ein Subtyp von `B` oder `B` selber ist. Folglich wird entweder ein freies Objekt mit dem möglichen Typ B (Zeile 10) oder den möglichen Typen $\{A, C, D, E\}$ (Zeile 12) zurückgegeben.

Zuletzt zeigt die Methode `virtualMethodInvocation()` die wohl interessanteste Funktionalität von freien Objekten. Für das freie Objekt `obj` wird die Methode `m()` aufgerufen (Zeile 17). Wie an dem Klassendiagramm erkennbar ist, wird diese Methode in drei verschiedenen Klassen implementiert: In `A`, `B` und in `E`. Sobald eine Methode aufgerufen wird, für die mehrere mögliche Typen eine Implementierung aufweisen, wird eine `Choice` erstellt. Die Menge an Optionen in dieser `Choice` ist gleich der Menge an Implementierungen für `m()` von gültigen Typen für `obj` [Dageförde, Winkelmann und Kuchen, 2021]. Im gegebenen Fall liegen also drei Optionen für die `Choice` folgend aus `obj.m()` vor: Entweder, der konkrete Typ für `obj` ist in $\{B\}$, $\{A, C, D\}$, oder $\{E\}$. Entsprechend wird entweder `B.m()`, `A.m()`, oder `E.m()` ausgeführt.

Freie Objekte dieser Art werden von keinem Constraint Solver unterstützt. Stattdessen implementiert Muli diese durch einen frei an bestehende Constraint-Solver anbindbaren Solver, der Mengen-Operationen evaluiert. Freie Objekte ermöglichen die effiziente Testfallgenerierung zu Klassenhierarchien, das objektorientierte Formulieren von Constraint Satisfaction Problemen, sowie das Generieren von Objektgraphen [Dageförde, Winkelmann und Kuchen, 2021] (siehe Abschnitt 1.4).

1.4 Anwendungsszenarien

Außerhalb von Suchregionen gleicht Muli Java 8. Nur innerhalb von Suchregionen werden Bytecode-Anweisungen verwendet, die nicht-deterministische Ausführungslogik berücksichtigen. Aus diesem Grund kann Muli als Ersatz für Java genutzt werden (vergleiche Abschnitt 1.5), mit der zusätzlichen Möglichkeit, Programme auch nicht-deterministisch ausführen zu können. Im Folgenden werden einige mögliche Anwendungsszenarien für nicht-deterministische Aus-

führung mit Muli zusammengefasst.

1.4.1 Suchprobleme

Muli bietet sich an, Suchprobleme mit imperativem Code darzustellen, ohne auf Framework-Klassen angewiesen zu sein. Klassische Probleme wie beispielsweise *SEND MORE MONEY*, *NQueens* und dem oben gezeigten *Taxicab number*-Problem sind leicht formulierbar und auch von Entwicklern, die vorrangig dem imperativen Paradigma zugehörig sind, gut verständlich. Auch komplexere Suchprobleme können mittels freier Objekte objektorientiert repräsentiert werden. Es ist kein Enkodieren von Typinformationen notwendig um diese Informationen in die Suche zu inkludieren (siehe Dageförde, Winkelmann und Kuchen [2021] und Dageförde und Kuchen [2020]).

1.4.2 Objektgraphgenerierung

Durch die nicht-deterministische Ausführung in Muli lassen sich verschiedene Strukturen generieren und aus der Suchregion extrahieren. Insbesondere mittels freier Objekte in Kombination mit dem Strategie-Entwurfsmuster können Objektgraphen leicht modifiziert werden. Hierfür kann eine `Strategie`-Klasse von verschiedenen konkreten Modifikationsstrategien als Subklassen erweitert werden. Mittels der iterativen oder rekursiven Anwendung von `Strategie s free; s.modify(graph)` können verschiedene Graphen generiert werden. Weitere Beschreibungen werden von Dageförde und Kuchen [2020] und Dageförde, Winkelmann und Kuchen [2021] gegeben.

1.4.3 Testfallgenerierung

Muli führt Programme vergleichbar zu Model-Checkern und Testfallgeneratoren wie Symbolic Pathfinder [Khurshid, Păsăreanu und Visser, 2003] aus. Durch das nicht-deterministische Ausführen von deterministischen Programmen lassen sich (möglichst) alle Pfade des Programms durchlaufen. Das Extrahieren von Werten aus der Suchregion liefert sogleich geeignete Werte, um einen Testfall zu generieren. Somit wird Testfallerzeugung ein Sprach-Bestandteil. Hierbei müssen allerdings noch *destruktive Updates* [Visser, Păsăreanu und Khurshid, 2004] berücksichtigt werden. Wir arbeiten aktiv daran, diese Testfallgenerierung mit bestehenden Tools zu vergleichen.

1.5 Technische Limitationen

Die in Abschnitt 1.4 vorgestellten Anwendungsfälle zeigen, dass Muli einen Mehrwert an Ausdruckskraft und neue Anwendungsfälle mit sich bringt und theoretisch, während der deterministischen Ausführung (also außerhalb von Suchregionen), wie Java 8 zu nutzen ist. Ein Hindernis dafür, Muli als "drop-in replacement" für Java 8 zu nutzen, ist die Geschwindigkeit der logischen JVM auf der Mulicode ausgeführt wird. Diese logische JVM ist in Java 8 geschrieben. Durch diese Indirektion entsteht ein großer Mehraufwand, sodass auch im deterministischen Fall die Geschwindigkeit von Java 8 nicht erreicht werden kann. Die Tatsache, dass Muli auf (Forward- und Backward-)Trails basiert erschwert zudem eine parallele Untersuchung der Suchregion.

2 Mulib – Ein Ausblick

Um diese technischen Limitationen von Muli anzugehen wurde Mulib konzipiert. Mulib transformiert Klassen und Methoden die in einer Suchregion genutzt werden in eine neue Repräsentation. Diese Repräsentation wiederum lässt sich nicht-deterministisch ausführen. Eine Besonderheit von Mulib ist, dass keine eigene logische JVM benötigt wird. Stattdessen steht die Programmtransformations- und Ausführungsfunktionalität als Java **Lib**rary bereit. Da die Programmtransformation als Eingabe gültigen Java Bytecode erwartet und wiederum gültigen Java Bytecode ausgibt, hat Mulib das Potenzial nicht-deterministische Ausführung in Suchregionen für, unter anderem, Java, Kotlin, Scala und Groovy bereitzustellen. Ferner gibt es keinen Mehraufwand für die Programmausführung außerhalb von Suchregionen. Die Programmtransformation ist nur für Suchregionen erforderlich. Somit bleibt die Leistung der deterministischen Programmausführung gleich.

Ein Mulib-Prototyp wurde bereits implementiert. Mit diesem ist es möglich, Programme mit primitiven freien Variablen zu transformieren und auszuwerten. Initiale experimentelle Vergleiche zwischen Muli und Mulib zeigen für evaluierte Programme eine zehnfache bis zwanzigfache Ausführungsbeschleunigung. Beispielsweise beträgt die durchschnittliche Lösungsgeschwindigkeit für das *8-Queens*-Problem in Muli 2.056 Sekunden nach dem Aufwärmen der JVM. Mulib gibt nach 0.105 Sekunden eine Antwort aus. In beiden Fällen wurde die entsprechende JVM mit 7 vorausgehenden Iterationen aufgewärmt.[2]

[2] Es sollte erwähnt werden, dass die Programmtransformation in diesem Experiment nur in der ersten, unaufgewärmten Iteration von Mulib ausgeführt wurde. Spätere Iterationen griffen auf das bereits transformierte Programm zurück. Die Lösungsdauer der ersten Iteration mit Mulib betrug 0.218 Sekunden.

Die zukünftig zu addressierenden Herausforderungen von Mulib sind es, die fortgeschrittenen Features von Muli, also freie Arrays, freie Objekte und Testfallgenerierung, vollständig zu implementieren. Auch noch nicht addressierte Themen wie freie Arrays, die freie Objekte als Elemente haben, sind geplant. Zudem sollten Programmtransformationen erstellt werden, mittels derer Mulib ab einer beliebigen `Choice`-Option die Ausführung des Programms fortsetzen kann. Hierbei sollte auch die parallele Auswertung einer Suchregion ermöglicht bleiben.

Literatur

Dageförde, Jan C. (2020). „An Integrated Constraint-Logic and Object-Oriented Programming Language“. Diss. University of Münster.

Dageförde, Jan C. und Herbert Kuchen (2019). „A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli“. In: *Journal of Computer Languages* 53, S. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001.

– (2020). „Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming“. In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems*. Hrsg. von Pedro Lopez-Garcia, Roberto Giacobazzi und John Gallagher. LNCS. Springer.

Dageförde, Jan C. und Finn Teegen (2020). „Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming“. In: *Declarative Programming and Knowledge Management*. Hrsg. von Petra Hofstedt u. a. Bd. 12057. Lecture Notes in Artificial Intelligence, S. 199–214. DOI: 10.1007/978-3-030-46714-2_13.

Dageförde, Jan C., Hendrik Winkelmann und Herbert Kuchen (2021). „Free Objects in Constraint-logic Object-oriented Programming“. In: *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021), September 6–8, 2021, Tallinn, Estonia*. ACM. DOI: 10.1145/3479394.3479409. URL: https://doi.org/10.1145/3479394.3479409.

Khurshid, Sarfraz, Corina S. Păsăreanu und Willem Visser (2003). „Generalized Symbolic Execution for Model Checking and Testing“. In: *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'03. Warsaw, Poland: Springer-Verlag, S. 553–568. ISBN: 3540008985.

Kuchcinski, Krzysztof (2003). „Constraints-driven scheduling and resource assignment“. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3, S. 355–383. ISSN: 1084-4309. DOI: 10.1145/785411.785416.

Lindholm, Tim u. a. (2015). *The Java® Virtual Machine Specification – Java SE 8 Edition*. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (besucht am 16. 09. 2021).

Moura, L. Mendonça de und N. Bjørner (2008). „Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Hrsg. von C. R. Ramakrishnan und Jakob Rehof. Bd. 4963. Lecture Notes in Computer Science. Springer, S. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

Visser, Willem, Corina S. Păsăreanu und Sarfraz Khurshid (Juli 2004). „Test Input Generation with Java PathFinder". In: *SIGSOFT Softw. Eng. Notes* 29.4, S. 97–107. ISSN: 0163-5948. DOI: 10.1145/1013886.1007526.

Winkelmann, Hendrik, Jan C. Dageförde und Herbert Kuchen (2021). „Constraint-Logic Object-Oriented Programming with Free Arrays". In: *Functional and Constraint Logic Programming*. Hrsg. von Michael Hanus und Claudio Sacerdoti Coen. Cham: Springer International Publishing, S. 129–144. ISBN: 978-3-030-75333-7.

Heterogene Übersetzung von echten Funktionstypen in Java-TX

Etienne Zink
i19056@hb.dhbw-stuttgart.de
Duale Hochschule Baden-Württemberg, Campus Horb

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

Abstract
Dieser Artikel erläutert zuerst die Probleme, welche mit der homogenen Übersetzung von Typparametern (*Type Erasure*) in Java einher gehen. Auf Grundlage dessen wird ein Ansatz vorgestellt, wie eine heterogene Übersetzung umgesetzt werden könnte. Dieser wurde unter Berücksichtigung bereits bekannter Konzepte von Odersky, Runne und Wadler., 2000 und Kennedy und Syme, 2001 entwickelt. Eine Umsetzung soll dabei im Rahmen des Projekts *Java-TX* (Plümicke, 2021) erfolgen. Zur Einschränkung des Ansatzes, wurde dieser vorerst für die in Java-TX eingeführten echten Funktionstypen formuliert. Abschließend erfolgt eine kurze Betrachtung hin zu einer Erweiterung auf beliebige parametrisierte Typen.

1 Einführung

Java-TX (**T**ype e**X**tended) Plümicke, 2021 ist eine Erweiterung von Java. Diese umfasst im Wesentlichen sowohl echte Funktionstypen für Lambda-Ausdrücke, als auch globale Typinferenz[1].

Die echten Funktionstypen werden hierbei durch den sogenannten „strawman approach“ Reinhold, 2009 implementiert. Anders als im „strawman approach“ bei der echte Funktionstypen als strukturelle Typen eingeführt werden, werden

[1] Globale Typinferenz erlaubt es im Gegensatz zu lokaler Typinferenz, dass man alle Typannotationen weglassen kann, ohne die Eigenschaft der statischen Typisierung zu verlieren. Globale Typinferenz ist bisher nur aus funktionalen Programmiersprachen bekannt. Java-TX ist die erste object-orientierte Programmiersprache die globale Typinferenz erlaubt.

in `Java-TX` ein Bündel von funktionale Interfaces Fun*N*$$, ähnlich wie im Package `java.util.Function`, eingeführt. Allerdings werden Lambda-Ausdrücke im Gegensatz zu Standard–Java explizit durch FunN$$-Typen typisiert. Diese Implementierung löst allerdings bisher nicht das Problem der *Type Erasure*. In diesem Artikel werden erste Überlegungen zur Lösung der *Type Erasure* vorgestellt.

2 Aktuelle Übersetzung generischer Parameter

Die aktuelle Implementierung des Java-Compilers übersetzt die Generics in Java homogen. Dies bedeutet, dass die Parameterinformation gelöscht wird. Diese Eigenschaft wird als *Java type erasure* bezeichnet. Der Hauptgrund für die *Type Erasure* ist die Verlangsamung der JVM bei einer heterogenen Übersetzung. Eine Verlangsamung ist damit zu begründen, dass für jeden Parameter eine eigene Klasse mit entsprechendem Typ geladen werden müsste. Damit einher geht unweigerlich ein höherer Speicherverbrauch.

Andererseits führt die *Type Erasure* zu folgenden Nachteilen:

instanceof Der Aufruf des *instanceof*-Operators mit einem parametrisierten Typ als zweiter Parameter ist bis Java 16 nicht gestattet. Somit können parametrisierte Typen in der JVM nicht differenziert werden. Der Fehler der Kompilation ist in Listing 1 dargestellt. Unter Java 16 ist die Kompilation und Ausführung dessen erfolgreich, solange der korrekte Parameter verwendet wird. Dies ist auf die neue Funktion des Pattern Matchings für den *instanceof*-Operator zurück zu führen. Jedoch wird auch hierbei der Parameter gelöscht. Daher handelt es sich hierbei um eine Inkonsistent der Implementierungen, welche keinen Vorteil zur Laufzeit generiert.

Generische Exceptions Klassen welche von Exception, genauer gesagt Throwable generalisieren, dürfen keine Parameter besitzen. Der Grund hierfür liegt in der Löschung der Parameter. Dies hat zur Folge, dass *Catch-Blöcke*, welche sich ausschließlich im Parameter unterscheiden würden, zur Laufzeit nicht unterschieden werden könnten. Ein Beispiel hierfür ist unter Listing 2 dargestellt. Nach der Kompilation könnten die Bedingung der *Catch-Blöcke* nicht unterschieden werden.

Generische Überlagerung Die Überlagerung einer Methode, bei welcher sich die Argumente ausschließlich in deren Parameter unterscheiden ist nicht gestattet. Der Grund hierfür liegt wie bei den *Catch-Blöcken* in der Unmöglichkeit die Signaturen der Methoden eindeutig zu unterscheiden. Ein Beispiel hierfür ist in Listing 3 dargestellt. Nach der *Type-Erasure* wären die Methodensignatur wie folgt:

$$\text{List} \rightarrow \text{V}$$
$$\text{List} \rightarrow \text{V}$$

statt wie zu erwarten:

$$\text{List<String>} \rightarrow \text{V}$$
$$\text{List<Integer>} \rightarrow \text{V}$$

```
InstanceOf.java:7: error: illegal generic type for instanceof
      if (list instanceof List<String>){
                              ^
1 error
```

Listing 1. Fehler bei Kompilation eines parametrisierten Typen des instanceof-Operators mit Java 15.

```java
public class Main {
  public static void main(String[] args) {
      try {
          //Do something
      } catch (CustomException<String> ces) { //Not allowed
          //Do nothing
      } catch (CustomException<Integer> cei) { //Not allowed
          //Do nothing
      }
  }
}
```

Listing 2. Fehlerhafter Code mit parametrisierten Exceptions.

```java
public class Main {

  public void f(List<String> listS){ }
  public void f(List<Integer> listI){ }

}
```

Listing 3. Fehlerhafte Überlagerung mit Parameter.

3 Heterogene Übersetzung in Java ähnlichen Sprachen

3.1 PIZZA

Eine der ersten Weiterentwicklungen von Java um die Jahrtausendwende war die Sprache PIZZA Odersky und Wadler, 1997. Sie erweitert Java um Parametrische Polymorphie (Generics), Funktionen höherer Ordnung und Abstrakte

Datentypen. PIZZA hatte großen Einfluss auf die weitere Entwicklung von Java Plümicke, 2019.

Bei der Übersetzung der Parametrischen Polymorphie in Bytecode wurden zwei Ansätze verfolgt. Neben einer homogenen Übersetzung, bei der Typparameter gelöst werden (`type erasure`, vgl. Abschnitt 4) – dieser Ansatz wurde später auch in Java verwendet – gibt es auch eine heterogene Übersetzung, bei der Typparameter auch im Bytecode enthalten sind Odersky, Runne und Wadler., 2000. Dieser Ansatz soll hier skizziert werden.

Für jede parametrisierte PIZZA Klasse C<$\overline{\texttt{A}}$[2]> werden drei Java Klassen erzeugt:

- ▻ Ein Interface `C$$I`, welches die homogene Übersetzung repräsentiert. Jede Klasse, bei der die Parameter instanziiert sind muss dieses Interface implementieren. Dadurch wird sicher gestellt, dass gemischte homogene und heterogene Übersetzung möglich sind.
- ▻ Eine Template–Klasse `C$_$0_$1_ ... _$n_$`, welche von einem speziellen PIZZA–Classloader jeweils spezialisiert wird.
- ▻ Eine statische Basis–Klasse, die alle statischen Methoden der Klasse `C` implementiert und Methoden zur Generierung neuer Instanzen von `C` bereitstellt.

Spezialisierter Classloader

Der PIZZA–Classloader ist ein Subtyp von `java.lang.ClassLoader`, der die Methode
`loadClass(String name, boolean resolve)` überschreibt. Wenn die zu ladende Klasse Parameter enthält, gekennzeichnet durch `$_` ... `_$`, wird die zugehörige Template–Klasse geladen. Die Spezialisierung des Templates erfolgt, je nachdem ob es sich um einen Referenztyp oder einen Basistyp handelt, unterschiedlich.

Die Spezialisierung für Referenztypen ist sehr einfach. Lediglich im Konstantenpool müssen die Parameter durch die jeweiligen einzusetzenden Typen ersetzt werden.

Bei den Basistypen muss darüber hinaus auch noch der Op-Code im Code–Attribut angepasst werden, da es abhängig vom jeweiligen Basistyp u.a. unterschiedliche Lade- bzw. Speicherbefehle gibt (z.B. `iload` für alle `int`–Typen und `fload` für `float`–Typen).

[2] $\overline{\texttt{A}}$ steht für einen Vektor von Typparametern $\texttt{A}_1, \ldots, \texttt{A}_n$.

Polymorphe Methoden

Neben den parametrisierten Klassen, erlaubt Java auch die Methoden–Parameter, die nur in der jeweiligen Methode gültig sind (z.B. `<A> id (A x) { return x; }`). Um den eingesetzten Typ zur Laufzeit nutzen zu können, wird bei der Übersetzung für jeden Typ-Parameter der polymorphen Methode ein weiteres Argument vom Type `Class` hinzugefügt, so dass zur Laufzeit mit Reflections auf den einsetzten Typ zugegriffen werden kann.

3.2 `C#`

Die Übersetzung der Parametrischen Polymorphie in `C#` Microsoft, 2017 ist in Kennedy und Syme, 2001 beschrieben. Dabei bedient sich `C#` einer Mischung der beiden Implementierungsansätzen *Code Spezialisierung*, wie wir sie z.B. aus `C++` kennen, und *Code Teilung*, wie es `Haskell` oder auch `Java` genutzt wird.

Die `Common Runtime Language` (`CLR`) von `.NET` nutzt sogenannte *vtables* zur Auflösung von virtuellen Methoden. Dabei ist in dem *vtable* die Signatur der jeweiligen Methode festgelegt. Die Implementierung wird über den Zeiger auf den auszuführenden Code realisiert.

Die *vtables* werden zur Übersetzung der Parametrischer Polymorphie genutzt. Zur Laufzeit wird für jede Instanziierung der Parameter einer Klasse ein spezialisierter *vtable* erzeugt, welcher die jeweiligen Parameterinstanziierungen enthält. In den `vtables` werden dann die Zeiger auf den jeweils geteilten Code gespeichert.

Wenn der Code einer Methode für eine bestimmte Parameterinstanziierung der Klasse übersetzt wird, wird zunächst geprüft, ob der jeweilige Code kompatibel zu einer bereits übersetzten Instanziierung ist. Wenn ja, werden die jeweiligen *vtables* auf diesen Code verlinkt. Wenn nein, so wird neuer Code erstellt und die *vtables* dann auf diesen neuen Code verlinkt. Zwei Instanziierung sind kompatibel, wenn u.a. der Code identisch ist. Grundsätzlich kompatibel sind Instanziierungen für alle Referenztypen.

Polymorphe Methoden

`C#` enthält ebenfalls polymorphen Methoden deren Parameter unabhängig von den jeweiligen Klassenparametern sind. Auch bei polymorphen Methoden wird der Ansatz der *Code Teilung* verwendet. D.h. bevor Code übersetzt wird, wird geprüft, ob der Code bereits für die jeweilige Parameterinstanziierung erzeugt wurde.

Bei der Übersetzung polymorpher Methoden wird zunächst bestimmt in welchen Typtermen die Methodenparameter vorkommen. In all diesen Typterme werden die jeweiligen Parameter eingesetzt. Die Menge einsetzten Typterme werden als weitere Parameter der Methode mitgegeben.

4 Problem der *Type Erasure* bei echten Funktionstypen

Die *Type Erasure* in Java führt zu Einschränkungen der echten Funktionstypen, welche in Java-TX bereits eingeführt wurden. Es tritt das folgende Problem auf: Wird eine Methode mit mehreren Instanzen des Interfaces Fun*N*$$ überladen, welche sich ausschließlich in deren Parameter unterscheiden, so ist dies nicht zulässig. In diesem Falle kompiliert das Programm nicht. Um dieses Problem zu umgehen, ist bislang eine Lösung wie in Listing 4 dargestellt nötig.

```
class OLFun {
  m(f, x) {
    x = f.apply(x+x);
    return x;
  }
}
```

Listing 4. Korrekte Überlagerung mit echten Funktionstypen.

Die berechneten Typen für `m` sind hierbei die Folgenden:

$$\begin{array}{c} \text{Fun1\$\$<Double, Double>} \times \text{ Double} \rightarrow \text{Double } \& \\ \text{Fun1\$\$<Integer, Integer>} \times \text{ Integer} \rightarrow \text{Integer } \& \\ \text{Fun1\$\$<String, String>} \times \text{ String} \rightarrow \text{String } \& \end{array}$$

Wie bereits erwähnt werden im Bytecode in Java die Parameter eines Typen nicht berücksichtigt. Diese sind zwar in den Signaturen im Bytecode vorhanden, werden jedoch lediglich für den Typcheck verwendet. Somit können diese nicht für die Überlagerung verwendet werden. Die JVM selbst betrachtet somit nur die Deskriptoren, ohne Parameter-Informationen, der Methoden . Für die in Listing 4 betrachteten Methode `m` führt dies zu folgenden Headern:

```
java.lang.Double m(Fun1$$ <java.lang.Double ,java.lang.Double >,
                     java.lang.Double);
descriptor: (LFun1$$;Ljava/lang/Double;)Ljava/lang/Double;

java.lang.Integer m(Fun1$$ <java.lang.Integer ,java.lang.Integer >,
                     java.lang.Integer);
descriptor: (LFun1$$;Ljava/lang/Integer;)Ljava/lang/Integer;
```

```
java.lang.String m(Fun1$$ <java.lang.String ,java.lang.String >,
                     java.lang.String);
descriptor: (LFun1$$;Ljava/lang/String;)Ljava/lang/String;
```

Wie zu erkennen ist, wurde der Parameter aus den Deskriptoren gelöscht und ist nur in der Signatur vorhanden. Aufgrund des angewandten Tricks, dass die Methode `m` ein zweites Argument besitzt, kann die Überlagerung dennoch aufgelöst werden. Somit kann dieser Code kompiliert werden.

Wird dieser Trick nicht angewandt, so kann der Code der Methode `m` wie folgt definiert werden:

```
class OLFun {

  m(f) {
    x;
    x = f.apply(x+x);
    return x;
  }
}
```

Listing 5. Problematische Überlagerung mit echten Funktionstypen.

Dies führt dazu, dass die neuen Header der Methode `m` nicht in deren Argumenten unterschieden werden können:

```
java.lang.Double m(Fun1$$ <java.lang.Double ,java.lang.Double >);
descriptor: (LFun1$$;)Ljava/lang/Double;

java.lang.Integer m(Fun1$$ <java.lang.Integer ,java.lang.Integer >);
descriptor: (LFun1$$;)Ljava/lang/Integer;

java.lang.String m(Fun1$$ <java.lang.String ,java.lang.String >);
descriptor: (LFun1$$;)Ljava/lang/String;
```

Anhand dieser Deskriptoren ist es der JVM nicht möglich, die Überlagerung korrekt aufzulösen. Das Argument LFun1$$ wird als einziges Argument jeder Methoden-Instanz zugeordnet.

5 Lösung zur heterogenen Übersetzung echter Funktionstypen

Um das im Abschnitt 4 aufgezeigte Problem zu lösen ist es notwendig, dass die Informationen der Parameter in den Deskriptoren vorhanden bleiben. Dies kann

ausschließlich durch eine heterogene Übersetzung der Funktion erreicht werden. Das Symbol '<' ist hierbei jedoch nicht in den Deskriptoren der Methoden erlaubt. Aufgrund dessen müssen diese anderweitig aufgebaut werden. Um dieses Problem zu umgehen, wird ein ähnlicher Ansatz wie in Odersky, Runne und Wadler., 2000 verwendet. Hierbei wird der Typ in eine Zeichenfolge übersetzt. Dies hat zur Folge, dass der Typ

$$\text{Fun}N\$\$\text{<}ty_1, ty_2, \ldots, ty_n, ty_0\text{>}$$

in die Zeichenfolge

$$\text{Fun}N\$\$\$_\$ty_1\$ty_2\$\ldots ty_n\$ty_0\$_\$$$

übersetzt. Das Zeichen \$ wird aus dem gleichen Grund wie für das Interface $\text{Fun}N\$\$$ verwendet. Laut Oracle, 2021 soll das Zeichen \$ nur für maschinell erzeugten Code verwendet werden. Somit ist davon auszugehen, dass keine weiteren Klassen, welche eine Programmiererin/ein Programmierer in dessen Sourcecode verwendet, ein \$ aufweist. Die Zeichenfolge setzt sich dabei derart zusammen:
Die spitzen Klammern um die Typparameter werden in die Zeichenfolge \$_-\$ übersetzt. Dies ist der Fall, da das Symbol \$ selbst im weiteren Verlauf als Trennsymbol der einzelnen Parameter verwendet wird. Das Zeichen _ kann laut Oracle, 2021 nicht als einzelnes Zeichen als ein Bezeichner dienen, da dieses ein Schlüsselwort repräsentiert. Somit kann es bei der Zeichenfolge \$_\$ nicht zu einer Verwechslung zwischen der Übersetzung einer spitzen Klammer und der Klasse _ als Parameter kommen. Die Übersetzung einer spitzen Klammer ausschließlich in das Symbol \$ wäre zudem korrekt. Um die Lesbarkeit zu wahren, wurde sich jedoch für die zuvor vorgestellte Variante entschieden. Somit kann der Bereich der Parameter, in der Übersetzung, eindeutig bestimmt werden.

Zu beachten ist jedoch, dass im Gegensatz zu Odersky, Runne und Wadler., 2000, der Classloader unangetastet bleibt. Dahingegen wird für jeden Typ $\texttt{Fun}N\$\$\$_\$ty_1\$ty_2\$\ldots ty_n\$ty_0\$_\$$ ein leeres Interface erstellt, welches wiederum das Basis-Interface $\texttt{Fun}N\$\$\text{<}ty_1, ty_2, \ldots, ty_n, ty_0\text{>}$ spezialisiert:

```
interface Fun$N$$$$_$$ty1$ty2$...tyn$ty0$_$$
  extends  FunN$$<ty1, ty2, ..., tyn, ty0>{ }


interface FunN$$<ty1, ty2, ..., tyn, ty0> { }
```

Somit würden für das obige Beispiel folgende Interfaces erstellt werden:

```
interface Fun1$$<T, R> { T apply(R r); }

interface Fun1$$$_$Integer$Integer$_$  extends Fun1$$<Integer, Integer>{ }

interface Fun1$$$_$Double$Double$_$ extends Fun1$$<Double, Double>{ }

interface Fun1$$$_$String$String$_$  extends Fun1$$<String, String>{ }
```

Dies führt dazu, dass die Klasse `OLFun` drei Methoden `m` mit den spezialisierten Interfaces als Argument besitzt. Es könnte ein Code ähnlich dem folgendem generiert werden:

```
public class OLFun {

  public String m(Fun1$$$_$String$String$_$ f) {
      String x = "a";
      x = f.apply(x + x);
      return x;
  }

  public Double m(Fun1$$$_$Double$Double$_$ f) {
      Double x = 1.0;
      x = f.apply(x + x);
      return x;
  }

  public Integer m(Fun1$$$_$Integer$Integer$_$ f) {
      Integer x = 1;
      x = f.apply(x + x);
      return x;
  }
}
```

6 Ausblick: Erweiterung auf beliebige parametrisierte Typen

Wie in Abschnitt 2 dargestellt ist die *Type Erasure* nicht nur bei Funktionstypen ein Problem, sondern auch bei beliebigen anderen parametrisierten Typen. Der im Abschnitt 2 dargestellte Ansatz, dass man den parameterinstanziierten Typ als Subtyp des homogen übersetzten Typ definiert, entfaltet erst bei beliebigen parameterinstanziierten Typen seine vollständige Kraft. Die Idee ist, dass nur in der homogen übersetzten Superklasse der Code der jeweiligen Methoden

enthalten ist. Die parameterinstanziierten Subklassen sind dann im Wesentlichen leere Klasse, die nur den Verweis auf ihre jeweilige Superklasse enthalten. Die Subklassen erben dann die homogen übersetzten Methoden der Superklassen. Dieser Ansatz funktioniert in Java, da Parameter nur durch Referenztypen instanziiert werden dürfen. Damit ist der Code der jeweiligen Methoden identisch. Dies ist auch bei der Übersetzung in PIZZA so. Im Abschnitt 3.1 haben wir dargestellt, dass der PIZZA-Classloader für Referenztypen in der Template–Klasse lediglich im Konstantenpool die Parameter ersetzt, den eigentlichen Op-Code aber unverändert lässt. Genauso werden in C# (vgl. Abschnitt 3.2) alle Referenztypen als kompatibel angesehen.

Example 6.1. Seien die Klasse Id und Main gegeben:

```
class Id<T> {
    T id (T x) {
        return x;
    }
}

class Main {
    public static void main(String[] arg) {
        Id<Integer> i = new Id<>();
        Id<String> s = new Id();
        Integer ii = i.id(1);
        String ss = s.id("xxx");
    }
}
```

Für die beiden parameterinstanziierten Typen Id<Integer> und Id<String> werden die beiden Subklassen

```
class Id$_$Integer$_$ extends Id<Integer> { }

class Id$_$String$_$ extends Id<String> { }
```

generiert. Die Klasse Main wird dann wie folgt übersetzt:

```
class Main {
    public static void main(String[] arg) {
        Id$_$Integer$_$ i = new Id$_$Integer$_$();
        Id$_$String$_$ s = new Id$_$String$_$();
        Integer ii = i.id(1);
        String ss = s.id("xxx");
    }
}
```

Da für jede neue Parameterinstanz von `Id` nur eine *leere* Klasse erzeugt bzw. zur Laufzeit geladen wird, wobei die homegene Superklasse nach erstmaligem Laden im Cache gehalten wird, wird sich die Laufzeit dieser heterogenen Übersetzung nur minimal gegenüber der homegenen Übersetzung erhöhen.

Dieser Ansatz führt allerdings zu einem Problem, sobald die parametrisierten Klasse selber Vererbungen enthalten. Dazu betrachten wir eine Subklasse `IdSub` von `Id`

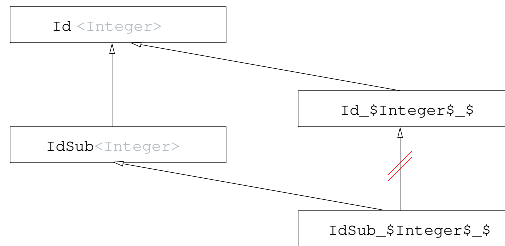


Abbildung 1. Problem Mehrfachvererbung bei parametrisierten Subklassen

Example 6.2. Sei die Klasse `IdSub` gegeben:

```
class IdSub<T> extends Id<T> {
  T idSub(T x) {
    return id(x);
  }
}
```

In Abb. 1 wird aufgezeigt, dass die parametrisierte Subklasse sowohl von der parametrisierten Superklasse als auch von der homogen übersetzten Klasse erbt.

Dieses Problem lässt sich dadurch lösen, dass man die Vererbungsbeziehungen bei den parametrisierten Klassen löscht. Diese sind nicht notwendig, da die parametrisierte Superklasse eine leere Klasse. Somit wird lediglich von der homogen übersetzten Superklasse geerbt. Diese Vererbung erfolgt aber ebenfalls indirekt über die homogen übersetzte Subklasse.

Als Ergebnis ergibt sich dann folgende übersetzt Klasse:

```
class idSub_$Integer$_$ extends IdSub<Integer> { }
```

Es lässt sich dann die Methode `idSub` aufrufen.

```
idSub_$Integer$_$ i = new idSub_$Integer$_$();
Integer ii = i.idSub(1);
```

7 Zusammenfassung

Im Rahmen des Artikels konnten einige Probleme, welche mit der homogenen Übersetzung der Typparameter einher gehen, beleuchtet werden. Es konnte hierfür eine Lösung formuliert werden. Diese beinhaltet die Übersetzung der parametrisierten Typen in eine konsistente Form. Anschließend werden für jede zu betrachtende Konstellation an Parametern, ein entsprechendes Interface generiert. Dieses spezialisiert hierbei das Basis-Interface der echten Funktionstypen aus Java-TX. Anhand des Sourcecodes konnte gezeigt werden, dass eine derartige Übersetzung möglich ist.

Literatur

Kennedy, Andrew und Don Syme (2001). „Design and Implementation of Generics for the .NET Common Language Runtime“. In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: Association for Computing Machinery, S. 1–12. ISBN: 1581134142. DOI: 10.1145/378795.378797. URL: https://doi.org/10.1145/378795.378797.

Microsoft (2017). *C# 6.0 draft specification*. [Online; Letzte Aktualisierung: 01.07.2017; Zuletzt geprüft am: 05.09.2021]. URL: https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/language-specification/introduction.

Odersky, Martin, Enno Runne und Philip Wadler. (2000). „Two Ways to Bake Your Pizza – Translating Parameterised Types into Java“. In: *Proceedings of a Dagstuhl Seminar, Springer Lecture Notes in Computer Science* 1766, S. 114–132.

Odersky, Martin und Philip Wadler (1997). „Pizza into Java: Translating Theory into Practice“. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: ACM, S. 146–159. ISBN: 0-89791-853-3. DOI: 10.1145/263699.263715. URL: http://doi.acm.org/10.1145/263699.263715.

Oracle (2021). *The Java Language Specification - 3.8. Identifiers*. [Online; Letzte Aktualisierung: 09.08.2021; Zuletzt geprüft am: 14.09.2021]. URL: https://docs.oracle.com/javase/specs/jls/se17/html/jls-3.html#jls-3.8.

Plümicke, Martin (2021). „Java–TX: The language – A Java extension with global type inference, real functions types, generated generics and heterogeneous translation of function types–“. (to appear).

– (2019). „Von Pizza zu Java-TX“. In: *Tagungsband des 20. Kolloquium Programmiersprachen und Grundlagen der Programmierung KPS 2019*. Hrsg. von Martin Plümicke und Fayez Abu Alia. Informatik. ISBN 978-3-8440-6933-4. Shaker Verlag, S. 129–140.

Reinhold, Mark (Dez. 2009). *Project Lambda: Straw-Man Proposal*. URL: http://cr.openjdk.java.net/~mr/lambda/straw-man.