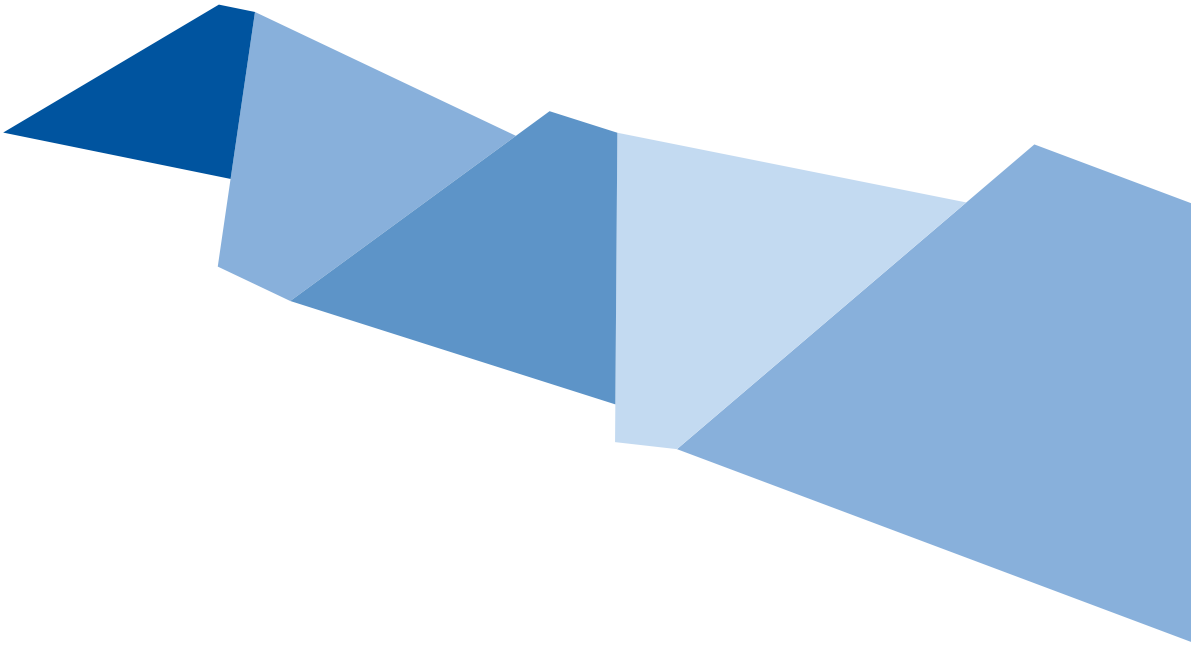


22. Kolloquium Programmiersprachen und Grundlagen der Programmierung

Thomas Noll
Ira Fesefeldt

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2023-02
RWTH Aachen University | Department of Computer Science | September 2023

Vorwort

Kasteel Bloemendal, Vaals (NL)

Das *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung* (KPS 2023) 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. Die erste Veranstaltung fand 1980 in Tannenfelde im Naturpark Aukrug in der Nähe von Neumünster in Schleswig-Holstein statt.

Die Kolloquien werden seitdem in etwa zweijährlichem Rhythmus organisiert. Aus den ursprünglich drei Arbeitsgruppen sind in der Zwischenzeit weitere Forschungsgruppen in ganz Deutschland und darüber hinaus 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 nunmehr fast 45-jährige Tradition dieser Treffen wird sichtbar in der Liste der bisherigen Tagungsorte und veranstaltenden Institutionen:

2021	Kiel	Uni Kiel
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	Fischbachbau	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 22. Kolloquium dieser Reihe fand mit 40 Teilnehmenden vom 25. bis zum 27. September 2023 im Kasteel Bloemendal im nierderländischen Vaals bei Aachen statt und wurde von der Arbeitsgruppe Softwaremodellierung und Verifikation der Fachgruppe Informatik der RWTH Aachen organisiert. Dieser Tagungsband enthält die wissenschaftlichen Beiträge, welche 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 sowie der Methodik der Programmierung abdecken.

September 2023

Thomas Noll
Ira Fesefeldt

Inhaltsverzeichnis

Deductive Verification of Probabilistic Programs*

Joost-Pieter Katoen[1]

Software Modeling and Verification Group, RWTH Aachen University, Germany
katoen@cs.rwth-aachen.de
https://moves.rwth-aachen.de

Abstract. Probabilistic programming is a fascinating new direction in programming. Meta, Google and Microsoft invest lots of research efforts in probabilistic programming. Nearly every programming language has a probabilistic version. Scala, JavaScript, Haskell, Prolog, C, Python, you name it, and even Excel has been extended with features for randomness. These languages aim to make probabilistic modelling and machine learning accessible to any programmer.

Probabilistic programs are sequential programs extended with random assignments and conditioning. Bayesian networks, a key model in decision-making under uncertainty, are simple instances of such programs. The increasing complexity of probabilistic graphical models such as Bayesian networks for real-life applications has been an important incentive for the development of probabilistic programming languages. Probabilistic programs are used to steer autonomous robots and self-driving cars, are key to describe security mechanisms, and naturally encode randomized algorithms. Their learning ability make them attractive for AI and probabilistic machine learning.

Probabilistic programs, though typically relatively small in size, are hard to grasp, let alone automatically checkable. Bugs can easily occur. The elementary question "does a program halt with probability one for a given input?" is as hard as the the universal halting problem. The question "is the expected number of steps until termination finite" is even more undecidable.

In this invited talk, we will present a deductive verification technique for imperative probabilistic programs. This technique uses weakest precondition reasoning and enables determining quantitative program properties such as the probability of divergence, bounds on the expected program outcomes, or the program's expected run-time. Like in classical program verification, one of the main challenges is to reason about loops. Invariants for probabilistic loops are typically lower and/or upper bounds of the true meaning of a loop. We will present some proof rules to determine such bounds and sketch some automated techniques to verify and synthesize loop invariants.

* Supported by ERC Advanced Grant 787914 FRAPPANT and DFG Research Training Group 2236 UnRAVeL.

R^{2}C: AOCR-Resilient Diversity with Reactive and Reflective Camouflage

Felix Berlakovich, Stefan Brunthaler

μCSRL – Munich Computer Systems Research Lab Research Institute CODE
University of the Bundeswehr Munich Neubiberg, Germany
felix.berlakovich@unibw.de,brunthaler@unibw.de

Abstract. Address-oblivious code reuse, AOCR for short, is a dangerous, yet unchallenged code reuse attack. By being able to bypass existing diversity techniques, AOCR poses a substantial security threat. AOCR's authors conclude that software diversity cannot mitigate AOCR, because it exposes fundamental limits to diversification.
Reactive and reflective camouflage, or R^{2}C for short, is a full-fledged LLVM-based defense that thwarts AOCR with software diversity and reactive capabilities through booby traps. In contrast to existing defenses, R^{2}C combines code randomization techniques with targeted data diversification. R^{2}C thus proves that AOCR poses no fundamental limits to software diversification, but merely indicates that code diversification *without* data diversification is a dead end. R^{2}C compiles complex real-world software, such as browsers, offers full support of C++, and different degrees of optimization with AVX2 instructions.
We evaluate the impact of our defense on performance, and find that on compute-intensive benchmarks R^{2}C shows an average performance overhead of 8.33% (geometric mean on SPEC CPU 2017). Our security evaluation indicates that attacker success at this performance impact is at most 9%, and conversely the probability of triggering a response is 91%. Given these odds, we argue that R^{2}C is effective at deterring and disincentivizing brute-force attacks.

Breaking Class Invariants Through Program Mutation-based Object State Space Exploration*

Jan H. Boockmann and Gerald Lüttgen

Software Technologies Research Group,
University of Bamberg, Germany
{firstname.lastname}@swt-bamberg.de

Comprehending a complex software component is challenging, but necessary for component reuse and maintenance. To make a component *reusable*, it usually suffices to document its external behavior and the constraints imposed on its method argument values. To make a component *maintainable*, however, information regarding its external behavior alone is insufficient, because maintenance requires the modification of the component's implementation. Although *class invariants* [5,6] that capture constraints on the component's program state are essential for maintainers, they are rarely documented. In addition, available tools [1,4,8,9,10,11] that learn assertions from program execution focus mainly on function contracts and only a few [2,7] support class invariants. One challenge is that a diverse set of training data is essential for assertion learning.

This talk presents a new dynamic approach for class invariant learning that addresses this challenge. In particular, we employ *program mutation* for the generation of invalid object states. Program mutation has already been used in [4,10,11] to assess the completeness of a learned assertion, but not to generate invalid objects as suitable training data for class invariant learning. We propose to leverage program mutations to artificially create objects through an object state space exploration that invokes unmutated and mutated constructors/methods. In contrast, some related work uses bounded-exhaustive enumeration [7] to create all objects within specified size bounds, which is only practical for small bounds and objects having few primitive attributes. Other related work uses state mutation [8] to create all object states that are slightly different from a given state. However, these mutations require information about the semantics of the type being mutated, which is usually only available for primitive types.

Our mutation-based approach uses behavioral oracles derived from (in)formal specifications to assess whether an artificially created object is invalid. For example, a specification may state that no method call should ever throw a null pointer exception. An artificially created object that throws such an exception is therefore invalid. We implement oracles as sound but incomplete *property-based tests* [3]: while an artificially created object that fails the test is guaranteed to be invalid, an object that passes the test is not necessarily valid.

Our preliminary evaluation examines how the choice of program mutation operators and the depth of exploration influence the diversity of invalid object states generated.

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

References

1. Astorga, A., Madhusudan, P., Saha, S., Wang, S., Xie, T.: Learning stateful preconditions modulo a test generator. In: Intl. Conf. on Programming Language Design and Implementation (PLDI). pp. 775–787. ACM (2019). https://doi.org/10.1145/3314221.3314641
2. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015
3. Fink, G., Bishop, M.: Property-based testing: A new approach to testing for assurance. ACM SIGSOFT Softw. Eng. Notes **22**(4), 74–80 (1997). https://doi.org/10.1145/263244.263267
4. Jahangirova, G., Clark, D., Harman, M., Tonella, P.: Oasis: Oracle assessment and improvement tool. In: Intl. Symp. on Software Testing and Analysis (ISSTA). pp. 368–371. ACM (2018). https://doi.org/10.1145/3213846.3229503
5. Meyer, B.: Eiffel: A language and environment for software engineering. J. Syst. Softw. **8**(3), 199–246 (1988). https://doi.org/10.1016/0164-1212(88)90022-2
6. Meyer, B.: Class invariants: concepts, problems, solutions. CoRR **abs/1608.07637** (2016)
7. Miltner, A., Padhi, S., Millstein, T.D., Walker, D.: Data-driven inference of representation invariants. In: Intl. Conf. on Programming Language Design and Implementation (PLDI). pp. 1–15. ACM (2020). https://doi.org/10.1145/3385412.3385967
8. Molina, F., Ponzio, P., Aguirre, N., Frias, M.F.: Evospex: An evolutionary algorithm for learning postconditions. In: Intl. Conf. on Software Engineering (ICSE). pp. 1223–1235. IEEE Computer Society (2021). https://doi.org/10.1109/ICSE43902.2021.00112
9. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: Intl. Conf. on Programming Language Design and Implementation (PLDI). pp. 42–56. ACM (2016). https://doi.org/10.1145/2908080.2908099
10. Pham, L.H., Thi, L.T., Sun, J.: Assertion generation through active learning. In: Intl. Conf. on Formal Engineering Methods (ICFEM). Lecture Notes in Computer Science, vol. 10610, pp. 174–191. Springer (2017). https://doi.org/10.1007/978-3-319-68690-5_11
11. Terragni, V., Jahangirova, G., Tonella, P., Pezzè, M.: Evolutionary improvement of assertion oracles. In: Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). pp. 1178–1189. ACM (2020). https://doi.org/10.1145/3368089.3409758

HOBBIT - Hashed Object Based Integrity

Matthias Bernad and Stefan Brunthaler

μCSRL – Munich Computer Systems Research Lab
Research Institute CODE
University of the Bundeswehr Munich
Neubiberg, Germany
{matthias.bernad,brunthaler}@unibw.de

Abstract. Still, in the year 2023, C and C++ belong to the most-used programming languages for developing software. The lack of memory safety, combined with a high adoption rate, makes programs written in C and C++ an attractive target for attackers. And so, even after decades of research, the arms race between attackers and defenders continues. An attack called Counterfeit Object-Oriented Programming (COOP) has proven to be particularly resistant to a range of code-reuse defenses. COOP exploits C++ semantics and, thus, is able to bypass defenses that do not consider C++.
We present Hashed Object Based Integrity, HOBBIT for short, a principled, compiler-based defense against COOP. HOBBIT prevents one of COOP's core primitives, the ability to manipulate existing, or inject new virtual table pointers. HOBBIT guarantees the integrity of virtual table pointers by computing signatures during object creation. To limit the attack surface of replay attacks, HOBBIT's signatures include run-time properties of the protected objects. HOBBIT instruments virtual functions with signature verification checks to detect modifications of virtual table pointers.
We implement a prototype based on clang, a C++ frontend for LLVM. We benchmarked our prototype with the SPEC CPU 2017 benchmark suite and measured a geometric mean run-time overhead between $\approx$ 14.88% and $\approx$ 106.12% over all C++ benchmarks, depending on the chosen hashing strategy. To optimize the run-time overhead, we have incorporated a novel static-analysis that detects vitally important COOP-gadgets. Analogous to profile-guided optimization, our analysis gives rise to *gadget-guided defense application*. Initial results confirm substantial optimization potential by a factor of up to 50×.

A Domain-Specific Language for Building Modular Interpreters

Niels Bunkenburg

Kiel University, Kiel, Germany
nbu@informatik.uni-kiel.de

Abstract. Interpreters are a common way to give semantics to a programming language. They are easier to implement compared to compilers and are especially useful for prototyping. At its core – omitting lexing and parsing – an interpreter is a function that takes the abstract syntax tree of a program and returns a value that represents the result of the program's execution. At some point, the interpretation function needs to decide how to interpret each individual syntactic construct of the input language. For example, if the language allows local bindings, the interpretation function needs an environment for storing the corresponding values of variables occurring in the program. Since many languages allow local bindings, this functionality needs to be implemented in almost every interpreter. We present a domain-specific language based on algebraic effects and effect handlers that provides a convenient way to build interpreters from reusable components. We focus on the implementation of an interpreter for the functional-logic programming language Curry. Thus, we not only show how to implement common concepts like literals, function declarations or pattern matching but also consider how to implement non-standard features like Curry's non-determinism, lazy evaluation and free variables. Finally, we evaluate the benefits of our approach with respect to code size, modularity and extensibility as well as the impact on performance. Finally, we discuss how the DSL can be used to implement interpreters for other languages.

Optional Type Systems: Current Approaches and Ongoing Efforts*

Werner Dietl
University of Waterloo
wdietl@uwaterloo.ca
https://ece.uwaterloo.ca/~wdietl/

Abstract. The Java type system gives useful guarantees about software, but there are many properties that cannot be expressed. One pervasive property that cannot be expressed is whether a reference can be null or not. The resulting null pointer exceptions are the bane of programmers and have been called the "billion dollar mistake". They happen even if you think hard about your code and test it thoroughly. There are many causes for null pointer exceptions, including object initialization, missing map keys, confusing contracts, and missing checks.
Optional type systems allow developers to improve the quality of their software by encoding additional properties as type systems and enforcing these properties at compile time. We will discuss a type system that prevents null pointer exceptions at compile time[1] and the general framework it builds upon[1]. These optional type systems have found hundreds of bugs in millions of lines of well-tested code. We will discuss alternative tools and interoperability, in particular with Kotlin, and the JSpecify effort[2] to standardize static analysis annotations across the Java ecosystem. Finally, we will briefly discuss whole-program type inference[2,4] and the interaction of optional type systems with deductive verification[3].

References

1. W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and Using Pluggable Type-Checkers. In *Software Engineering in Practice Track, ICSE*, 2011.
2. W. Dietl, M. D. Ernst, and P. Müller. Tunable Static Inference for Generic Universe Types. In *ECOOP*, 2011.
3. F. Lanzinger, A. Weigl, M. Ulbrich, and W. Dietl. Scalability and Precision by Combining Expressive TypeSystems and Deductive Verification. *OOPSLA*, 2021.
4. T. Xiang, J.Y. Luo, and W. Dietl. Precise Inference of Expressive Units of Measurement Types. *OOPSLA*, 2020.

* We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants program, RGPIN-2020-05502, and an Early Researcher Award from the Government of Ontario. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSERC or the Governments of Ontario or Canada.

[1] https://eisop.github.io/
[2] http://jspecify.org/

DEPS: Leveraging Hardware Faults for Binding Software to Hardware*

Ruben Mechelinck[1], Daniel Dorfmeister[2], Bernhard Fischer[2], Stijn Volckaert[1], and Stefan Brunthaler[3]

[1] imec-DistriNet, KU Leuven, Leuven, Belgium
{ruben.mechelinck,stijn.volckaert}@kuleuven.be
[2] Software Competence Center Hagenberg GmbH (SCCH), Hagenberg, Austria
{daniel.dorfmeister,bernhard.fischer}@scch.at
[3] μCSRL, CODE Research Institute, Bundeswehr University Munich, Neubiberg, Germany
brunthaler@unibw.de

Abstract. Industrial-scale reverse engineering affects the majority of companies in the mechanical and plant engineering sector and imposes significant economic damages. Although reverse engineering mitigations exist, economic damage has not been impacted, indicating that they have failed to address the problem. A closer investigation shows that industrial-scale reverse engineering typically only expends efforts on replicating hardware, since software can often be copied verbatim—no reverse engineering effort required.

We present DEPS, a system that binds software to hardware through physically unclonable functions and software diversity. DEPS transforms programs such that they only exhibit their intended behavior on the single machine they are bound to at compile time. When run on any other machine, the programs will exhibit a different functionality. DEPS relies on unclonable machine features and thereby forces counterfeiters to reverse-engineer not just clone the hardware, but to also clone software. Cloning both hard- and software drives up reverse engineering costs, thereby, also decreases the economic viability of industrial-scale reverse engineering.

DEPS works on commodity hardware and does not rely on expensive hardware components. Our evaluation shows that DEPS is effective and it incurs less than 5 % run-time performance overhead in a practical case.

* The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Labour and Economy (BMAW), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) (FFG grant no. 888338) and the SCCH competence center INTEGRATE (FFG grant no. 892418) within the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

Fix Spectre in Hardware! Why and How

M. Anton Ertl[1]

TU Wien

Abstract. Spectre can be fixed in hardware by treating speculative microarchitectural state in the same way as speculative architectural state: On mis-speculation throw away all the speculatively-performed changes. The resource-contention side channel needs to be closed, too. This position paper also explains how Spectre works, why software mitigations are not sufficient.

1 Introduction

Spectre [SSLG18] is a hardware vulnerability that has been reported to hardware manufacturers such as AMD and Intel in June 2017, and to the general public on January 3, 2018. Unlike Meltdown, which has been published at the same time and has been fixed in hardware relatively quickly[1] (or, in the case of AMD, not built into the hardware from the start), even the latest CPUs with speculative execution from all manufacturers are vulnerable to Spectre, and hardware manufacturers leave it to software to "mitigate" these vulnerabilities.

New Spectre variations that bypass existing mitigations are discovered regularly, e.g., the recent discoveries of Intel's DownFall [Mog23] and AMD's Inception [TWR23] vulnerabilities. Intel lists[2] 6 "transient execution attacks" published in 2018–2021, and, as of this writing (August 2023), 5 published in 2022-2023 that require software mitigations (sometimes with hardware support) even on Intel's most recent Sapphire Rapids server CPU. This includes the original two Spectre variants (v1 and v2) reported to Intel in June 2017.

In this position paper I present a general approach to fix Spectre in hardware (Section 9) that would fix not only Spectre v1 and v2, but also, e.g., the recently-discovered Downfall and Inception vulnerabilities. In order to make this work understandable to a wide audience, Section 2 explains architecture and microarchitecture, Section 3 side channels, Section 4 speculative execution, Section 5 Spectre and Section 6 its relevance. Section 7 argues why we should not seek the solution to the problem in software mitigations. One possible hardware fix for Spectre is to eliminate speculative execution, but the performance impact is unacceptably big (Section 8). Instead, a better fix is to eliminate the side

[1] https://www.anandtech.com/show/13450/intels-new-core-and-xeon-w-processors-fixes-for-spectre-meltdown

[2] https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html

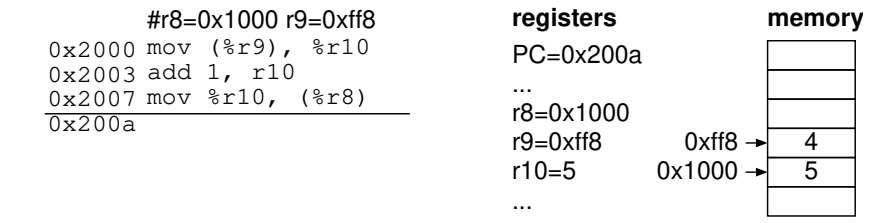


Fig. 1. Architectural state: register and memory contents; this example shows the architectural state right after the instrucion at `0x2007`

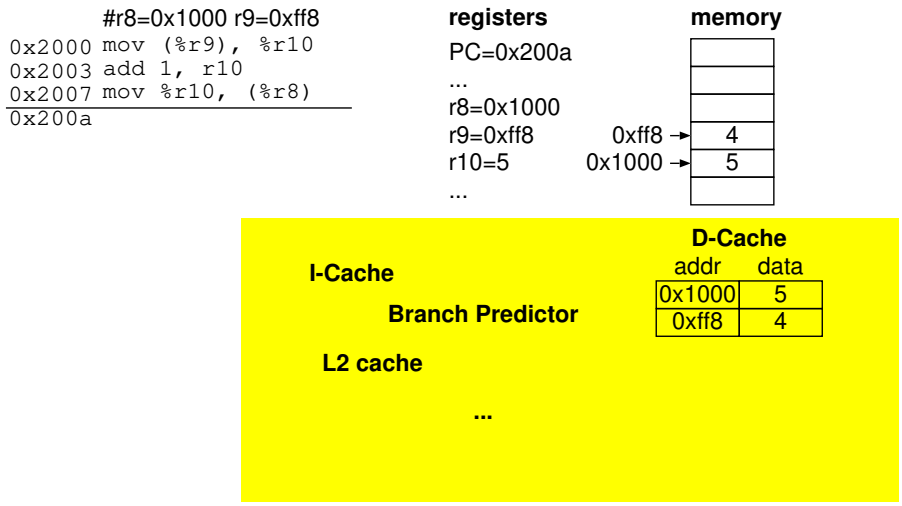


Fig. 2. Microarchitecture (yellow background) is normally invisible to software, apart from its performance effects

channels back from the speculative world into the committed world (Section 9). Section 10 discusses the costs of this fix. Finally, Section 11 is a call to action for computer buyers, researchers and CPU manufacturers.

2 What is architecture and microarchitecture?

The architecture (aka instruction-set architecture, ISA) is the interface between the hardware and the software. Software sees main memory and registers, and instructions that work on them (see Fig. 1).

On the hardware side of this interface the highest design level is called microarchitecture. Microarchitecture is generally not visible in the functionality presented to the software, only through the performance. I.e., instructions gen-

erally deal only with architectural features such as memory and registers, not with microarchitectural features such as caches.[3]

E.g., the cache is a microarchitectural feature, and the CPU works functionally in the same way with the cache as without it (or with caches with different sizes); the only difference is that CPUs with caches run faster. While an access to main memory takes several hundred cycles on a modern general-purpose CPU, accessing the level-1 (L1) data (D) cache costs 3–5 cycles. However, the (L1) D-cache is much smaller (32-128KB on recent CPU cores), and contains only recently-accessed data.

Speculative execution is another microarchitectural feature and is discussed in Section 4.

3 What are side channels?

A side channel (aka covert channel) reveals data not directly by letting attacker read the secret data, but through ancillary properties of data processing.

E.g., if the run-time a program takes depends on the secret, an attacker can often use this fact to extract the secret (this kind of attack is known as a timing attack). E.g., a program could contain an `if`-statement where the condition depends on the secret, and the run-time of the two branches differs. For program code that deals with the dearest secrets (encryption keys and passwords), avoiding secret-dependent branches has long been best practice.

More generally, the best practice has been to write code that runs in constant time with respect to the secret.

The timing of memory accesses depends on the input address, thanks to caches. Caches provide such a big performance boost that we prefer to keep them and deal with the security implications in some other way rather than use CPUs without caches.

One case where memory access timing has played a role is AES encryption. It has been designed in a way that is hard to implement without loads from an address that depends on the secret key. While that dependence is quite convoluted, Bernstein has found a way to use the timing variation due to loads in such AES implementations to extract the key [Ber05].

3.1 Defending against side-channel attacks

The defense against side-channel attacks first requires realizing that there is a side-channel, and then taking measures that eliminate the leakage of secret information through that side channel.

[3] There are a few cases where microarchitectural features are managed by software, and there are instructions for that, e.g., instructions for fetching data into caches (prefetch), instruction-cache management, or for draining the pipeline to ensure strictly in-order execution, but these instructions are not relevant for the rest of this paper.

As mentioned above, for timing side channels this has usually been done by writing the pieces of code that deal with the dearest secrets as constant-time code. These pieces of code tend to be miniscule (hundreds of lines) compared to the huge amounts of code (millions of lines) for complete programs like a web browser or an operating system.

While this makes the defense sound like being the responsibility of the software people alone (and this perception may have contributed to the lack of efforts on hardware fixes for Spectre), the software people cannot do it without support from hardware manufacturers.

In order to write constant-time code, the programmer needs guarantees that the timing of the used instructions does not depend on the input. Such guarantees have been historically hard to come by (and were only specified for specific implementations rather than the architecture), but recently Intel has guaranteed the input-independence of a subset of instructions for all of its implementations.[4]

In the AES case, the hardware manufacturers helped, not by making load timing address-independent (which would be impractical, as mentioned above), but by providing instructions that perform the problematic steps of AES in constant time without needing loads.

The discipline of writing constant-time code is used only for cryptographic and password-handling code, because it requires additional effort and specialized competencies, because it often results in slower run-time, but also because it is too limiting and impractical for implementing the requirements of most code. E.g., while the contents of spreadsheets of big companies and intelligence agencies may be considered by their users to be at least as secret as the encryption keys of ordinary citizens, to my knowledge nobody has tried to write a spreadsheet program with content-independent timing.

4 What is speculative execution?

Most modern general-purpose CPU core use speculative execution, a microarchitectural technique that works as follows:

The core's branch predictor predicts a likely future execution path and then executes (but does not commit) instructions on that path. The catch is that the prediction may turn out to be incorrect. In that case the architectural state (registers and memory) must not be changed in the way indicated by the misprediction prediction. If the speculation turns out to be correct, the changes can be committed (see Fig. 3).

Modern CPUs with speculative execution do this by conceptually dividing the core into a speculative part, which contains architectural results-to-be of unconfirmed speculative execution, and a committed part which contains the actual architectural state at the current architectural program counter (PC). So when the core architecturally processes an instruction (by committing (aka retiring)

[4] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html

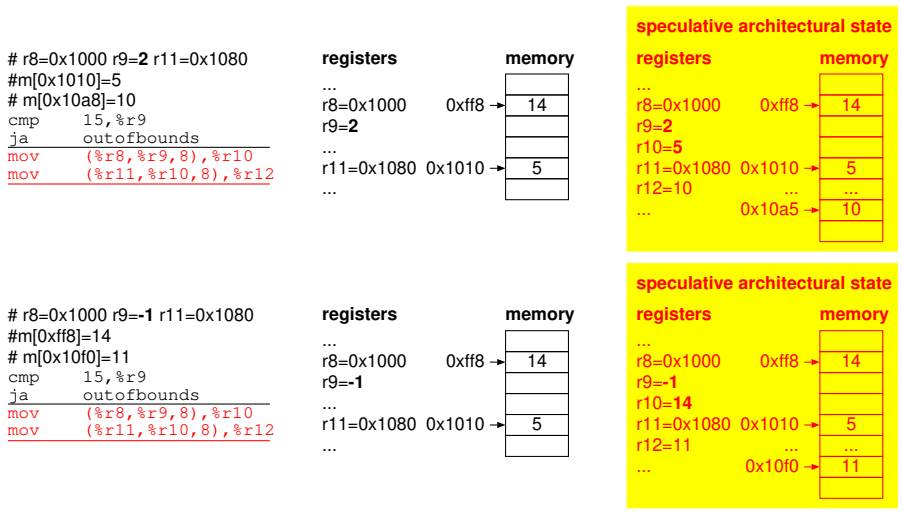


Fig. 3. Two examples of speculative execution, in both cases the `ja` instruction is predicted as being not taken. Above: The prediction is correct, and the speculative architectural state is eventually committed. Below: The prediction is incorrect, and the speculative architectural state is squashed.

it in the reorder buffer), that instruction has often been speculatively executed some time earlier, and its result is lying around, waiting to be committed; and the commit takes this result and turns it into committed architectural state.

However, when a branch turns out to be mispredicted, and this branch is committed, all the speculative results after the branch (i.e., on the wrong path) are thrown away, and the processor starts executing on the correct path.

Note that this speculative execution not only contains register updates, but also stores to memory, possibly including several stores to the same memory location, and (speculative) loads from that location in between.

There have been many speculative-execution implementations of various architectures since the 1990s, and almost[5] all of them implemented the handling of architectural state correctly, both for correctly predicted branches and for mispredictions, for various kinds of registers, and for memory.

5 What is Spectre?

For microarchitectural state, e.g., the contents of the cache, existing processor cores do not discern between speculative and committed changes. After all, the

[5] The recently-published Zenbleed bug in AMD's Zen2 core (https://lock.cmpxchg8b.com/zenbleed.html) is the exception that proves the rule.

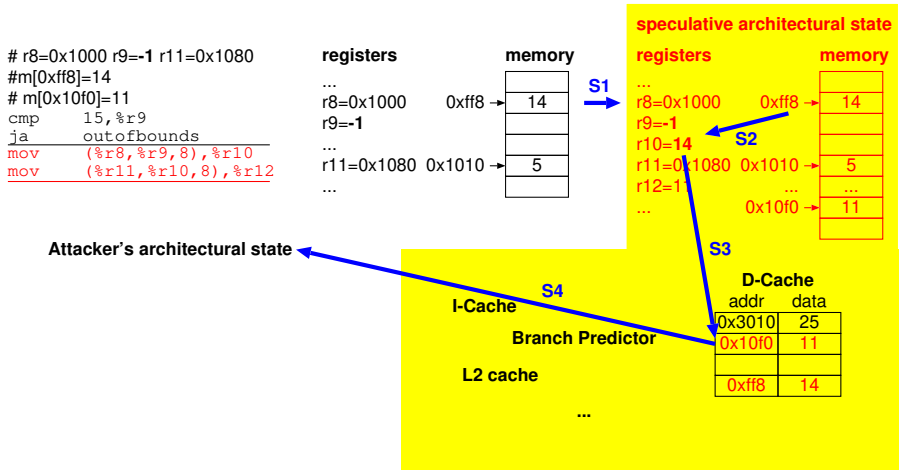


Fig. 4. A Spectre v1 attack starts with a misprediction (S1), loads the secret (in `0xff8`) into speculative architectural state (S2), changes the cache state in a secret-dependent way (S3), and finally uses cache timing to extract the secret into the architectural state of the attacker (S4).

idea is that microarchitectural state is invisible anyway. If a speculative load puts a line in the D-cache (and evicts another line), this has no architectural significance, so the hardware designers have had no qualms at performing this change speculatively, without a mechanism that cancels it in case of a misprediction.

Unfortunately, this approach opens a side channel that allows to leak data from the otherwise ephemeral world of misspeculation.

Figure 4 shows an example: The `cmp` and `ja` instructions architecturally prevent an out-of-bounds access to the array in `r8`, but if the branch is mispredicted to be not-taken, the following code is speculatively executed, and it reads the address `0xff8` speculatively; by using any other index, any other 64-bit value in the address space of the process could be accessed, including, e.g., secret keys or passwords that are there for cryptographic or authentication purposes. Let us assume that the secret is in memory location `0xff8`. In itself the load of the secret value into the speculative r10 does not appear a problem, because this is still the ephemeral world of misspeculation, and it cannot get out, right?

Unfortunately, on current it can get out hardware through a cache-based side channel: The second `mov` instruction loads a value into the D-cache, and the address of this load depends on the secret. The loaded cache line replaces a line that used to be in the cache, and which cache line is replaced depends on the address. An attacker can repeatedly access a number of memory locations in order to prime the cache, and can see from the timing of the cache accesses whether a cache line has been replaced, and in this way learn something about the secret.

There are a number of steps involved in Spectre attacks:

S1 The speculation itself: In this example (which is a Spectre v1 attack) a conditional branch causes speculative execution, but there are others. E.g., Spectre V2 uses indirect branches. There are also other speculative mechanisms in modern cores, such as speculating whether a memory load is to a different or the same address as an earlier store, and this has also been used in a number of attacks.

S2 The mechanism for getting the secret data into speculative architectural state. In the example above it is the load from `a[i]`. In the recently-published Downfall vulnerability [Mog23], it's gather instructions as implemented on some Intel microarchitectures.

S3 The sending side of the side channel for getting the data out of the misspeculation realm. In the example above it's the access to `b[j]` that channels information about `j` through the cache side channel. But other microarchitectural state can also be used, such as the power state of the AVX unit [SSLG18].

S4 The receiving side of that side channel. For a cache side-channel this consists of the attacker priming the cache and monitoring through timing which lines are replaced by the victim.

There is a lot of variation on all of these steps, leading to the stream of vulnerabilities that have been found up to now and continue to be found. For more details (and more aspects) there is a survey of the Spectre and Meltdown attacks until December 2020 [XS21]. A term that has been used to cover all these vulnerabilities and attacks against them is "transient execution vulnerabilities/attacks", but in this paper I just use "Spectre" in the same meaning as referring to all of these speculation-based side-channel attacks.

6 How relevant is Spectre?

Has Spectre been used in the wild? It's hard to know. Consider the case where attackers use Spectre to determine your password or encryption key. If they use that to decrypt your files, you may never know; maybe it was bad luck that your competitor undercut you by a narrow margin. But even if somebody does something very blatant like publish your documents on the Internet or encrypt your files and demand ransom, you usually don't know how the attacker got at your password or your encryption key.

However, a working exploit for reading normally unaccessible files on Linux has been discovered by Julien Voisin.[6] There is no proof that this exploit has been used for an actual attack, but given that it is widely available, it would be surprising if it has not.

Some people argue that Spectre is not relevant because there are many software vulnerabilities that may be used for subverting your system; so why, they

[6] https://dustri.org/b/spectre-exploits-in-the-wild.html

argue, should an attacker bother with Spectre, which is supposedly harder to use. On the other hand, software vulnerabilities may be discovered and fixed at any moment, while Spectre exists unfixed on all desktop and server hardware, and is not even mitigated against in most software. So Spectre may be more attractive to attackers than some people give it credit for.

7 Why is software mitigation not a good way to deal with Spectre?

The mitigation of non-speculative timing attacks is to write the few hundred lines of code that deals with keys and passwords in a constant-time way. Can we not just deal with Spectre in the same way?

Unfortunately, for Spectre *all* the software that has the secret in its address space can potentially be used for an attack, and consequently would have to be hardened. For a web browser or an OS kernel that is typically millions of lines of code.

As an example of a mitigation, for the Sprectre V1 example in Fig. 4, speculative load hardening has been proposed. A simple variant would change the code as follows:

```
     cmp    15,%r9
     ja     end
     mov    $0x0,%rax
     cmova  %rax, %r9
     mov    (%r8,%r9,8),%r10
     mov    (%r11,%r10,8),%r12
end:
```

Here the `cmova` hardens the following load, by setting `r9` to 0 if `r9>15`. While this condition cannot architecturally be true at that place, it can be true during misspeculation. The `cmova` instruction uses the same flags as the `ja` branch, but the mitigation assumes that `cmova` does not speculate.

In reality speculative load hardening is substantially more complicated, because it also has to also deal with possible speculation on earlier branches [ZBC+23].

Software mitigations have apparently led to the impression that Spectre is under control and no hardware fix is necessary, but they have a number of problems:

7.1 Still vulnerable

It has often turned out that many mitigations do not even completely close the vulnerability for which they are designed.

One reason for that is that the mitigation defends against a too-narrow attack scenario. E.g., speculative load hardening (SLH) has been implemented in the

LLVM compiler and is intended to close the Spectre V1 vulnerability presented above, but it still has some leakage; this was then improved in Strong SLH [GP19] and recently Ultimate SLH [ZBC+23].

Another reason is that the mitigation relies on assumptions about microarchitectural mechanisms that turn out to be wrong; e.g., earlier Spectre V2 mitigations assumed that returns would only be predicted from the return stack, but there are some microarchitectures that use the general indirect-branch predictor to predict returns when the return stack is empty (so returns can also be used in Spectre V2 attacks).

Also, these mitigations tend to work only against a specific variant, but new variants are discovered all the time.

7.2 Performance

These mitigations cost performance, for the whole program (because with Spectre the whole program can be used to reveal the secret). E.g., Zhang et al. report a factor of around 2.5 slowdown (compared to no mitigation) for SPEC CPU 2017 (int and fp, rate and speed) [ZBC+23]. I saw a slowdown (compared to using no mitigation) by a factor 2–9.5 from compiling Gforth with the fastest retpoline mitigation against Spectre V2[7].

7.3 Effort

Because the slowdowns that you get from applying compiler-based mitigations across the board are often considered to be unacceptable, there is the idea that programmers should be more selective and analyse whether each specific place in a program can actually be used by an attacker, and only harden those places, lowering the performance cost.

However, this requires a huge amount of effort, and it takes only one place in the potential attack surface that the programmer mistakenly has not hardened, and the program is still vulnerable.

And when the next vulnerability and mitigation shows up, you have to do it all again. And when the program is changed (due to, e.g., new requirements), you have to analyse more than just the changed lines.[8]

8 Why is the first idea for a fix not so great?

The first idea many people have for fixing Spectre is to eliminate speculative execution. While this certainly fixes Spectre by preventing step 1 of the exploits, the performance impact of this measure is pretty bad: E.g., the core without speculation that shows the best performance on our LaTeXbenchmark[9] is the

[7] news:<2023Jan15.105348@mips.complang.tuwien.ac.at>

[8] The idea that you do not have to reanalyse code when the requirements change resulted in the Ariane 501 explosion.

[9] https://www.complang.tuwien.ac.at/franz/latex-bench

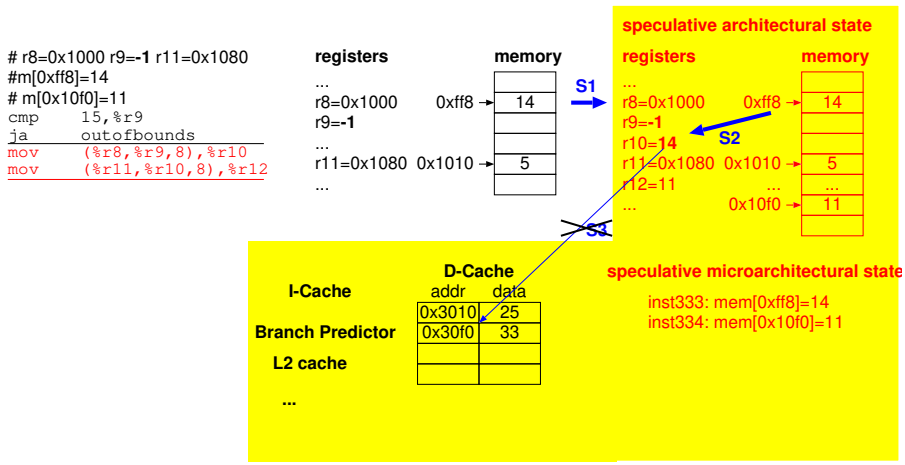


Fig. 5. Separating speculative microarchitectural state from committed microarchitectural state eliminates the S3 part (and therefore also S4) of a Spectre attack, as far as microarchitectural state is concerned; resource-limitation side channels also need to be addressed, see text.

1800MHz Cortex-A55 on the Rock 5B single-board computer. The Cortex-A76 core (with speculative execution) running at 2275MHz on the same computer is 3.3× as fast for this benchmark, and the 3000MHz Apple Firestorm is 7.8× as fast.

Given the performance impact, it's no surprise that we have not seen a resurgence of microarchitectures without speculation. The number of customers that would exchange this much performance for security against Spectre is small. The customers' reasoning is as follows: There are lots of vulnerabilities in the software we use, so fixing Spectre is not going to make our computers that much safer. Therefore we are not willing to sacrifice that much performance for this benefit.

9 How to fix Spectre in hardware?

The less costly and therefore better way to fix Spectre is to prevent step S3.

9.1 Side channels based on microarchitectural state

In particular, for the side channels through microarchitectural state, we can use the same approach for microarchitectural state as for architectural state: keep the speculative state separate from the committed state, and squash it when it turns out that the speculation is wrong. This goes for all microarchitectural state: D-cache, I-cache, branch predictor, TLBs, etc.

In the case of D-cache, several papers [YCS+18,KKS+19,AJ20] have already proposed ways of doing that, probably because the cache side channel has been the most popular one for Spectre-type attacks. But of course, the other state-based side channels need to be closed, too.

Some attempts at fixes for state-based side channels have tried to undo the changes when a speculation turns out to be false, e.g., CleanupSpec [SQ19]. However, I think that it is better to keep the speculative changes separate until commit time, for the following reasons:

- The microarchitectural state is changed, albeit only for a short time, and this is visible to potential attackers, given enough effort.
- It is harder to reason about the correctness of an undoing approach than about an approach that never speculatively changes the non-speculative state.
- Undoing approaches have been tried for architectural state [DA92], but committing approaches have won. It is likely that the same reasons will make undoing of microarchitectural state unattractive.

9.2 Side channels based on resource contention

Apart from the popular state-based side channels, another kind of side channel is contention for resources such as execution ports, functional units, or cache ports. SMoTherSpectre [BSN+19] attacks another SMT thread on the same core by using execution port contention by the speculatively executing victim as a side channel. Even worse, speculative interference attacks [BSP+21] use resource contention to affect the timing of an older (eventually committing) instruction in the same thread.

For the SMT side channel, a solution is to have a fixed partitioning of resources between the threads, so that no thread can use resource contention as a side channel. This means that resources that exist only once have to be time-shared. E.g., if there is an non-pipelined divider that takes 20 cycles for the division, there are fixed 20-cycle time slots for each thread, and when a thread does not have a division ready at the start of its time slot, that time slot goes unused. This fixed partitioning will cost some performance; it could be made optional, allowing the full benefit of SMT to be used in settings where the sibling threads are believed to not spy on each other.

For the same-thread problem, Behnia et al. [BSP+21] describe the high-level principle: "a speculative instruction must not influence the execution of a non-speculative instruction". And they describe two rules that ensure that:

- "No instruction ever influences the execution time of an older instruction." They propose to achieve this by giving priority to older instructions in case of resource contention. They discuss several options how to deal with non-pipelined execution units. The slot idea above is another way to deal with that: If a thread can start using the execution unit only at the start of a slot, the priority approach works for non-pipelined units (although one might wish for better performance).

– "Any resources allocated to an instruction at the front end and the execution engine are not deallocated until the instruction becomes non-speculative". This rule ensures that misspeculated code cannot produce timing variations by congesting the front end.

9.3 Other side channels

Another known side channel is energy consumption. In particular, Meltdown-Power [KJG+23] uses speculation for S1 and S2, and then a power-based side channel for S3 and S4. However, it requires that the speculative load updates the cache, which does not happen with the fix for speculative microarchitectural state outlined above, so fixed hardware would be immune against this particular attack.

Still, one can imagine that the energy consumption of e.g., functional units working on mis-speculatively loaded data could reveal something about the data. At the moment I have no good hardware answer for that. On the other hand, the question is if such an attacks can be made practical (i.e., leak relevant amounts of data in realistic time frames).

10 How much does the fix cost?

The fixes certainly cost design complexity. Hardware architects have been remarkably good at handling the increasing complexity of modern high-performance CPUs, and I expect them to rise to the challenge of designing fixed hardware, if they are given the task.

The resulting CPU cores will require more area, for the speculative state. E.g., if we want to be able to buffer, say, 30 cache lines loaded from outer cache levels in speculative microarchitectural state, the memory for these 30 cache lines is needed, as well as the infrastructure to look up data in them and deal with snoop messages. Compared to the 224 physical ZMM registers (each with 64 bytes) in Intel's Sunny Cove core, this does not seem to add that much area; and I expect that the area for other microarchitectural features will be even smaller.

Concerning performance, the additional buffers can even help, and for MuonTrap [AJ20] the Parsec benchmarks indeed see a speedup by a factor 1.05. However, SPEC 2006 sees a slowdown by a factor 1.04 compared to vulnerable hardware. And then there is the question of how much speed the additional area could have produced if it was invested just in performance. On the other hand, compared to applying software mitigations to all software (e.g., a factor 2.5 for defending only against Spectre v1), even the SPEC slowdown and the opportunity performance cost of the additional area are small.

One may want to compare with the more selective hardening approach that is used in, e.g., the Linux kernel. This kind of hardening has not been applied to the SPEC benchmarks, and the hardware fixes have not been measured on the benchmarks that are typically used for measuring the Linux kernel performance,

so a direct comparison is not possible. Looking at Michael Larabel's results for how the kernel mitigation of just Inception[10] and the firmware mitigation of just Downfall [11] slows down applications, the slowdowns are often larger than what has been reported as slowdown from hardware fixes for the cache side channel. While these are numbers for different programs and mitigations/fixes for different vulnerabilities, and both comprehensive software mitigations and comprehensive hardware fixes will have higher cost, I expect that the majority of the performance cost of a hardware fix is in dealing with the cache (because of stuff like cache coherence), so I don't expect the cost of a comprehensive hardware fix to be that much higher than the cache-only approaches we have seen yet, while on the software mitigation side, every vulnerability seems to require its own mitigation, with a program-dependent performance impact, sometimes very expensive, as discussed above.

11 What should I do?

As computer **customers**, we should keep asking the CPU manufacturers when they will finally fix Spectre in hardware; we should tell them that software mitigations are not good enough.

And when one of the manufacturers comes out with a CPU with a Spectre fix, we should prefer these CPUs in our buying decisions even if they are a little slower at running unmitigated software (or software with mitigations that are unnecessary for the fixed CPUs). After all, such a CPU will be safer than an unfixed CPU when both run unmitigated software (the usual case). And such a CPU will be faster and at least as safe (probably safer) when the fixed CPU runs software without mitigations and the unfixed hardware runs software with mitigations.

When CPU manufacturers claim that they have fixed Spectre, only believe them when they explain how they did it (and only if that explanation does not have holes); don't accept hand-waving along the lines of "Differences in AMD architecture mean there is a near zero risk of exploitation"[12].

As **computer architecture researcher**, you can work at designing and evaluating mechanisms for fixing Spectre. Even if there is already some work in that direction, there is probably still some microarchitectural state or other side channels that have not been covered yet. And even for the microarchitectural state that has been covered, there are probably ways to improve on it, i.e., a solution that costs less area and/or less performance.

If your research leans more towards **theory**, you could work out a formal description of speculative side channels, and a way how computer architects could prove that they have closed these side channels. I do not know if they worked out such an approach to make sure that speculation works correctly for

[10] https://www.phoronix.com/review/amd-inception-benchmarks
[11] https://www.phoronix.com/review/intel-downfall-benchmarks
[12] https://web.archive.org/web/20180104014617/https://www.amd.com/en/corporate/speculative-execution

architectural state; it may be (usually) good enough to validate the architectural design by running test programs, but for microarchitectural state and other side channels, such an approach is needed, because the side channel does not show up in the usual architectural validation.

If you work at a **CPU manufacturer** (or CPU design house), you have the best opportunity to fix this problem. If the decision is up to you, go ahead and decide that you will make a Spectre-immune high-performance CPU core. If the decision is up to someone else, make a case that convinces them that the fix is worth the development and manufacturing costs by making your CPU safer than the competition, and to put a stop to the constant stream of new Spectre- and Meltdown-type vulnerabilities (and the slowdowns from firmware and software mitigations). Also, imagine what happens if your competition is first at presenting a Spectre-immune CPU.

12 Conclusion

Attacks like Spectre that extract speculative state through a side channel are different from earlier side-channel attacks in being impractical to mitigate in software: not just the small piece of code that deals with the secret, but all software in the same address space as the secret (including libraries) needs to mitigate these attacks; E.g., an automatic compiler approach against Spectre v1 alone costs a factor 2.5 in performance, and that does not defend against all Spectre attacks (e.g., not against Spectre v2). One way to reduce this cost taken in, e.g., the Linux kernel, is to try to identify places that can be attacked and only harden those; this costs a lot of programmer effort, has the potential danger of leaving a hole open, and when another attack is discovered, this effort often has to be repeated.

Therefore the right way to deal with Spectre is to fix it in hardware. For speculative microarchitectural state, it should be treated just like speculative architectural state: During speculation, keep it separate from the committed state; and when the speculation turns out to be wrong, just squash the speculative state (including speculative microarchitectural state). When the speculation is correct, turn the speculative state into commited state (e.g., during instruction commit).

In addition to state-based side channels, resource contention can also provide a side channel. This can be addressed with a fixed partitioning of resources in an SMT setting, by always prioritizing older instructions in resource conflicts, and by managing front-end resources in a specific way.

A hardware fix for Spectre costs some chip area and often also performance compared to a vulnerable core, but much less than applying a software mitigation against just Spectre v1 across the board.

References

[AJ20] Sam Ainsworth and Timothy M. Jones. MuonTrap: Preventing cross-domain Spectre-like attacks by capturing speculative state. In *International Sympo-*

sium on Computer Architecture (ISCA), pages 132–144, 2020.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. 2005.

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *Conference on Computer and Communications Security*, 2019.

[BSP+21] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christo pher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, pages 1046–1060, 2021.

[DA92] Keith Diefendorff and Michael Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, pages 40–63, April 1992.

[GP19] Marco Guarnieri and Marco Patrignani. Exorcising Spectres with secure compilers. *CoRR*, abs/1910.08607, 2019.

[KJG+23] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking inaccessible data with software-based power side channels. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7285–7302, Anaheim, CA, August 2023. USENIX Association.

[KKS+19] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *Design Automation Conference*, 2019.

[Mog23] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*, 2023.

[SQ19] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An undo approach to safe speculation. In *International Symposium on Microarchitecture*, page 73–86, 2019.

[SSLG18] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. *CoRR*, abs/1807.10535, 2018.

[TWR23] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7303–7320, Anaheim, CA, August 2023. USENIX Association.

[XS21] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys*, 54(3), May 2021.

[YCS+18] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *International Symposium on Microarchitecture*, page 428–441, 2018.

[ZBC+23] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7125–7142, Anaheim, CA, August 2023. USENIX Association.

Pushing the Limits of Template Metaprogramming with Code Generation

Lukas Gail

Technische Hochschule Mittelhessen, Wiesenstr. 14, 35390, Giessen, Germany
lukas.gail@mni.thm.de.de

Abstract. Template Metaprogramming is a niche and orignally unintended aspect of C++ that allows arbitrary (turing-complete) computations at compile time purely within the type system [2]. Its computational power is mostly based on template specializations and recursive template instantiations. While some of its applications have been made obsolete by the introduction of constexpr-functions, template metaprogramming still sees use in the domain of generic programming in both standard and user provided libraries.
An unfortunate side effect of template metaprogramming being introduced by accident is a substantial lack of usability features. Its verbose syntax makes template metaprograms inaccessible as well as hard to read, write and maintain. There have been various attempts to fix this situation, ranging from C++-libraries to code generators [1].
Due to the absence of stateful computations in the type system, template metaprogramming is frequently considered to be a purely functional programming system. This motivates the idea of having programs in a functional programming language translated into template metaprograms to provide programmers with a more concise language to work with that is both easier develop and maintain programs in.
The presented work explores this idea further and provides useful patterns to translate a higher level functional programming language into template metaprograms.
This includes a) a generalized, structure-preserving pattern to translate expressions into so called expression classes, combined with code reduction techniques that reduce the structural overhead introduced by the generalized translation, b) working around C++'s strict define before use principle to allow mutual recursive template metafunctions both locally and across multiple files/modules, and c) a technique to support a first-match policy rather than a most-specific-match policy for pattern matching.

References

1. Šefl, V.: Translating Lambda Calculus Into C++ Templates. In: Zsók, V., Hughes, J. (eds.) Trends in Functional Programming. pp. 95–115. Springer International Publishing, Cham (2021)
2. Veldhuizen, T.L.: C++ Templates are Turing Complete (2003)

Fault-tolerance with the TeamPlay Coordination Language

Wouter Loeve[1] and Clemens Grelck[1,2]

[1] University of Amsterdam, Amsterdam, Netherlands
{w.loeve,c.grelck}@uva.nl
[2] Friedrich Schiller University Jena, Jena, Germany
clemens.grelck@uni-jena.de

Abstract. Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches. The functional coordination language TeamPlay follows the approach of exogenous coordination and organises an application as a streaming data-flow graph of independently operating, state-free components.
In this work we capitalise on this stringent application architecture for fault-tolerance against both permanent and transient hardware failure. We extend the TeamPlay language by a range of fault-tolerance features to be selected by the system integrator. We further propose a multi-core runtime system that is able to isolate hardware failures and manages to keep an application running flawlessly in the presence of hardware failure through adaptively morphing the application.

1 Introduction

Cyber-physical systems (CPS) are a computing paradigm which involves systems that interact with both the hardware they run on and the physical world around us [1]. Examples of this can be found in several areas such as self-driving cars, autonomous drones, robotics, and building & environmental control. Commonly, these systems also have non-functional requirements. These requirements are manifested in facets such as timing, energy-consumption, security, and robustness [2, 3]. These requirements often involve trade-offs; e.g., executing an action in less time usually incurs higher power consumption. Adding robustness or fault-tolerance through redundancy likewise incurs a higher power consumption and a higher response time or more hardware to meet deadlines.

Safety-critical systems (especially in multi- or heterogeneous systems) require extensive validation, which is hard to do when non-functional properties, like fault-tolerance, are intertwined with computational code. In the CPS paradigm, the scientific community has been calling out to the creation of higher-level abstraction layers to alleviate the burden on the programmer [2, 4, 3].

We propose the coordination language TeamPlay [5]. Coordination languages [6] can be used to manage the interaction between separate activities or components into an often parallel system [7]. The TeamPlay coordination language [5] enables the management of non-functional aspects of cyber-physical systems

and facilitates the separation of concerns between coordination and computation code. Coordination code defines the structure of the application and manages non-functional properties on a high level. The computation code, in contrast, can focus on functional correctness while letting the coordination code actively manage the non-functional properties.

2 Coordination model

The term *coordination* goes back to the seminal work of Gelernter and Carriero [8] and their coordination language Linda. Coordination languages can be classified as either *endogenous* or *exogenous* [9]. Endogenous approaches provide coordination primitives within application code; the original work on Linda falls into the category. We pursue an exogenous approach that completely separates the concerns of coordination programming and application programming. Software *components* serve as the central artefact in between.

2.1 Components

Our exogenous approach fosters the separation of concerns between intrinsic component behaviour and extrinsic component interaction. The notion of a component is the bridging point between low-level functionality implementation and high-level application design. We illustrate our component model in Figure 1. Following the keyword `component` we have a unique component name that serves the dual purpose of identifying a certain application functionality and of locating the corresponding implementation in the object code.

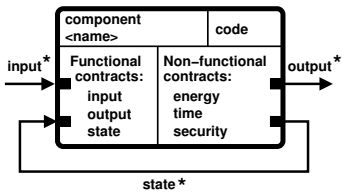


Fig. 1. Illustration of component model

A component interacts with the outside world via component-specific numbers of typed and named input ports and output ports. As the Kleene star in Figure 1 suggests, a component may have zero input ports or zero output ports. A component without input ports is called a *source component*; a component without output ports is called a *sink component*. Source components and sink components form the quintessential interfaces between the physical world and the cyber-world characteristic for cyber-physical systems. They represent sensors and actors in the broadest sense. We adopt the firing rule of Petri-nets, i.e. a component is activated as soon as data (tokens) are available on each input port.

Technically, a component implementation is a function adhering to the C calling and linking conventions whose name and signature can be derived from the component specification in a defined way. This function may call other functions using the regular C calling convention. However, the execution of the function,

including execution of all subsidiary functions, must not interfere with the execution environment. Exceptions to the former restriction are source and sink components that are supposed to control sensors and actors.

2.2 Stateful components

Our components are conceptually stateless. However, some sort of state is very common in cyber-physical systems. We model such state in a functionally transparent way, as illustrated in Figure 1. We employ so-called *state ports* that are short-circuited from output to input. In analogy to input ports and output ports, a component may well have no state ports, which is what we consider the norm rather than the exception.

Our approach to state is in an interesting way not dissimilar from mainstream purely functional languages, such as Haskell or Clean. They are by no means free of state either, for the simple reason that many real-world problems and phenomena are stateful. However, purely functional languages apply suitable techniques to make any state fully explicit, be it monads in Haskell [10] or uniqueness types in Clean [11]. Making state explicit is key to properly deal with state and state changes in a declarative way. In contrast, the quintessential problem of impure functional and even more so imperative languages is that state is potentially scattered all over the place. And even where this is not the case in practice, proving this property is hardly possible.

2.3 Non-functional properties

We are particularly interested in the non-functional properties of code execution, namely energy, time and security while we now add fault-tolerance for robustness against hardware failure, both permanent and transient. Hence, any component not only comes with functional contracts but additionally with non-functional contracts. These contracts can be inherently different in nature. Execution time and energy consumption depend on a concrete execution machinery. In contrast, security, more precisely algorithmic security, depends on the concrete implementation of a component, e.g. using different levels of encryption. However, different security levels almost inevitably incur different computational demands and, thus, are likely to expose different runtime behaviour in terms of time and energy consumption as well.

2.4 Multi-version components

As illustrated in Figure 2, a component may have multiple versions, each with its own energy, time and security contracts, but otherwise identical functional behaviour. More security requires stronger encryption which requires more computing and, thus, more time and energy. However, many systems do not need to operate at a maximum security level at all times. Take as an example a reconnaissance drone that adapts its security protocol in accordance with changing

mission state: low security level while taking off or landing from/to base station, medium security level while navigating to/from mission area, high security level during mission.

In low security mode, the drone can use a less resilient encryption when communicating with the base station while highest possible security is paramount in a potentially hostile environment. Continuous adaptation of security levels results in less computing and, thus, in energy savings that could be exploited for longer flight times. Our solution is to embed different versions of the same component that are all functionally equivalent, but expose different trade-offs regarding non-functional properties, similar to [12].

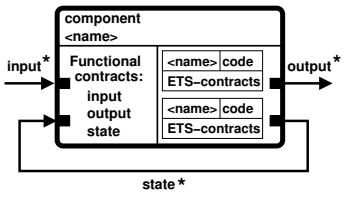


Fig. 2. Multi-version component with individual energy, time and security contracts

2.5 Component interplay

Components are connected via channels to exchange data, as illustrated in Figure 3. Depending on application requirements, components may start computing at statically determined time slots, when all input data is guaranteed to be present or may be activated dynamically by the presence of all required input data. Components produce output data on all or on selected output ports.

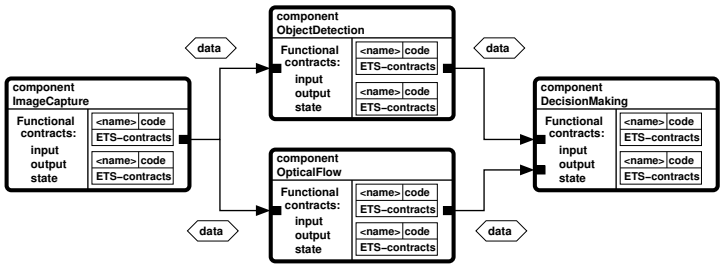


Fig. 3. Illustration of data-driven component interplay via FIFO channels

3 The TeamPlay Language

Figure 4 shows the TeamPlay coordination code an imaginary subsystem of a car, where two sensors feed messages to a decision controller, which synchronises the messages pair-wise and sends commands to two subsequent actuators.

A TeamPlay application definition starts with the keyword `app` followed by an identifier that serves as the application's name. Enclosed within curly brackets we can identify the two major code regions of any TeamPLay coordination

program: components with their inports and outports as well as channels connecting outports to inports. The syntax of inport and outport specifications is inspired by C struct definitions: pairs of type name and port name separated by semicolons and embraced in curly brackets. The mapping of type names used on the coordination level and type implementations in compiled code as wel as in component implementations is not provided through the coordination program, but rather through a separate configuration file.

```
app car {
  components {
    DistSensor {
      outports {num dist}
    }
    ImageCapture {
      outports {frame frameData}
    }
    Decision {
      inports { num dist; frame frameData}
      outports { num voltage}
    }
    LeftActuator {
      inports {num voltage}
    }
    RightActuator {
      inports {num voltage}
  } }
  channels {
    DistSensor.dist -> Decision.dist;
    ImageCapture.frameData -> Decision.frameData;
    Decision.voltage -> LeftActuator.voltage
                      & RightActuator.voltage;
} }
```

Fig. 4. TeamPlay code for example of Figure **??**

3.1 Components

Components serve as representations of stateless computations that map input data tokens on typed incoming streams to output data tokens emitted on typed output streams. Following the approach of exogeneous coordination the actual computation is outside the scope of the coordination code. We link our compiled coordination code with independently provided and compiled component implementation code. Given our focus on cyber-physical systems we assume component implementations to be written in C or possibly in C++. A component definition starts with a name followed by a pair of curly brackets enclosing further information about the component.

As components communicate with other components via (FIFO) channels, the corresponding ports are the most vital functional properties (or contracts) of components. Following the key words `inports`, `outports` or `state` (The latter is not shown in the running example, but the syntax is identical.) we have a list of pairs of port type and port name adopting a syntax inspired by C struct defintions.

3.2 Channels

Components are connected with each other via *channels*. In Figure 4 we can identify two kinds of channels: regular channels connect a single outport to a single inport while broadcast channels using an ampersand send a single data item from one outport to multiple inports. The TeamPlay compiler applies static analyses to guarantee type correctnes, absence of cycles and restriction that any port is connected to at most one channel. Normally, ports are identified by component name and port name separated by a dot, but the port name may be omitted if a component only has a single outport or a single inport.

3.3 Non-functional properties

As pointed out before, one of the goals in the design of the TeamPlay coordination language is the active management of non-functional properties, namely energy, time and security. Both energy and time can only be considered in relation to some concrete execution machinery. Thus, any mentioning of energy or time in the coordination source code would inherently make the code hardware-specific, which is not what we want. In contrast, we employ a *non-functional properties file (NFP)* that functions as a data base storing per component time and energy consumption values for the variety of hardware execution units of interest (and perhaps even DVFS settings). Depending on the concrete hardware properties, concrete values can be derived from static code analysis, dynamic profiling or simply asserted by the user.

```
components {
  Encryption {
    inports { frame original}
    outports { frame encrypted}
    security 4;
    arch "arm/big"
}
```

Fig. 5. Example of a component with non-functional properties: security and architecture

The third non-functional property of interest, security, differs from energy and time as security (in our interpretation of the word) is an algorithmic or

code property that is independent of the actual execution machinery. As demonstrated in Figure 5, line 5, TeamPlay supports the specification of a security level in form of a natural number, using the `security` key word. Here, we interpret higher numbers as denoting better security. Any concrete meaning of security levels, however, are application-specific. In a similar way we can specify a class of hardware on which the component must be scheduled for execution using the `arch` key word and a string.

3.4 Multi-version components

With our focus on non-functional properties, it becomes particularly interesting to have multiple versions of a component that expose identical functional behaviour, but that implement different trade-offs of the non-functional properties of interest. Figure 6 demonstrates how this can be accomplished in TeamPlay.

```
components {
  Encryption {
    inports {frame original}
    outports {frame encrypted}
    version WeakerEncryption {security 4}
    version MediumEncryption {security 6}
    version StrongEncryption {security 9}
} }
```

Fig. 6. Example of a multi-version component. The Encryption component has three different implementations, each with a different security value.

The new definition of the `Encryption` component features three different versions, distinguished by three different security levels. Different versions of one component all share the same port specifications and must behave identically from the functional perspective.

4 Extensions for fault-tolerance

We extend the core TeamPlay language by a range of fault-tolerance methods. We opt for a user-directed approach where the user can specify which of the predefined options to apply in different parts of the application. This is due to major challenges in having a compiler or scheduler analyse the criticality of a component in the application as a whole. Furthermore, the way fault-tolerance is implemented and achieved needs to be transparent to the programmer in order to make sure they they fit the application requirements.

4.1 Checkpoint/restart

Checkpoint/restart lets the system return to a stable (backup) state when a fault has occurred [13, 14]. Generally, the downside of checkpoint/restart methods is

the concrete state of some failing software unit is difficult to assess and, thus, in the worst case the entire process image needs to be saved at each checkpoint. That is prohibitively expensive, both in storage space and execution time.

Here the architecture of our coordination-based approach pays off. It creates a middleware layer where our system software can precisely keep copies of the arguments of an individual component invocation before giving control to the third-party provided component implementation. The stateless nature of TeamPlay components ensures that no other data affects the computation. Note here that this property remains valid even if state ports are used as described in Section 2.2. The backup copies of the argument values only need to be stored while the component is computing. As soon as it emits its output data on its outports, the backup copies can be discarded.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  checkpoint {}
}
```

Fig. 7. Defaults of the checkpoint/restart specification.

The benefits of using checkpoint/restart are (potentially) three-fold: implementation is straightforward, coordination between hardware components is not needed, and it only requires extra memory and copy time but no redundant active components. Figure 7 shows how checkpoint/restart can be specified on the `Decision` component from Figure 4. Currently, our specification of checkpoint/restart has no supported options, hence the empty pair or curly brackets.

4.2 Standby or primary-backup

In standby or primary-backup methods, standby components can take over the active computing component in case of failure. Initially, the output of the primary process is used. Should a fault be detected, the output of the standby component is used instead. A distinction can be made between cold, warm and hot standby which differ in the amount of synchronisation the backup components have to the active components [15]. These types can be distinguished as follows:

cold: backup component(s) are initialised but then turned off;
warm: backup component(s) are synchronised with primary component at specified points;
hot: backup component(s) are continuously synchronised with primary component.

In case of crash failures, components with long startup times benefit from this type of synchronisation because the working component can be taken over

faster when using primary-backup [15]. Generally, the number of required computing resources for primary-backup are lower compared to methods like NMR. Especially using cold standby can save a lot of energy resources, which is important in (often) battery powered systems like those in the cyber-physical class. The main disadvantage of this method is that a fault needs to be detected before the redundant component can take over. Furthermore, primary-backup cannot detect value faults (i.e., there is no voting) and it relies on an error detection method to detect faults, so that the active component can be taken over. Thus, this method is primarily useful in systems which require fast switch times while keeping a state close to the state of the original, crashed process [15].

In primary-backup, the state of the primary and standby component is synchronised. The degree of this synchronisation depends on which flavour, cold, warm or hot is implemented. This allows the application to quickly switch outputs when a fault is detected. In TeamPlay, the firing of a component is discrete, i.e., every execution produces output tokens only once. This, together with the fact that state is made explicit in the buffers of the edges, means that it is not necessary to run the primary and standby components at the same time, i.e., they do not have to be synchronised. We can simply provide copies of the input tokens to the hardware units and take the first unit who delivers an output as the primary component. If it fails, one of the standby components will deliver output instead. This makes this method predictable as one does not have to account for switching from the primary to the replica component or synchronisation mechanisms.

Figure 8 shows the way primary-backup can be specified. The specified options again use default values. The `replicas` option can be specified as an integer denoting the number of replicas to run for this component.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  standby {replicas 2}
}
```

Fig. 8. Defaults of the primary-backup specification.

4.3 N-Modular redundancy

A classic example of physical redundancy is N-modular redundancy (NMR). In this strategy, n independent identical processes are executed with identical input [13, 15]. These n processes are followed by voting processes, which vote which answer they will be outputting. This method primarily focuses on masking transient faults. Depending on the fault-model for the application, it can be possible that the voter processes fail. In order to decrease the chance of this happening it is possible to increase the number of voters [16].

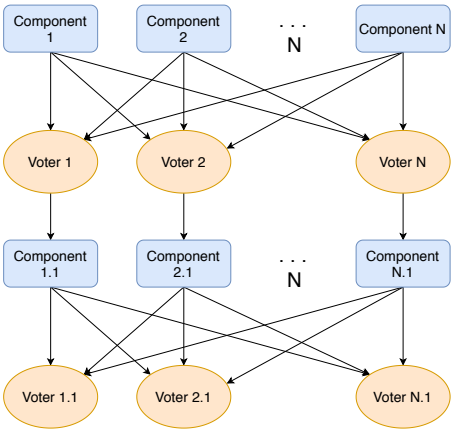


Fig. 9. N-modular redundancy

If a minority of the computational processes have faults, a majority vote will still result in the correct answer. Triple modular redundancy (TMR) [17, 13, 15] is a special case of NMR in which n is minimally chosen such that the computation does not have to be repeated (when a single fault is present). Since we can't know which process is likely to be faulty in case $n = 2$. However, this double-modular redundancy method has uses as an error detection method since it can be used to detect transient errors. Figure 9 illustrates N-modular redundancy with a pipeline consisting of two stages of components followed by voting processes. NMR is a mechanism that deals with transient faults without employing low level (hardware) error detection techniques. Furthermore, NMR is a time-predictable method, i.e., it is suitable for use in real-time systems [18]. Figure 10 shows the defaults of the N-modular redundancy. We support the following options:

- `replicas` (line 6), integer signifying the number of replicas. Default is `3`, meaning a TMR setup.
- `votingReplicas` (line 7), integer signifying whether and how much the voting processes need to be replicated.
- `waitingTime` (line 8), how long processes should wait before initiating the voting process. Given as a percentage of the average execution time of the finished components, the percentage can be higher than `100%`.
- `waitingStart` (line 9), defines the starting point of waiting. When `waitingStart (cont.)` is `majority`, processes start waiting based on the execution time when a majority of processes are done. In the case of `single`, the waiting will start when a single process is ready.
- `waitingJoin` (line 10), boolean defining whether processes that are finished later should be added in the waitingTime calculation. Can apply on both a `waitingStart` value of `majority` and `single`.

4.4 N-version programming

In N-version programming (NVP) multiple functional equivalent implementations of the same component are created [19]. At runtime, they can be run in the same way as when using N-modular redundancy. The advantage of NVP over NMR is that software faults present in one implementation are caught the same way as transient hardware faults are caught.

```
Decision {
  inports {frame frameData; int dist}
  outports {int voltage}
  nModular {
    replicas 3
    votingReplicas 2
    waitingTime 30
    waitingStart majority
    waitingJoin true
} }
```

Fig. 10. Default options for N-modular redundancy

The disadvantage of NVP is that it combines high runtime overhead with additional development cost. However, TeamPlay already supports the concept of multi-version components. What has primarily been intended to exploit different energy/time/security trade-offs, can now be reused for fault-tolerance. By leveraging our existing version mechanic, the programmer can kill two birds with one stone by reusing existing versions for NVP.

The options we support in NVP are similar to N-modular redundancy (Section 4.3) adding an option to specify `versions`, which defines which versions should be used and how many of each of these versions should be run. This is ilustrated in Figure 11.

```
components {
  Encryption {
    inports {frame original}
    outports {frame encrypted}
    version Encryption1 {security 4}
    version Encryption2 {security 6}
    version Encryption3 {security 9}
    nVersion {
      versions [ (Encryption1, 2) (Encryption2, 1) ]
} } }
```

Fig. 11. Example of `versions`. The first entry in the tuple specifies the version while the second specifies how many replicas of that version should exist. Not all versions have to be specified because the default value is `0`.

5 Fault-tolerant runtime system

In order to support the various TeamPlay language extensions specifically geared at fault-tolerance we designed and implemented a corresponding fault-tolerant runtime environment that dynamically reconfigures running applications upon

detection of hardware failures. This runtime environment targets both permanent faults as well as transient faults. The runtime environment comes with a fault injection facility for demonstration purposes.

5.1 Base system

The coordination approach we take requires us to make as few assumptions as possible about the underlying target hardware configuration as our coordination approach aims to be hardware architecture agnostic. We assume that the main property of CPS(oS) holds: CPS(oS) are distributed (possibly heterogeneous) systems which are not necessarily in the same physical location [1]. This means we deal with nodes of hardware components. We simulate these distributed systems using a thread for each node with *PThreads*.

One of the first decisions we need to make is: in what way do threads in the simulator correspond to the real world? A straightforward idea is that each coordination component corresponds with a thread. This has the advantage that it is not necessary to manage the threads separately. Since the coordination task graph is static, each component knows which thread to communicate their output data to. In other exogenous streaming coordination systems like S-Net [20] (which targets HPC systems), having components correspond with threads is feasible. But when constructing a simulator for a CPS(oS) it is not realistic to assume that there are always sufficient hardware components to accommodate each coordination component separately. Hence we choose for an architecture in which the number of computation threads is static, related to the number of hardware components in the CPS(oS) but unrelated to the number of coordination components. This requires us to work with task queues, as each thread can execute multiple components.

We choose for a design in which there are two types of threads, a main or control thread and multiple computation threads. This design is illustrated in Figure 12. The control thread checks whether components are ready and puts the tasks into their appropriate queues. It is activated as soon as a component has finished computing and matches a centralised hardware component in a real world architecture. The choice for a centralised system is motivated by security concerns as it makes it more difficult to disrupt the entire system by taking control of a single computation node. In reality, this centralised system will have to be hardened against security faults. In order to deal with faults in this management hardware component, fault-tolerance methods such as primary-backup or n-modular redundancy can be applied. The computation threads mirror the hardware components running the actual coordination component code from the real world.

5.2 Thread interaction

The interaction between the control thread and computation threads is displayed in Figure 13. The blue block on the left and green on the right indicate whether the action takes place in the worker threads or in the control thread. In the

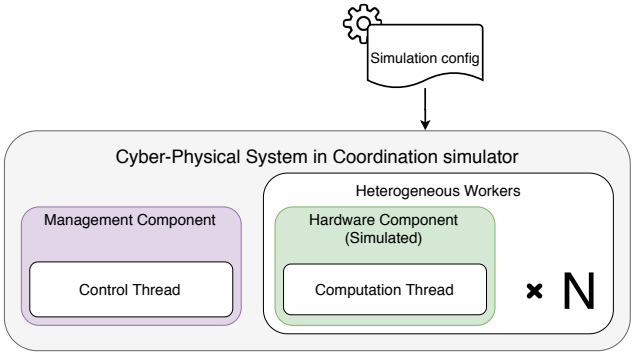


Fig. 12. Illustration of the base architecture of the simulator. The management component is simulated with the use of a control thread. The heterogeneous worker elements are simulated by n computation threads.

figure we show only one worker thread to highlight the interaction with the control thread. The figures in the middle show the shared data structures between the control and computation threads. The top data structure is the task queue associated with the thread. The middle figure is the coordination structure containing the coordination task graph. The bottom structure is the finished list which is used to communicate that a thread is finished and tokens are added to the buffers. The tables in the figure show the first four iterations on the simple coordination structure in the middle.

When the control thread launches (top right node), it goes through the source nodes and adds them to the task queues of the assigned threads. Each of these threads has a counting semaphore which corresponds to the number of items in their queue. When the semaphore reaches zero, the threads wait until new items appear in the task queue.

When a thread is alerted that new items appeared in the task queue (top data structure), it pops a component from the task queue (task queues are FIFO) to execute. After execution, it stores the output data in the graph data-structure and appends the id of the executed component into the finished list. The control thread is alerted that components are finished, so it can check whether new items can be added to the task queues. The computing thread will then wait for the task queue semaphore. If the semaphore's value higher than zero it can continue popping another item from the task queue to start computing again.

After items are added to the task queues of the threads and the threads are alerted, the management thread will wait until items appear in the finished list. This is indicated in the figure by the bottom data structure with the dotted line facing right. This mechanism is implemented with a condition variable as one cannot reset a counting semaphore when the finished list is emptied. When items appear in the finished list, we need to check which components can fire again. First, we need to check whether the predecessors of the finished component can fire since, by firing, it can have opened a spot in the (bounded) FIFO buffers

of the predecessors. Then, we check whether the successors of this component can fire since it has produced a token on its outports which may trigger the firing rule of the successor. Finally, we check whether the component itself can fire again. This way of checking ensures we only have to traverse the parts of the graph that have been changed. The components that can fire are added to the task queues belonging to the threads and the threads are alerted so they can continue computing. The components that are ready are added to the task queue. This marks the completion of a cycle.

Now we will explain the example found in the figure. First, both threads will launch. The worker thread sees that there are no items in the task queue (i.e., semaphore value is zero) so it will wait. In the first cycle, the control thread adds the `Source` component to the task queue. As the source component does not have any dependencies, it can fire as long as the buffers can hold the data and it is not already present in any task queue. The task is put in the task queue associated with the computing thread to which `Source` is assigned. The control thread will increment the semaphore. This leads to the awakening of the worker thread, which will pop `Source` from the FIFO queue. The worker thread will then execute the code associated with `Source` component. After computation, the output token of `Source` will be added to the buffer on the edge leading to the next component, `A`. The computing thread puts the id of the `Source` component into the finished list and sends a signal to the condition variable on which the control thread is waiting. The computing thread loops back to the first item (after initialisation) and will wait until the control thread has added new items to the task queue owned by the worker thread.

When the control thread receives the signal for the condition variable, it will loop trough the finished list and check the task graph for components which are ready. This is done by looping trough the predecessors, successors and the component itself, to see if they can fire. `Sink` has no predecessors but it does have one successor, `A`, which can fire since `Source` just fired. `Source` can also fire again. The components which can fire again are put into the task queue. Next, component `A` is fired, the result is again stored in the buffer after the fired component, this time leading to `Sink`. Then the control thread is again alerted that the worker thread has finished a computation. The control thread notices that `Source` can be fired since it has no dependencies, but it is already in a task queue, so it cannot be added again. Following the execution of `A`, `Sink` can be fired, but `A` has insufficient tokens from `Source` to fire again. Now, `Source` is taken from the task queue and executed, as it was added the previous cycle. The component checking process of this cycle is identical to the first cycle, as `Source` and `A` are added again. Then for the last round explained in this example, `Sink` will be popped from the task queue and executed. In the control thread, `A` cannot be added to the task queue again since it was already added when `Source` finished. `Sink` cannot fire again since the buffer on the edge coming from `A` does not have sufficient tokens.

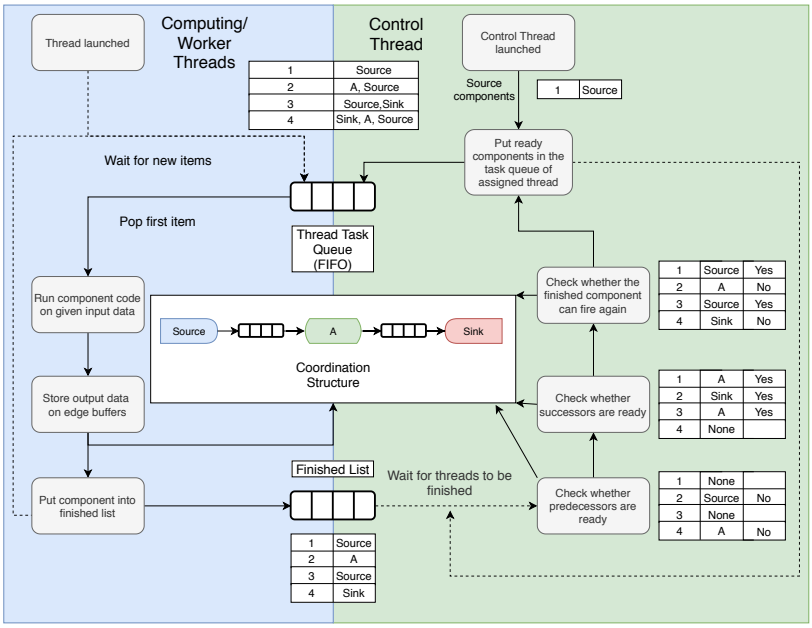


Fig. 13. Schematic overview of the interaction between the computation components and the management component. The dotted lines indicate a waiting process via thread communication (e.g., condition variable or semaphore). The double-column tables show the components in the list per iteration. The triple-column tables show the components which are checked for firing, the third column shows whether they are added to the task queue.

5.3 Configuration file

Our simulation run-time uses a configuration file in which the user can specify options such as the number of threads and options related to fault-tolerance. Figure 14 shows an example of a configuration file. `numThreads` signifies the number of computation threads. Setting `debug` to `true` turns on debug prints. `sleepTime` is the period of the heartbeat worker threads in microseconds. `controlSleep` is the period of the heartbeat control thread in microseconds. `heartbeatTries (cont.)` is the threshold of the counter incremented by the heartbeat control thread, if the counter is higher than `heartbeatTries`, it is deemed to have crashed. `heartbeatCheckerPrio` and `heartbeatWorkerPrio` are the real-time scheduling priority of the control heartbeat thread and heartbeat worker threads respectively. Setting `standbyEarlyTaskCompletion` to `true` allows standby threads which start computation after the primary thread (i.e., first finished thread) has finished to skip computing since the task is already delivered. The `edgeBufferSize` indicates the number of tokens an edge buffer can hold.

```
numThreads = 6
debug = false
sleepTime = 100
controlSleep = 1000
heartbeatTries = 10
heartbeatCheckerPrio = 10
heartbeatWorkerPrio = 15
standbyEarlyTaskCompletion = false
edgeBufferSize = 20
```

Fig. 14. Simulator configuration file example.

5.4 Checkpoint/restart

Checkpoint/restart can be implemented in the coordination language by checkpointing the FIFO buffers on the edges between the components, as the state of the entire application resides in these buffers. This is done in practice by adding an extra buffer on each outport that leads to a component. After the execution of the previous components, (i.e., the dependency components), copies of the output tokens ares made. For primitive types, this is an easy task but for user-defined types for which only a pointer is passed, the user needs to provide a copy function. When the thread executing the component fails, a new structure of input tokens is created from the checkpointed buffer and assigned to the task which takes place during the rejuvenation phase. The entire rejuvenation process, in which checkpoint/restart plays a role, is illustrated in Figure 15. We will visit this figure in full during the rejuvenation section. In normal operation, we need to remove the checkpointed data upon finishing execution and delivering the output, in order to prevent the buffers from overflowing.

5.5 Primary-backup

In primary-backup, a standby component takes over the main component when a failure is detected. Usually, this is done directly as the backup component synchronises with the active component to ensure a quick switch. In our coordination language, we do not need this behaviour as, again, the state of the application is completely saved in the FIFO buffers. We assign copies of the input tokens of the component to a number of threads equal to the number of replicas. The first thread that finishes the computation actually delivers the output. If a thread starts the computation after another thread has already delivered its answer, the thread starting the second computation can skip the task. However, it is unlikely that this behaviour is schedulable on real-time systems. Thus, we build an option into the simulator whether this form of task completion is allowed. Disabling this setting gives us the worst case, all threads compute even if the task is already delivered. When this setting is enabled, the task will only be computed multiple times if the backup threads start the computations while the thread that will finish the earliest has not yet finished. Again, the primary-backup rejuvenation process is illustrated in Figure 15.

5.6 N-version programming & N-modular redundancy

N-modular redundancy requires more control over the processes compared to primary-backup since NMR utilises a voting process at the end. What we can use from the primary-backup system, is that these processes do not have to execute at the same time. When all processes are done or when one of the threads executing the processes has crashed or sustained a timeout conform the coordination settings, the voting process is executed on a designated voter node. This voter node requires a copy of all output tokens in order to execute the majority voting process. Our application requires that the output of NMR is a single answer, since the next component may not have NMR specified. This is why we cannot have a voter array without having an extra voting step afterwards to choose one correct answer from the voter replicas.

5.7 Component assignments & heterogeneous architectures

In order to support heterogeneous rejuvenation, we present an extension to the base system. Before we can create the rejuvenation mechanism we need to know which thread can be reconfigured to do which tasks. In time-critical systems, TeamPlay components are assigned to a hardware component in both the time and space dimensions. As the timing dimension is out of the scope of this work, we introduce a spatial assignment of the components to threads. This way, the system provides the minimum to test rejuvenation mechanisms for our fault-tolerance methods.

We further introduce the concept of a thread class. This class mirrors a specific hardware architecture (as threads mirror hardware components) in heterogeneous systems. When a component needs to be reconfigured because it has crashed, it needs to be assigned to a component of the same class to ensure compatibility. The mapping of components to threads are passed to the simulation run-time. It is not embedded in the coordination language as the number and types of threads are hardware architecture-specific.

5.8 Rejuvenation

Strategies that deal with crash faults, checkpoint/restart and primary-backup require this mechanism as it is not guaranteed that crashed nodes operate normally when restarted. In our runtime architecture a hardware component is mimicked by two threads, a computation thread and a heartbeat thread. These hardware components form a group of heterogeneous workers, managed by the management component. The management component consists of two threads. The control thread checks and adds components to the task queue of the worker threads. The main heartbeat thread monitors the heartbeat threads associated with the workers and launch the rejuvenation process.

The rejuvenation process is illustrated in Figure 15. The process can be split into two parts: the invalidation and recovery of the task queue of the crashed thread and the reassignment of the threads' assigned components. The path of

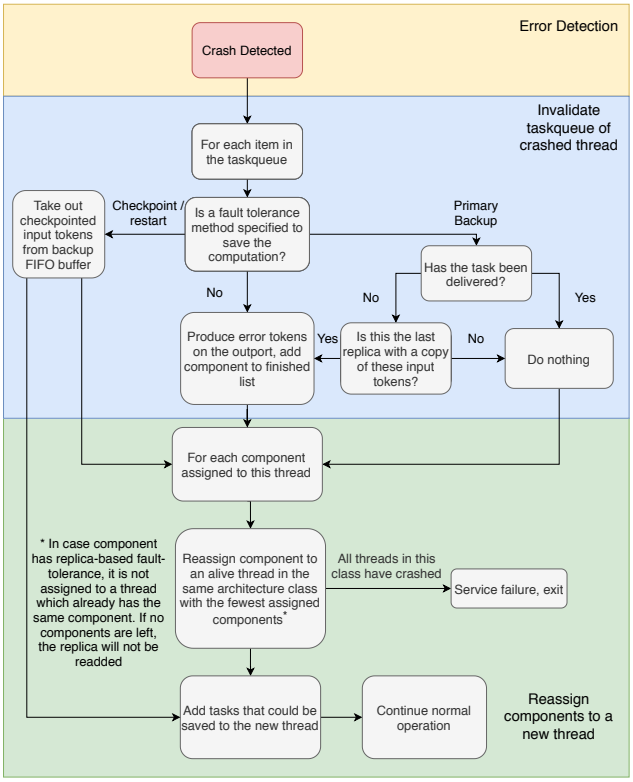


Fig. 15. Illustration of a crashed thread rejuvenation mechanism.

the rejuvenation process depends on which fault-tolerance mechanisms are specified. First, we explain the middle path which is taken when no fault-tolerance method is specified on the component. On this path, error tokens are produced on the outports of the components in the task queue. The component is marked as finished as the computation could not be saved by a fault-tolerance method. Then we arrive at the rejuvenation process. This rejuvenation process works by finding a non-crashed thread in the same architecture class with as extra condition that it is the thread with the fewest assigned components, to prevent from one thread taking all crashed tasks, consequently becoming a bottleneck for the application. We do not produce error tokens if a source component is present in the task queue of the crashed thread as it can simply fire again since it does not have any input tokens that need to be invalidated.

6 Conclusions

In cyber-physical systems, there is still a lot to be done in the area of creating tools, frameworks, and languages to aid the programmer in processes related to software evolution like creating and maintaining [2, **?**,**?**]. We have taken the first steps towards facilitating separation of concerns between computation and coordination code, thus creating the opportunity to manage non-functional properties like fault-tolerance separately from computation code.

In this work, we have extended the TeamPlay coordination language [5] by a number of fault-tolerance methods that provide fine-grained control over various forms of replication from N-modular redundancy to multi-version programming. Furthermore, we have devised a fully-fledged rutime environment that seamlessly runs TeamPlay coordination code with the specified fault-tolerance.

We are currently pursuing two directions of research. First, we plan to integrate the fault-tolerance extensions of TeamPlay as described in this paper with the time- and energy-aware heterogeneous multi-core scheduling techniques that we have developed over recent years [21–23]. Second, we work on statistical methods to quantify the impact of fault-tolerance techniques on the system reliability in the presence of single-event upsets [24, 25]. Here, we particularly address weakly-hard real-time systems, where components are permitted to fail a bounded number of times in a gliding average before disaster strikes [26].

Acknowledgement

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 871259 (ADMORPH).

References

1. Baheti, R., Gill, H.: Cyber-physical systems. The impact of control technology **12** (2011) 161–166
2. Park, K.J., Zheng, R., Liu, X.: Cyber-physical systems: Milestones and research challenges. Computer Communications **36** (2012) 1–7
3. Rajkumar, R.Y., Lee, I., Sha, L., Stankovic, J.A.: Cyber-physical systems: The next computing revolution. Design Automation Conference (2010) 731–736
4. Romanovsky, A.: A looming fault tolerance software crisis? ACM SIGSOFT Software Engineering Notes **32** (2007) 1–4
5. Roeder, J., Rouxel, B., Altmeyer, S., Grelck, C.: Towards energy-, time- and security-aware multi-core coordination. In Bliudze, S., Bocchi, L., eds.: 22nd International Conference on Coordination Models and Languages (COORDINATION 2020), Malta. Volume 12134 of LNCS., Springer (2020) 57–74
6. Gelernter, D., Carriero, N.: Coordination languages and their significance. Communications of the ACM **35** (1992) 96–108
7. Arbab, F., Ciancarini, P., Hankin, C.: Coordination languages for parallel programming. Parallel Computing **24** (1998) 989 – 1004

8. Gelernter, D., Carriero, N.: Coordination Languages and their Significance. Communications of the ACM **35** (1992) 97–107
9. Arbab, F.: Composition of interacting computations. In Goldin, D., Smolka, S., Wegner, P., eds.: Interactive Computation. Springer (2006) 277–321
10. Peyton Jones, S., Launchbury, J.: State in Haskell. Lisp and Symbolic Computation **8** (1995) 293–341
11. Achten, P., Plasmeijer, M.: The ins and outs of Clean I/O. Journal of Functional Programming **5** (1995) 81–110
12. Rusu, C., Melhem, R., Moss'e, D.: Multi-version scheduling in rechargeable energy-aware real-time systems. Journal of Embedded Computing **1** (2005) 271–283
13. Tanenbaum, A.S., Van Steen, M.: Distributed systems: principles and paradigms. Prentice-Hall (2007)
14. Sultana, N., et al.: Toward a transparent, checkpointable fault-tolerant message passing interface for hpc systems. (2019)
15. Oriol, M., Gamer, T., de Gooijer, T., Wahler, M., Ferranti, E.: Fault-tolerant fault tolerance for component-based automation systems. In: Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems. (2013)
16. Avizienis, A., Laprie, J.., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1** (2004) 11–33
17. Lyons, R.E., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM Journal of Research and Development **6** (1962) 200–209
18. Poledna, S.: Fault-tolerant real-time systems: The problem of replica determinism. Volume 345. Springer Science & Business Media (2007)
19. Peng, Z.: Building reliable embedded systems with unreliable components. In: ICSES 2010 International Conference on Signals and Electronic Circuits. (2010)
20. Grelck, C., Penczek, F.: Implementation architecture and multithreaded runtime system of S-Net. In: Symposium on Implementation and Application of Functional Languages (IFL 2008), LNCS 5836, Springer (2011) 60–79
21. Roeder, J., Rouxel, B., Altmeyer, S., Grelck, C.: Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems. In: 36th ACM/SIGAPP Symposium on Applied Computing (SAC 2021), ACM (2020) 500–510
22. Roeder, J., Rouxel, B., Grelck, C.: Scheduling DAGs of multi-version multi-phase tasks on heterogeneous real-time systems. In: 14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021), Singapore, IEEE (2021)
23. Roeder, J., Pimentel, A., Grelck, C.: GCN-based reinforcement learning approach for scheduling DAG applications. In: Artificial Intelligence Applications and Innovations, 19th IFIP WG 12.5 International Conference, AIAI 2023, León, Spain. Volume 676 of IFIPAICT., Springer (2023) 121–134
24. Miedema, L., Grelck, C.: Strategy Switching: Smart Fault-tolerance for Weakly-hard Resource-constrained Real-time Applications. In: Software Engineering and Formal Methods, 20th International Conference, SEFM 2022. Volume 13550 of Lecture Notes in Computer Science., Springer (2022) 129–145
25. Miedema, L., Grelck, C.: Change of plans: optimizing for power, reliability and timeliness for cost-conscious real-time systems. In: 26th Euromicro Conference on Digital System Design (DSD 2023), Durrës, Albania. (2023)
26. Bernat, G., Burns, A., Liamosi, A.: Weakly hard real-time systems. IEEE Transactions on Computers **50** (2001) 308–321

Self-Active Objects: A Unifying Approach towards Structured Concurrency

Jürg Gutknecht, Prof. em. ETH Zürich, September 7, 2023

Abstract

While the development of software models from sequences of machine instructions operating on a uniform linear address space all the way up to structured data and metaphoric worlds of classes and objects supported by sophisticated runtime mechanisms such as automatic garbage collection is truly impressive, parallelism in programming languages has largely been treated stepmotherly.

Programmers occasionally writing concurrent software still need to rely on theoretic and historic concurrency models dealing with “critical sections”, “race conditions”, “semaphores” [1], “communication channels [2] and the like, or some workarounds badly compatible with modern programming styles. This is particularly relevant in view of multicore and manycore processing hardware now all around when concurrency is becoming the norm and real in contrast with just exceptional and simulated. We argue in favor of amalgamating concurrency with modern programming by changing the paradigm from “code operating on data” towards “data becoming self-active and operating on the environment”. Our final concern is converting concurrency from explicit to implicit in programming. The obvious price to pay for such a metaphoric integration of concurrency on the level of the programming model and language is a substantial extension of the runtime support behind the scenes.

Embarrassingly Parallel Processing

Let us take a simple but archetypal example of an iteration through a number of mutually independent computations. The problem is to use a Monte Carlo simulation for computing an approximation to π, something like this:

```
MonteCarlo(int n) {
  double count = 0;
  for(i = 0; i < n; i++) {
    x = randr(-1, 1);
    y = randr(-1, 1);
    if (x*x + y*y <= 1) count++;
  }
  return count;
}
```

Obviously, random points can be generated in any order without compromising the correctness of the result. Considering the computational effort in the body of the *for* statement and the availability of multiple cores, a parallel version is surely the preferred option here.

Now, how would such a parallel version of the *MonteCarlo* algorithm look like when expressed in terms of tools typically available today? Obviously, the very first idea is to simply replace the *for* statement with some parallel counterpart like *parfor*, indicating that its body may be executed in parallel. However, this does not work because *count* is a “shared variable”, shared among all running instances of the body of the loop, meaning that care must be exercised to protect it from “race conditions”. A race condition would occur, if *count++* would be called simultaneously by several bodies in the loop. Using a symbolic magnifier glass the following sequence of atomic actions could result:

```
body 1 loads the value of count into its accumulator
body 1 adds 1 to its accumulator
body 2 loads the value of count into its accumulator
```

```
body 2 adds 1 to its accumulator
body 1 stores the result to count
body 2 stores the result to count
```

Obviously wrong! What would be needed to remedy the situation in this and all other such cases is called “mutual exclusion” of modifying accesses to shared variables. Furthermore, apart from race conditions impairing the correctness, differentiating between shared and non-shared variables is of importance from an efficiency perspective: non-shared variables may remain in accumulators local to the executing hardware core in contrast to shared variables that must be shipped across different cores, a much more costly operation.

An option to tag variables as *shared*, or even let the compiler find out variable-sharing all by itself might help. However, the question “shared among what” would come up immediately, and finding the answer by merely looking at an otherwise unstructured code image might not be possible at all.

Unfortunately, all this leads to an implementation of the parallel version of *MonteCarlo* along the lines of an explicit master/slave paradigm, thereby exhibiting ample artificial complexity that is clearly out of the skills of many not to say of most programmers :

```
Master(n) {
  BroadcastToSlaves(n/nofCores);
  masterCount = 0;
  for (i = 0; i < nofCores; i++) {
    Receive(count);
    masterCount += count;
  }
  return 4 * masterCount/n;
}
Slave(n) {
  myCount = MonteCarlo(n);
  Send(myCount, 0);
}
```

Admittedly, there is a hard core of complexity inherent in parallel computing that cannot be abstracted away but the aim of language designers and runtime developers should certainly be to move as much of it “behind the scenes”.

Our approach towards this aim is equally straightforward as it is generic: changing perspective. It is a paradigm shift from an imperative view of “operations operating on data op(data)” towards metaphorically “data acting on environment”. Note that the term “data” is used here as a collective term for any kind of data, be it elementary (such as int or float), structured (such as sets, arrays, matrices, vectors, dictionaries etc.) or any other kind of objects. However, with the aim of unification we will express all our “active data” as objects in an object-oriented Java-like language.

In addition to the conventional sections of objects such as local data, constructor, and methods we add a section *act { … }* declaring the object’s action, to be run as a *separate thread* (typically mapped onto a separate processing core by the runtime system) from beginning to the end without interruption. Regarding inheritance, *act-sections* are treated the same way as methods. In particular, they may be overwritten in extending classes.

Coming back to the *MonteCarlo* example, a parallel version on the basis of active data would look like this:

```
public class Sample {
  int x, y;
  int act { // run as a separate thread
    x, y = randr(-1, 1), randr(-1, 1);
    if (x*x + y*y <= 1) return 1: else return 0;
  }
```

```
}

S := set(Sample() for (int i = 0; i < n; i++)); //create and activate
nofInCircle := 0;
for (S, s) { nofInCircle += s; }
pi = 4 * nofInCircle / n;
```

You may ask what behind the scenes of *randr(-1, 1)* is and what its influence on concurrency is both from a correctness and from an efficiency point of view. We will come back to general method calls by self-activities later but for now just imagine that the call of *randr* simply means atomically picking up a random value that has been prepared by some magic demon.

Another, caricaturally simple example of an embarrassingly parallel style just for the purpose of illustrating the paradigm would be a concurrent version of computing the norm of a vector. Here, we obviously count on a "smart" runtime support optimizing matters as much as at all possible.

```
public class Square {
  float x;
  Square(x0) { x = x0; } // constructor
  float act { // running as a separate (micro)thread;
    return x * x
  }
}

D = vector(Square(a[i]) for (int i = 0; i < N; i++));
sum := 0;
for (D, d): sum += d;
norm := sqrt(sum);
```

In both of these examples we assume that a comprehensive *set* constructor is available that creates a set or vector of objects for the given range.

Synchronizing Parallel Processing

As we saw in the Monte-Carlo example with *randr*, self-activities in objects may themselves call methods. Obviously, there is a need to guarantee mutual exclusion within "critical sections" in such cases. As our effort is raising the level of abstraction with concurrent programming and amalgamating it with the object-oriented model, we simply map critical sections to method implementations and decorate these methods with a *synchronized* keyword.

Using the archetypal example of banking transactions for the purpose of illustration and modelling a corporate account with access by multiple users, we get the following code:

```
public class User { // class of self-active objects
  Account acc;

  User(Account a) { acc = a; } // constructor

  void act { // run as separate thread
    while true {
      // earn salary
      acc.Deposit(salary)
      // spot goods
      cash = acc.Withdraw(price) ;
      // buy goods for cash }
    }
  }
}
```

```
public class Account { // class of passive objects
  int bal;

  Account(int cash) { bal = cash; } // initializer

  public synchronized Deposit(int amount) { // synchronizing across object
    bal := bal + amount;
  }

  public synchronized int Withdraw(int amount) {
    if bal < amount then amount := bal;
    bal := bal - amount;
    return amount;
  }
}
```

Notice that synchronization extends across the entire object. In other words, the entire set of *synchronized methods* within the object runs under mutual exclusion. In particular, self-activities in objects need to access critical sections in the same object via explicit method call.

Just for the purpose of illustration: if the *Withdraw* method would not run synchronously under mutual exclusion the door for the famous *double-spending* problem would be wide open: Assuming an initial balance of $30'000.- and two users *A* and *B* each of them trying to withdraw $25'000.- from the account. Then, if fate wants it to happen:

1) *A* tries to withdraw $25'000.-: the *Withdraw* method computes *bal – amount* and returns $25'000.-
2) At roughly the same time and before the method has updated the *bal* variable, *B* also tries to withdraw $25'000.-: the *Withdraw* method again computes *bal - amount*, returns $25'000.- and leaves a final balance of $5'000.-.

Let us challenge our approach by another example typically to be found in textbooks on concurrent programming. The problem is to develop a concurrent program that reads lines from some input file, processes each of them individually, and writes the results to an output file.

```
public class inFile {
  File f;

  public inFile(name) { f = openFile(name); }
  public synchronized line read() { … // read line from f }

}

public class outFile { // synchronizing object
  File f;

  public outFile(name) { f = openFile(name); }
  public synchronized write(result) { … // write result to f }

}

public class Line { // class of self-active objects
  String line;
  int res;

  void act {
    line = myInFile.read()
    … // process line and compute result
    myOutFile.write(res);
  }

}
```

```
public class main { // main program is an active object too

  public main {
    myInFile, myOutFile = new inFile(…), new outFile(…);
    for i in maxLines { new Line(); }
  }

}
```

Our last example finally demonstrates how an elegant concurrent version of an originally sequential algorithm can be achieved via clever fabrication of the participating self-active objects . It is a concurrent version of the *Bellman-Ford* algorithm for finding the nearest distances from any given ground-zero node to all other nodes in some weighted graph. According to Wikipedia Bellman-Ford-Algorithmus – Wikipedia and assuming that *E* denotes the set of all edges in the graph, *e.start*, *e.end* and *e.weight* the starting node, ending node and weight respectively of edge *e*, *d[v]* the distance of node v from the ground-zero node number 0, *pred* the predecessor for any given node and *n* the number of vertices, the core of the algorithm reads like this:

```
for (i := 1, i < n, i++) {
  for (E, e) { // all edges
    if d[e.start] + e.weight < d[e.end] {
      d[e.end] := d[e.start] + e.weight;
      pred[e.end] := e.start
```

The question arises if the inner *for*-loop qualifies for concurrency. Obviously, *pred* and *d* are “shared variables” that have to be protected from “race conditions”.

The answer is “yes” but to make our approach work we need to reshuffle the *for*-loop by focusing on *pred* and mapping it to an array of self-active objects, something like this:

```
public class Pred { // predecessor class
  int me;
  public Pred(v) { me := v; } // initializer
  public void act { // behavior (re)activateable explicitly
    for (E, e) {
      if e.end = me { // my predecessor
        if d[e.start] + e.weight < d[e.end] {
          d[e.end] := d[e.start] + e.weight;
          pred[e.end] := e.start;
        }
      }
    }
  }
}

for (V, v): pred[v] := new Pred(v); // just create self-active objects
for (i := 1, i < n, i++) {
  for (V, v) { act pred[v] // activate or reactivate the object }
}
```

Conclusion

On the background of the availability of modern multicore hardware we argue that the support for concurrency in mainstream programming languages and runtime support systems is lagging behind both the hardware development and the general conceptual progression of programming paradigms. Concurrency support in mainstream languages both on the levels of « embarrassingly parallel » and

«lightweight concurrency» is either missing or trying to mimic some classical theoretical model from the 70s of the previous century, with mixed results mainly in terms of programmer-friendliness.

As a possible remedy for both levels “embarrassingly parallel code” and “objects and threads” we suggest shifting the paradigm from data and objects being operated on by some “remote” piece of code towards data and objects becoming self-active and taking full control of their entire life cycle, with the immediate benefit of disappearing unstructured “race conditions”. Obviously though for this model to fly additional support from all levels programming language, compiler and runtime support will be unavoidable.

References

[1] Cooperating Sequential Processes CSP, Technical Report, Edsger W. Dijkstra, EWD 123, 1965

[2] Tony Hoare, Communications of the ACM, Volume 21, Issue Nr. 8, August 1978,

Communicating Sequential Algorithms

Automated Verification of Fail-Free Declarative Programs

– Extended Abstract –

Michael Hanus

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

Abstract. Unintended failures during a computation are painful but frequent during software development. Failures due to external reasons (e.g., missing files, no permissions) can be caught by exception handlers. Programming failures, such as calling a partially defined operation with unintended arguments, are often not caught due to the assumption that the software is correct. This paper presents an approach to verify such assumptions. For this purpose, non-failure conditions for operations are inferred and then checked in all uses of partially defined operations. In the positive case, the absence of such failures is ensured. In the negative case, the programmer could adapt the program to handle possibly failing situations and check the program again. Our method is fully automatic and can be applied to larger declarative programs. The results of an implementation for functional logic Curry programs are presented.

1 Introduction

The occurrence of failures during a program execution is painful but still frequent when developing software systems. The two main reasons for such failures are

- external, i.e., outside the control of the program, like missing files or access rights, unexpected formats of external data, etc.
- internal, i.e., programming errors like calling a partially defined operation with unintended arguments.

External failures can be caught by exception handlers to avoid a crash of the entire software system. Internal failures are often not caught since they should not occur in a correct software system. However, in practice, they occur during software development and even in deployed systems which results in expensive debugging tasks. For instance, in imperative programs a typical internal failure is dereferencing a pointer variable whose current value is the null pointer (due to this often occurring failure, Tony Hoare called the introduction of null pointers his "billion dollare mistake"[1]).

[1] http://qconlondon.com/london-2009/speaker/Tony+Hoare

Although null pointer failures cannot occur in declarative programs, they might contain other typical programming errors, like failures due to incomplete pattern matching. For instance, consider the following operations (shown in Haskell syntax) which compute the first element and the tail of a list:

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

In a correct program, it must be ensured that `head` and `tail` are not evaluated on empty lists. If we are not sure about the data provided at run time, we can check the arguments of partial operations before the application. For instance, the following code snipped defines an operation to read a command together with some arguments from standard input (the operation `words` breaks a string into a list of words separated by white spaces):

```
readCommand = do
  putStr "Input a command:"
  s <- getLine
  let ws = words s
  case null ws of True  → readCommand
                  False → processCommand (head ws) (tail ws)
```

By using the predicate `null` to check the emptiness of a list, it is ensured `head` and `tail` are not applied to an empty list in the `False` branch of the case.

In this work we present a fully automatic tool which can verify the non-failure of this program. Our technique is based on analyzing the types of arguments and results of operations in order to ensure that partially defined operations are called with arguments of appropriate types. The principle idea to use type information for this purpose is not new. For instance, with *dependent types*, as in Agda [8], Coq [1], or Idris [2], or *refinement types*, as in LiquidHaskell [10,11], one can express restrictions on arguments of operations. Since one has to prove that these restrictions hold during the construction of programs, the development of such programs becomes harder [9]. Another alternative, proposed in [4], is to annotate operations with *non-fail conditions* and verify that these conditions hold at each call site by an external tool, e.g., an SMT solver [3]. In this way, the verification is fully automatic but requires user-defined annotations and, in some cases, also the verification of post-conditions or contracts to state properties about result values of operations [5].

The main idea of this work is to *infer* the non-fail conditions of operations. Since the inference of precise conditions is undecidable in general, we approximate them by using *abstract types* which are finite representations of sets of values. In particular, our methods performs the following steps:

1. We define a *call type* for each operation. If the actual arguments belong to the call type, the operation is reducible with some rule.
2. We define *in/out types* for each operation which approximate the input/output behavior of the operation.
3. For each call to an operation g occurring in a rule defining operation f, we check, by considering the call structure and in/out types, whether the call

type of g is satisfied. If this is not the case, the call type of f is refined and we repeat the checks with the refined call type.

At the end of this process, each operation has some correct call type which ensures that it does not fail on arguments belonging to its call type. Note that the call type might be empty on always failing operations. To avoid such situations, one can modify the program to encapsulate possibly failing computations so that a different action can be taken in case of a failure.

To sketch an example application of our method, consider the example above. Since in most cases declarative programs are defined by case distinctions on data constructors, we use as abstract types the set of top-level data constructors, where $\top$ denotes the set of all constructors. This domain is finite and can be order by set inclusion. For instance, the abstract type $\{\texttt{:}\}$ denotes all terms having the list constructor "`:`" at the top, i.e., all non-empty lists. Therefore, the call types of the operations `head` and `tail` can be characterized by the abstract type $\{\texttt{:}\}$. This call type is easy to derive from the left-hand sides of the rules defining `head` and `tail`. The call type states that, if the argument to `head` and `tail` is a non-empty list, the application is reducible.

Next we approximate the input/output behavior of operations by in/out types. These are basically disjunctions of abstract types for the input and the associated output. For instance, the operation `null` is defined by

```
null :: [a]  → Bool
null []    = True
null (_:_) = False
```

so that the in/out type of `null` is

$$\{\{\texttt{[]}\} \hookrightarrow \{\texttt{True}\},\ \{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\}$$

(where the disjunction is represented as a set). The in/out types can also be computed from the structure of the program with a fixpoint computation for recursive operations.

Now we want to verify the non-failure of `readCommand`. Since its definition contains calls to the partially defined operations `head` and `tail`, we have to show that the call types of these operations are satisfied at their call sites. This can be deduced by analyzing the case expression

```
case null ws of True   → readCommand
                False  → processCommand (head ws) (tail ws)
```

In the branch containing the calls to `head` and `tail`, we know that the result of `null ws` is `False`. From the in/out type of `null`, we can infer that this is only the case of the argument `ws` is a non-empty list. Thus, the call types of `head` and `tail` are satisfied by the argument `ws` at the call site.

In order to make our approach accessible to various declarative languages, we formulate and implement it in the declarative multi-paradigm language Curry [7]. Since Curry extends Haskell by logic programming features and there are also methods to transform logic programs into Curry programs [6], our approach can also be applied to purely functional or logic programs. A consequence of

using Curry is the fact that programs might compute with failures, i.e., it is not an immediate programming error to apply `head` and `tail` to possibly empty lists. However, subcomputations involving such possibly failing calls must be encapsulated so that it can be checked if such a computation has no result (this corresponds to exception handling in deterministic languages). If this is done, one can ensure that the overall computation does not fail even in the presence of encapsulated logic (non-deterministic) subcomputations.

References

1. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
2. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.
3. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer LNCS 4963, 2008.
4. M. Hanus. Verifying fail-free declarative programs. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming(PPDP 2018)*, pages 12:1–12:13. ACM Press, 2018.
5. M. Hanus. Combining static and dynamic contract checking for Curry. *Fundamenta Informaticae*, 173(4):285–314, 2020.
6. M. Hanus. From logic to functional logic programs. *Theory and Practice of Logic Programming*, 22(4):538–554, 2022.
7. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-lang.org, 2016.
8. U. Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International School on Advanced Functional Programming (AFP'08)*, pages 230–266. Springer LNCS 5832, 2008.
9. A. Stump. *Verified Functional Programming in Agda*. ACM and Morgan & Claypool, 2016.
10. N. Vazou, E.L. Seidel, and R. Jhala. LiquidHaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 39–51. ACM Press, 2014.
11. N. Vazou, E.L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 269–282. ACM Press, 2014.

An Extremely Flexible Programming Language

Christian Heinlein

Hochschule Aalen (University of Applied Sciences)
Aalen, Baden-Württemberg, Germany
christian.heinlein@hs-aalen.de

September 2023

Abstract. MOSTflexiPL is a general-purpose programming language, whose syntax can be freely extended and customized by every programmer. Based on a small set of predefined operators, it is possible to define new operators with arbitrary syntax, which do not only cover prefix, infix, and postfix operators, but also control structures, type constructors, and declaration forms. The paper gives an overview of major concepts of MOSTflexiPL in a tutorial-like manner by giving numerous examples from everyday programming.

1 Introduction

MOSTflexiPL, which is an acronym for **mo**dular, **s**tatically **t**yped, **fl**exibly **ex**tens**i**ble **p**rogramming **l**anguage, is a general-purpose programming language, whose syntax can be freely extended and customized by every programmer. The logo used in the paper title shall express the extreme flexibility provided by the language allowing even fancy constructions unimaginable with conventional languages. (Therefore, it might be advisable to forget almost all familiar and seemingly necessary limitations of other languages to be able to fully recognize MOSTflexiPL's capabilities.)

A basic principle enabling that flexibility is: Everything is an expression, i. e., the application of an operator to subexpressions, where operators might possess any number of names and operands in an arbitrary order. Apart from well-known prefix, infix, and postfix operators, this also includes "circumfix" operators such as `(•)` (an operand depicted by the bullet sign enclosed in parentheses), control structures such as `if•then•else•end`, declaration forms such as `•:•` (a name and a type separated by a colon), and so on. Another basic principle is, that the language provides only a small set of predefined operators covering arithmetic and logic operations as well as basic control structures, which can be used to define arbitrary new operators.

As the name indicates, the language is statically typed – which imposes numerous challenges with respect to the already mentioned flexibility –, and it is currently implemented by a compiler and a run-time system written in C++.

The primary goal of this paper is to give a broad overview of MOSTflexiPL's major concepts in a tutorial-like manner by showing numerous examples taken from every-

day programming. The examples also demonstrate, that writing syntactic extensions in MOSTflexiPL is just as easy as writing normal code, which is a significant difference and advantage over other approaches to syntactic extensibility.

2 Simple Operator Declarations

To give a first example, the following simple declarations define operators computing the square and the absolute value, respectively, of an integer value `x`, which can afterwards be applied using well-known mathematical syntax, e. g., 5^2 or $|2-7|^2$:

```
(x:int) "²" -> (int = x * x);
"|" (x:int) "|" -> (int = if x > 0 then x else -x end)
```

An operator declaration generally consists of a *signature*, an arrow and a *result declaration*, where the signature is a sequence of *names* and *parameter declarations*, while the result declaration consists of a *type* (the result type of the operator), an equality sign, and the *implementation* of the operator, enclosed in parentheses. A name is either a sequence of letters and digits starting with a letter (denoting exactly this sequence of characters) or a sequence of arbitrary characters enclosed in quotation marks (denoting this sequence of characters without the quotation marks). A parameter declaration consists of a name, a colon, and a type, enclosed in parentheses. Finally, the implementation and the types mentioned above are – according to the basic principle mentioned in Sec. 1 – expressions. (At the moment, types are atomic expressions such as `int` or `bool`, but see Sec. 4 and Sec. 7 for more complex type expressions.)

When an operator application such as $|2-7|^2$ is evaluated at run time, the parameters of the operator are initialized from left to right by recursively evaluating the corresponding operands and then the value of the expression is determined by evaluating the implementation of the operator.

In the examples above, the implementation of the square operator uses the predefined multiplication operator `•*•`, while the implementation of the abs operator uses the predefined change sign operator `-•` as well as the conditional operator `if•then•else•end` that returns, according to the truth value of its first operand, either the value of its second or its third operand.

The semicolon used to separate the two operator declarations is a simple predefined infix operator that evaluates its left and right operand and returns the value of the latter and therefore is typically used to denote sequential execution of subexpressions. But – again according to the basic principle mentioned in Sec. 1 – since declarations are expressions, too, the semicolon is also used to separate multiple declarations. (Sec. 8 explains the precise meaning of the result value of an operator declaration.) In contrast to many other programming languages, however, semicolon must not be used at the end of a sequence of subexpressions, because it is an infix operator.

To give another example, the following declaration defines an operator that recursively computes the factorial of an integer value `n`, which can also be applied using

well-known mathematical syntax, e. g., `5!` or 5^2`!`:

```
(n:int) "!" -> (int = if n <= 1 then 1 else (n-1)! * n end)
```

The particular challenge for the compiler with a declaration like this is to already recognize and accept the new syntax defined by the declaration inside of its own implementation to allow recursive applications of the operator.

MOSTflexiPL does not provide a predefined syntax for function declarations and applications, because any desired syntax is actually covered by the general operator declaration syntax mentioned above. For example, the style used by many procedural languages with function applications of the form `max(2, 3)`:

```
max "(" (x:int) "," (y:int) ")"
                          -> (int = if x > y then x else y end)
```

Or the syntax of functional languages such as Haskell with function applications of the form `max 2 3`:

```
max (x:int) (y:int) -> (int = if x > y then x else y end)
```

Or even a more natural-language-like flavour with function applications of the form `max of 2 and 3`:

```
max of (x:int) and (y:int)
                          -> (int = if x > y then x else y end)
```

3 Exclude Declarations

The predefined operators of MOSTflexiPL obey common rules for precedence and associativity, e. g., multiplication and division bind stronger than addition and subtraction, and all of them are left-associative. In contrast, user-defined operators have no predefined precedence or associativity, which frequently leads to ambiguous expressions. For example, the expression `2 + 3`^{2} might not only have the intended meaning `2 + (3`^{2}`)`, but could also be interpreted by the compiler as `(2 + 3)`^{2} (where the parentheses shall only indicate the different groupings).

Exclude declarations can be used to resolve such ambiguities by specifying interpretations of expressions which are not intended, i. e., excluded, for example:

```
excl (2 + 3)^2 end
```

Here, `2` and `3` are arbitrary placeholders for integer operands. The effect of this declaration is that applications of the operator •+• (the operator at the top of the parenthesized subexpression) are excluded as operands of the operator •^{2}. Therefore, the expression `2 + 3`^{2} will now be unambiguously interpreted as the addition of `2` and the square of `3`, because the alternative interpretation as the square of the addition of `2` and `3` is now forbidden.

Because outside of exclude declarations, the predefined parentheses `(•)` are just a normal operator (that simply returns the value of its operand), they can still be used

for explicit grouping: In the expression `(2 + 3)²` with explicit parentheses around the addition, the operand of the operator `•²` is not an application of the operator `•+•`, but rather an application of the operator `(•)` (whose operand is an application of the operator `•+•`), which is not forbidden.

In general, the expression between `excl` and `end` can contain any number of parenthesized subexpressions, each of which is interpreted as described above. Therefore, the effect of the following exclude declaration is, that the square operator binds stronger than all basic arithmetic operators:

```
excl (1 + 2)²; (1 - 2)²; (1 * 2)²; (1 / 2)² end
```

To give another example, the following exclude declarations encode exactly the rules for precedence and associativity of the basic arithmetic operators that have been mentioned at the beginning of this section:

```
excl (1+2)*(3+4); (1-2)*(3-4); (1+2)/(3+4); (1-2)/(3-4) end;
excl 1*(2*3); 1*(2/3); 1/(2*3); 1/(2/3) end;
excl 1+(2+3); 1+(2-3); 1-(2+3); 1-(2-3) end
```

4 Constants and Variables

A declaration of the form *name* `:` *type* `=` *init* declares a constant with the given name and type whose value is obtained by evaluating the initializer expression *init*, e. g., `N : int = 5²`. If the type is omitted, e. g., `N := 5²`, it is automatically deduced from the type of the initializer. If the initializer is omitted, the constant receives a unique new value that is different from every other value of the type. While this is of limited usefulness for numeric types such as `int`, it is crucial for variable types described below and for user-defined types described later in Sec. 6.

For any type `T`, the type `T?` denotes memory cells containing values of type `T`. Therefore, a declaration such as `x : T?` defines `x` as a constant referring to a unique new memory cell that contains a value of type `T`, i. e., `x` actually denotes variable with content type `T`. The current value contained in such a variable can be queried with the prefix question mark operator `?•`, and it can be changed with the assignment operator `•=!•`. The initial value of a variable is nil, which is a predefined value for any type that is different from every other value of the type. Because a variable with content type `T` is itself a value of type `T?`, it might itself be stored in a variable of type `T??`. If the content of such a variable is queried prior to any assignment to the variable, the returned value is the nil value of type `T?`. If the content of this nil variable is queried in turn, it will be the nil value of type `T`, and assigning any value to such a nil variable has no effect. (This behaviour is roughly comparable to reading and writing the Unix special file `/dev/null`.)

By using variables and the predefined loop operator `while•do•end`, the factorial operator mentioned in Sec. 2 can also be implemented in a more procedural style:

```
(n:int) "!" -> (int =
    f : int?; f =! 1;
    i : int?; i =! 2;
    while ?i <= n do
        f =! ?f * ?i;
        i =! ?i + 1
    end;
    ?f
)
```

In the implementation of the operator, the variables `f` and `i` are declared and assigned their initial values as described above, where `i` is used as a loop counter running from `2` to `n`, while `f` accumulates the factorial value that is finally returned.

5 Optional, Alternative, and Repeatable Syntax Parts

To provide even more syntactic flexibility, the signature of an operator declaration might also contain optional, alternative, and repeatable parts using well-known EBNF syntax.

For example, the following declaration defines a variadic maximum operator that can be applied to any number of operands, e. g., `max of 1`, `max of 1 and 2`, `max of 1 and 2 and 3`, and so on:

```
max of (x:int) { and (y:int) } -> (int =
    m : int?; m =! x;
    { if y > ?m then m =! y end };
    ?m
)
```

According to EBNF, the curly brackets in the signature indicate that an application of this operator might contain the word `and` followed by an operand corresponding to the parameter `y` any number of times (zero or more). To access the different values of this parameter in the implementation of the operator, a corresponding curly bracket operator `{•}` is provided there, whose operand is repeatedly evaluated for every value of `y`. For the particular application `max of 1 and 2 and 3` this means, that the variable `m` declared in the implementation is initialized with the value of `x` (i. e., `1`), and then the `if` expression inside the curly brackets is evaluated in turn for `y` equal to `2` and to `3`, changing the value of the variable `m` to `2` and to `3`, respectively. Finally, the resulting value of `m` is returned.

To give another example, the following operator performs arbitrary calculations consisting of additions and subtractions, e. g., `calc minus 1 plus 2` or `calc 1 minus 2 plus 3`:

```
calc [minus] (x:int) { (plus|minus) (y:int) } -> (int =
    res : int?;
    res =! [-x | x];
```

```
    { res =! (?res + y | ?res - y) };
    ?res
)
```

In addition to the curly brackets denoting a repeatable part, the signature of this operator also contains square brackets denoting an optional part as well as round brackets containing two or more alternative parts separated by vertical bars. To find out in the implementation of the operator, whether the optional word `minus` after the word `calc` is present or not in a particular application of the operator, a corresponding square bracket operator `[•|•]` is provided, whose first or second operand, respectively, is evaluated accordingly. Similarly, a round bracket operator `(•|•)` with two operands corresponding to the round brackets with two alternatives in the signature is provided, whose first or second operand is evaluated according to whether the first or second alternative has been chosen in a particular application of the operator or – because in this example the round brackets are nested inside the curly brackets – in the respective pass through the curly brackets. The result value of a square or round bracket operator is the value of the operand that has been evaluated, while the result value of a curly bracket operator is the number of passes through these brackets. Taken together, these bracket operators allow the implementation of the operator to exactly determine the structure a particular operator application and to process the values of its operands in a rather concise manner.

Generally speaking, all three kinds of brackets can have any number of alternatives, except that round brackets must contain at least two, because round brackets with just one alternative are useless. Therefore, the corresponding bracket operators provided in the implementation of the operator have a corresponding number of operands separated by vertical bars, where the i-th operand is evaluated if the i-th alternative has been chosen in a particular operator application or pass through curly brackets. As an exception, an operator corresponding to square brackets has an additional optional operand, that is evaluated (if it is present) if none of the alternatives has been chosen.

Therefore, the `calc` operator could also be defined as follows:

```
calc [minus] (x:int) { plus (y:int) | minus (z:int) } -> (int =
    res : int?;
    res =! [-x | x];
    { res =! ?res + y | res =! ?res - z };
    ?res
)
```

The different kinds of brackets can be used any number of times in an operator signature, and they can be arbitrarily nested to describe rather complex syntactic constructs, for example:

```
dnf [na: not] (a:bool) { and [nb: not] (b:bool) }
    { or [nc: not] (c:bool) { and [nd: not] (d:bool) } } end
                                                    -> (bool =
    res : bool?;
```

```
        res =! [na: ~a | a];
        { res =! ?res & [nb: ~b | b] };
        {
            tmp : bool?;
            tmp =! [nc: ~c | c];
            { tmp =! ?tmp & [nd: ~d | d] };
            res =! ?res | ?tmp
        };
        ?res
    )
```

This operator can be used to express arbitrary logic expressions in disjunctive normal form (DNF), e. g., `dnf x or not u and v and w or y and not z end`, if `u`, ..., `z` denote values of type `bool`. The predefined operators of MOSTflexiPL for logic operations used in the implementation of the `dnf` operator are `~`• for negation, •`&`• for conjunction, and •`|`• for disjunction. To disambiguate the multiple square brackets in the signature and their corresponding bracket operators in the implementation, they are labeled with unique names and a colon after the opening bracket. While the curly brackets could be disambiguated in the same way, this is actually not necessary, because parameters defined inside of particular curly brackets are only visible in the operands of the corresponding bracket operator. Therefore, the bracket operator used first in the implementation must correspond to the first curly brackets in the signature, because parameter `b` is only visible in the operator corresponding to these brackets. For the same reason, the (outer) bracket operator used next in the implementation must correspond to the second (outer) brackets in the signature (due to the visibility of parameter `c`), while the bracket operator used inside of the former must correspond to the inner brackets in the signature (due to the visibility of parameter `d`).

6 Static Operators and User-Defined Data Structures

As already mentioned in Sec. 2, a constant declaration *`name : type`* defines a constant with the given name and type and a unique new value. Constants whose type is the predefined meta-type `type` denote unique new types, e. g.:

```
Color : type
```

Afterwards, any number of unique values or "objects" of such a type can also be defined as constants, e. g.:

```
red : Color;
blue : Color;
green : Color
```

As also mentioned in Sec. 2, constants of a variable type `T?` actually denote unique variables with content type `T`.

Additionaly, if the implementation of an operator (including the equality sign) is omitted, a unique new value of the result type is returned whenever the implementation would be evaluated.

If the arrow `->` in an operator declaration is replaced by a double arrow `=>`, the declared operator is a so-called *static operator*. In contrast to a normal operator defined with a single arrow (that is also called a *dynamic operator*), a static operator has a runtime memory to store the parameter and result values of all applications of the operator performed so far. If the parameter values of a particular application are equal to the corresponding values of an earlier application, the implementation of the operator is not evaluated again, but the result value stored from the earlier application is returned instead. Therefore, a static operator guarantees that applications to the same parameter values always return the same value.

While this could be used to automatically optimize operators with runtime-intensive implementations by means of memoization, this is in fact neither the primary goal nor the typical use of static operators. Instead, they can be used as follows to flexibly define data structures:

```
Point : type;
(p:Point) "@" x => (int?);
(p:Point) "@" y => (int?);

p1 : Point; p1@x =! 1; p1@y =! 2;
p2 : Point; p2@x =! 3; p2@y =! 4;

dx := ?p1@x - ?p2@x;
dy := ?p1@y - ?p2@y
```

According to the semantics of static operators and omitted operator implementations described above, the operator `•@x` (and likewise the operator `•@y`) guarantees, that applications to the same point object always return the same `int` variable, while applications to different point objects return different variables (that are also different from all other existing variables). Therefore, the variables returned by `p1@x` and `p1@y` can be used to store the `x` and `y` coordinates of `p1`, while the variables returned by `p2@x` and `p2@y` can be used to store the coordinates of `p2`. Therefore, the constants `dx` and `dy` defined at the end of the example denote the difference of the `x` and `y` coordinates, respectively, of the points `p1` and `p2` (i. e., both have the value `-2`).

Because it is always possible to add further operators like `•@x` and `•@y` later on, i. e., to modularly extend a type such as `Point` with new "attributes," these types are called *open types* [5]. Furthermore, it is possible that different objects of a type possess values for different subsets of attributes. According to the semantics of variables described in Sec. 4, querying the value of a missing attribute of an object simply returns nil.

7 Generic Operators

If an optional parameter appears in the type of another parameter of the same operator, its value can be automatically deduced from the type of the operand corresponding to the other parameter, and therefore, the former parameter is called a *deducible parameter*. This can be used to define generic operators similar to C++ templates and Java generics, for example:

```
[(T:type)] (x:T?) "<->" (y:T?) -> (T? =
    z := ?x; x =! ?y; y =! z; y
);

excl u : int?; u <-> (u <-> u) end;

v1 : int?; v1 =! 1;
v2 : int?; v2 =! 2;
v1 <-> v2
```

Because `v1` and `v2` both have type `int?`, `v1 <-> v2` is a correct application of the previously defined swap operator, where the parameters `x` and `y` are initialized with the explicit operands `v1` and `v2`, respectively, while the optional parameter `T` is implicitly initialized with the type `int` causing the type `int?` of the operands `v1` and `v2` to match the type `T?` of the corresponding parameters `x` and `y`. The implementation of this operator swaps the values contained in the variables `x` and `y` and returns the variable `y`. (The latter enables concatenated applications of the operator to a sequence of variables, e. g., `v1 <-> v2 <-> v3`, actually performing a leftward rotation of their values. To disambiguate expressions like that, the exclude declaration is necessary which makes the operator left-associative.)

Generic operators can also be used to generalize the idea of open types and attributes already introduced in Sec. 6:

```
(U:type) "-->" (V:type) => (type);
[(U:type) (V:type)] (u:U) "@" (a:U-->V) => (V?);

Point : type;
x : Point --> int;
y : Point --> int;

p1 : Point; p1@x =! 1; p1@y =! 2;
p2 : Point; p2@x =! 3; p2@y =! 4
```

For every pair of types `U` and `V`, `U-->V` is a unique type intended to represent attributes for type `U` with target type `V`. Therefore, the constants `x` and `y` represent attributes for type `Point` with target type `int`. Furthermore, for every combination of an object `u` of some type `U` and an attribute `a` for type `U` with some target type `V`, `u@a` is a unique variable with content type `V` that can be used to store the value of attribute `a` for object `u`. Therefore, expressions such as `p1@x` and `p2@y` have exactly

the same meaning as in Sec. 6, but the definition of the attributes `x` and `y` is much more convenient now if the operators `•-->•` and `•@•` are provided by a library.

To make things even more convenient, another operator `•.•` can be defined to simplify the querying of attribute values by allowing to write, e. g., `p1.x` instead of `?p1@x`:

```
[(U:type) (V:type)] (u:U) "." (a:U-->V) -> (V = ?u@a)
```

Furthermore, it is possible to define a more advanced generic operator to directly construct objects of an open type with a set of initial attribute values:

```
(U:type) "(" [(V1:type)] (a1:U-->V1) "=" (v1:V1)
        { "," [(V2:type)] (a2:U-->V2) "=" (v2:V2) } ")" -> (U =
    u : U;
    u@a1 =! v1;
    { u@a2 =! v2 };
    u
);

p1 := Point(x = 1, y = 2);
p2 := Point(x = 3, y = 4)
```

As a final improvement, the operator `•@•` can be redefined as follows to make it return nil instead of a unique new variable if either the object `u` or the attribute `a` is nil (note that a constant declaration returns the value of the constant and an `if` expression without an `else` part returns nil if the condition is not satisfied; the predefined operator `•=/•` tests for inequality):

```
[(U:type) (V:type)] (u:U) "@" (a:U-->V) => (V? =
    if u =/ nil & a =/ nil then
        v : V?
    end
)
```

The effect of this modification is that querying an attribute value of a nil object or the value of a nil attribute of any object always returns nil (because querying a nil variable returns nil) and that modifying such attributes has no effect (because modifying a nil variable has no effect). Without this modification, it would be possible to accidentally assign a value to an attribute of a nil object, that would afterwards be returned by querying this attribute of the nil object, for example:

```
Line : type;
beg : Line --> Point;
end : Line --> Point;

ln := Line(beg = Point(x = 1, y = 2));
ln.end@x =! 4;
ln.end.x
```

Because the `Line` object `ln` does not have a value for the attribute `end`, `ln.end` returns a nil point, whose attribute `x` is then (formally) assigned the value `4`. With the

original definition of the operator •@• given at the beginning of this section, `ln.end@x` would return a "real" variable that would store the assigned value `4`. Therefore, `ln.end.x` would return this value, which is illogical because the line `ln` does not have a defined endpoint at all. With the modified definition of the operator •@• given above, `ln.end@x` returns a nil variable causing the assignment of the value `4` to have no effect, and therefore, `ln.end.x` also returns nil which is more logical.

The above operators can also be used to define generic open types such as lists using a Haskell-like syntax `[|T|]` to denote lists with element type `T`:

```
"[|" (T:type) "|]" => (type);
[(T:type)] head => ([|T|] --> T);
[(T:type)] tail => ([|T|] --> [|T|])
```

Note that the generic attributes `head` and `tail` are not defined as constants, because constants cannot have any parameters and therefore cannot be generic, but rather as static operators that return different values for different types `T` to make sure that list types with different element types have logically different attributes.

Furthermore, `head` and `tail` are remarkable because the value of their type parameter `T` cannot be deduced from their bare application (which is simply `head` or `tail`), but only from the context of such an application, e. g.:

```
ls1 : [|int|]; ls1@head =! 1;
ls2 : [|bool|]; ls2@head =! true
```

Because (i) the type of `ls1` is `[|int|]`, (ii) the operand types of the operator •@• must be `U` and `U-->V` for suitable types `U` and `V`, and (iii) the type of `head` must be `[|T|]-->T` for some suitable type `T`, the type of `head` in the expression `ls1@head` is uniquely deduced by the compiler as `[|int|]-->int` by solving this "constraint puzzle." Likewise, the type of `head` in the expression `ls2@head` will be `[|bool|]-->bool`.

Using the above definitions, the "cons" operator for constructing a list from a head element `h` and a tail list `t` can be defined as follows, again with a syntax •:• known from Haskell:

```
[(T:type)] (h:T) ":" [(t:[|T|])] -> ([|T|] =
    [|T|](head = h, tail = t)
);

excl (ls := 1) : end;

ls1 := 1 : 2 : 3 :;
ls2 := true : false :
```

As a convenient extension, the tail list can be omitted, because the parameter `t` is optional and therefore is automatically nil if the corresponding operand is missing in an application of the operator.

Note that it is not necessary to explicitly make the operator •:• right-associative by means of an exclude declaration, because an expression such as `1:2:3:` can only be interpreted as `1:(2:(3:))`, because any other syntactically possible interpretation (e. g., `((1:2):3):`) would not be type-correct. And in fact, the MOSTflexiPL compiler – in contrast to the compilers of many other programming languages – does not artificially separate the semantic analysis (i. e., type checking) from the syntactic analysis of the source code, but rather performs them together in close cooperation in order to rule out expressions which are not type-correct as early as possible.

On the other hand, the exclude declaration contained in the example is necessary to exclude an interpretation such as `(ls1:=1):2:3:`, which would in fact be type-correct.

8 Implicit Parameters

Another useful example of generic operators would be a generic maximum operator that can be applied to operands of any type `T`:

```
[(T:type)] max of (x:T) and (y:T) -> (T =
    if x > y then x else y end
);

max of 1 and 2;
max of p1 and p2
```

While the application of this operator to the `int` values `1` and `2` appears reasonable, its application to the points `p1` and `p2` does not make sense, because there is no operator •>• to compare points. And in fact, the compiler would already reject the above declaration of the maximum operator and not only its application to points, because there is no operator •>• that can be applied to the operands `x` and `y` of an arbitrary type `T` whose precise value is not known there.

To make the compiler accept the operator declaration, it is necessary to express that the operator might only be applied to operands of a type `T`, if there is an operator •>• which accepts two operands of that type `T` and which returns a value of type `bool`. This can be expressed with an *implicit parameter*:

```
[(T:type)] max of (x:T) and (y:T) [(+ (T) ">" (T) -> (bool))]
-> (T = if x > y then x else y end)
```

This requires explanations of some details which have not been mentioned yet:

- The name of a parameter including the subsequent colon can be omitted if it is not needed.
  Therefore, `(T) ">" (T) -> (bool)` is a correct operator declaration describing exactly the kind of operator that is required by the maximum operator.
- Furthermore, an operator declaration actually constitutes a type, i. e., the type of the declared operator, which includes the types of its parameters and its result as well as

its syntax.
Therefore, `((T) ">" (T) -> (bool))` is a correct declaration of an anonymous parameter whose type is the operator type `(T) ">" (T) -> (bool)`.

- Prefixing this parameter type with a plus sign marks the parameter as an implicit parameter of the maximum operator, which means that it is implicitly bound to an operator of the same type that is visible at the point where the maximum operator is applied. (Alternatively, it would be possible to pass an explicit operand of that type, cf. Sec. 9.)
  Therefore, applications of the parameter in the implementation of the maximum operator (i. e., `x > y`) will actually be forwarded to the operator that has been passed (either implicitly or explicitly).

For an application such as `max of 1 and 2` with operands of type `int` that means, that an operator with type `(int) ">" (int) -> (bool)`, i. e., the predefined •>• operator for integer values, is implicitly passed to the maximum operator and thus used in its implementation to compare the operands.

An application such as `max of p1 and p2` with operands of type `Point`, however, is rejected by the compiler, because there is no operator with type `(Point) ">" (Point) -> (bool)` that could be passed implicitly. This could be remedied, however, by defining such an operator before applications of the maximum operator to points:

```
(p1:Point) ">" (p2:Point) -> (bool = p1.x > p2.x)
```

Here, `p1` is considered greater than `p2` if the x coordinate of `p1` is greater than that of `p2`.

In fact, the operator that is implicitly passed for an implicit parameter is not required to have exactly the same type as the parameter. It is rather sufficient, that the parameter *can be replaced* by the operator according to the following definition: An operator or parameter can be replaced by another operator or parameter, if every correct application of the former is also a correct application of the latter with the same type.

For example, there is actually no predefined operator •>•, but rather a much more general comparison operator that also supports comparison chains of multiple operands such as `a > b = c >= d` with well-known semantics from mathematics. (In contrast to mathematical practice, however, it is even allowed to form "inconsistent" chains such as `a > b <= c`.) But because this operator can replace the implicit parameter according to the definition above, it can and will in fact be passed implicitly to applications of the maximum operator to integer operands.

Another implication of this replacement rule is, that an operator that has implicit parameters itself can be implicitly passed to an implicit parameter, if there are in turn matching operators for its own implicit parameters, and so on. For example:

```
[(T:type)] (x:T) "²" [((T) "*" (T) -> (T))] -> (T = x * x);
[(T:type)] (x:T) "⁴" [((T) "²" -> (T))] -> (T = x²²)
```

An application of the bisquare operator •⁴ to an integer value such as 5⁴ requires a square operator •² for integers, which in turn requires a multiplication operator •*• for integers, which is available as a predefined operator. Therefore, the expression 5⁴ is correct.

On the other hand, an application of the bisquare operator to a point would require a square operator for points, which would be available if there would be a multiplication operator for points, which is not the case, however. Therefore, an expression such as p1⁴ would be rejected by the compiler. Again, this could be remedied in principle by defining such a multiplication operator.

9 Higher-Order Operators

Operators with implicit parameters as described in the previous section are actually higher-oder operators, i. e., operators having parameters that are itself operators. This is not restricted to implicit parameters, however, but operators can also be passed explicitly to other operators.

To give a typical example from functional programming:

```
[(X:type) (Y:type)]
map (f: f (X) -> (Y)) (ls: [|X|]) -> ([|Y|] =
    if ls =/ nil then
        (f ls.head) : (map f (ls.tail))
    end
);

sq: (x:int) "²" -> (int = x * x);

sq: f (x:int) -> (int = x * x);
ls := (1 : 2 : 3 :);
map sq ls
```

According to the explanations given in Sec. 8, `(f: f (X) -> (Y))` is the declaration of a parameter with name `f` whose type is the operator type `f (X) -> (Y)`, i. e., `f` is used both as the name of the entire parameter and as the first name of the operator contained in its type. Therefore, `f ls.head` is an application of this operator to `ls.head`, while the `f` in `map f ls.tail` is used to pass this operator to the recursive invocation of `map`.

According to the same principle, `sq` is a constant whose type is the operator type given after the colon of the constant declaration and whose value is the operator contained in that type. That is, in fact, an exception to the rule given in Sec. 4, which must now be restated as follows: If the initializer in a constant declaration is omitted, the value of the constant is either the operator contained in its type – if this type is an operator type – or otherwise a unique new value as stated before. Therefore, `sq` can actually be used to refer to the square operator and to pass it to the `map` operator.

When an operator is passed explicitly, the requirement given in Sec. 8, that the operator must be able to replace the corresponding parameter, is relaxed in order to allow operators whose syntax is different from the syntax of the parameter as in the above example. (The operator's syntax is $\bullet^2$, while the parameter's syntax is `f•`.) Instead, only the parameter and result types of the operator and the parameter must match, i. e., their names are completely ignored because they are not important. (The precise rules, which are currently developed in detail, are more complex, because if an operator has optional, alternative, or repeatable parts, at least some of its names might be important to disambiguate applications of this operator.)

Finally, it is also possible to return operators from other operators, for example:

```
add (y:int) -> (f (int) -> (int) =
    f: f (x:int) -> (int = x + y)
);

map (add 5) ls
```

Because the result type of the operator `add•` is an operator type `f (int) -> (int)`, its implementation must return an operator of that type. And because the type of the constant `f` is also an operator type – in fact, exactly the same operator type –, the value of this constant is, according to the restated rule above, the operator contained in that type. Finally, because a constant declaration returns the value of the constant, the implementation of the operator `add•` returns exactly this operator, which can then be passed, e. g., to an application of the map operator. Please note, that it is in fact necessary to declare that dummy constant (with an arbitrary name), because the operator declaration itself does not return the declared operator, but rather its type (cf. Sec. 8).

10 Outlook

The language MOSTflexiPL and its compiler are still under active development, and several useful features that have already been developed and prototypically implemented in older versions of the compiler, have not been integrated into the current compiler, including:

- "Call by expression" parameters, which are required to define control structures such as branches and loops, whose operands shall be evaluated conditionally or repeatedly.
- Import and export declarations, which are required to define user-defined scoping rules and locally confined syntax extensions.
- Virtual operators, which are required to define type aliases to abbreviate or abstract from complex types and to define declaration operators, i. e., to be able to extend even the syntax that is used to define new syntax.
- Dynamic redefinitions of operators [4], which allow amongst other things strictly modular extensions of existing code and thus support unanticipated software evolution.

- Basic operators for parallel execution and synchronization, which can be used to define more convenient and advanced constructs for parallel programming.
- User-defined literals, e. g., for types representing date and time values.
- User-defined whitespace and comments to allow any desired syntax for both block and line comments.
- More descriptive compiler messages in case of errors and ambiguities.
- Meta-operators, which are required to pass the values of a repeatable parameter to another operator accepting repeatable parameters.
  This is in fact a completely new idea that is still under development and has not been implemented in any of the compiler versions yet.

A feature that is already implemented in the current version of compiler, but has not been described in this paper, is type deduction: Basically, it is possible to omit almost all types from declarations as long as they can be deduced by the compiler, i. e., the types of constants (this has in fact been mentioned in Sec. 4), parameters, and results of operators.

11 Related Work

During the history of programming language development, the idea of an extensible programming language has appeared every now and then.

One of oldest and most well-known examples is Lisp [9] with its different dialects and flavors. Similar to MOSTflexiPL, Lisp does neither distinguish between operators and functions nor between predefined and user-defined operators/functions. By defining new functions – or macros, whose syntactic appearance is identical to that of functions – a programmer is actually extending the language all the time. Another parallel to MOSTflexiPL is the fact that language extensions are defined in the language itself, and that a very small language core is sufficient for that purpose. However, there are also essential differences: First of all, Lisp does not possess a static type system. Furthermore, Lisp expressions must always be parenthesized, which significantly restricts the possibilities for defining new syntax. Finally, MOSTflexiPL does not have a "procedural" macro engine, i. e., no user code will be executed at compile time in order to perform syntactic transformations. In summary, MOSTflexiPL has considerable advantages over Lisp (complete syntactic freedom and static type safety), while the deliberately omitted procedural macro facility has not been perceived as a major limitation yet.

Dylan [3] is a more modern language that has been strongly influenced by Lisp's ideas. It also supports syntactic extensibility in the language itself (actually in a rewrite macro system which is an integral part of the language). Even though the programmer has more freedom than with Lisp's simple s-expressions, there are also strict syntactic limitations which cannot be exceeded. In contrast, the operator concept of

MOSTflexiPL offers virtually unlimited syntactic freedom. Apart from that, Dylan does not have a static type system either.

Many different languages, e. g., Haskell [7], Prolog [2], and Scala [8], allow the user to extend at least the syntax of expressions by defining new operator symbols. Since functional languages, just as MOSTflexiPL, do not distinguish between expressions and statements, the syntax of statements (e. g., control structures) becomes also extensible in principle. However, the syntax of types and declarations still remains fixed. In MOSTflexiPL, however, the basic principle "*everything* is an expression" implies that *all* parts of the language can be extended simply by defining new operators.

An approach whose basic ideas and objectives are almost identical to that of MOSTflexiPL is "π – a Pattern Language" [6]. The concept called pattern there – which is "the only language construct in π" – directly corresponds to an operator in MOSTflexiPL: It possesses a syntax, composed of names (or symbols) and placeholders for operands, and an associated meaning corresponding to the implementation of a MOSTflexiPL operator. Thus, both approaches provide the same virtually unlimited syntactic flexibility that ultimately stems from the lack of any predefined grammar.

A significant difference and advantage of MOSTflexiPL over π is once again the static type system, since π is completely dynamically typed. In fact, the endeavour to reconcile extreme flexibility on the one hand with a maximum of static checkability on the other hand has been and still is the most ambitious challenge in the development of MOSTflexiPL.

Apart from that, MOSTflexiPL provides several other useful facilities not found in π, e. g., implicit and deducible parameters (where the latter are dispensable in a dynamically typed language) or import, export, and exclude declarations which allow, amongst others, user-defined scoping rules and locally confined syntax extensions.

Finally, MOSTflexiPL might also be considered an adaptive grammar formalism [1, 10]. Because "everything is an expression," there is a single non-terminal symbol `X` denoting expressions. Every operator declaration induces a new production for `X` whose right hand side can be derived from the operator's signature by treating the operator's names as terminal symbols and replacing explicit parameters with the non-terminal `X`. The type information associated with the parameters and the result type of the operator can be added as grammar attributes. Import and export declarations control the set of currently active productions, while exclude declarations can be used to rule out some otherwise possible derivations.

12 Conclusion

MOSTflexiPL is a programming language currently under development whose syntax can be extended and customized by its users in a virtually unlimited way, where a rather small number of core constructs is sufficient to support a broad range of different programming styles. Therefore, it can be used, amongst others, as an extensible general purpose programming language, but also as a host language for developing

domain-specific languages. It possesses a static type system and is implemented by a compiler and a run-time system written in C++.

Acknowledgements

I want to thank all the students that have contributed to the development of MOST-flexiPL with their master's thesis, bachelor's thesis, or student's project (in chronological order): Marco Perazzo, Stefan Billet, Miriam Klement, Frank-Stephan Schierle, Rainer Ißler, Sascha Simon, Christian Homeyer, Dominik Biener, Tobias Sachon, Maximilian Blenk, Jakob Loskan, Niklas Brendle, Lukas Neubauer, David Sugar, Lukas Pietzschmann.

I also want to thank the colleagues at the Compilers and Languages Group of Jens Knoop at TU Wien for many fruitful discussions during my last sabbatical.

Finally, I want to thank my father in heaven for every challenge that I have been able to master with him.

References

[1] H. Christiansen: "A survey of adaptable grammars." *ACM SIGPLAN Notices* 25 (11) November 1990, 35–44.

[2] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[3] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[4] C. Heinlein: "Global and Local Virtual Functions in C++." *Journal of Object Technology* 4 (10) December 2005, 71–93, http://www.jot.fm/issues/issue_2005_12/article4.

[5] C. Heinlein: "Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance." *Journal of Object Technology* 6 (3) March/April 2007, 101–151, http://www.jot.fm/issues/issue_2007_03/article3.

[6] R. Knöll, M. Mezini: "$\pi-$ a Pattern Language." In: *Proc. 24th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)* (Orlando, FL, October 2009). *ACM SIGPLAN Notices* 44 (10) October 2009, 503–521.

[7] S. Marlow (ed.): *Haskell 2010 Language Report*. HaskellWiki, 2010.
http://haskell.org/definition/haskell2010.pdf

[8] M. Odersky: *The Scala Language Specification* (Version 2.9). Programming Methods Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), May 2011.

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

[10] Wikipedia Contributors: *Adaptive Grammar*.
https://en.wikipedia.org/wiki/Adaptive_grammar (2023-08-30)

Untersuchung der Domänenabhängigkeit tiefer Lernverfahren für die Typinferenz in Python

Bernd Gruner[1], Tim Sonnekalb[1], Thomas S. Heinze[2] und Clemens-Alexander Brust[1]

[1] DLR-Institut für Datenwissenschaften, Mälzerstraße 3-5, 07747 Jena
{bernd.gruner,tim.sonnekalb,clemens-alexander.brust}@dlr.de
[2] Duale Hochschule Gera-Eisenach, Weg der Freundschaft 4, 07546 Gera
thomas.heinze@dhge.de

Zusammenfassung. In diesem Beitrag stellen wir unsere in [2] veröffentlichten Arbeiten zur Untersuchung der Domänenabhängigkeit von tiefen Lernverfahren für die Typinferenz vor. Wir zeigen am Beispiel Type4Py und Python, dass ein Wechsel zwischen den Anwendungsdomänen Webprogrammierung und wissenschaftliches Rechnen zu einer Verschlechterung in der Vorhersagequalität führt.

1 Einführung

Moderne dynamisch typisierte Programmiersprachen erlauben durch Spracherweiterungen wie *PEP-484* für Python und TypeScript für JavaScript optionale Typannotationen zu nutzen und damit auf die Vorteile der statischen Typisierung zurückzugreifen. In Verbindung mit einer Typinferenz zur automatisierten Ableitung von Typannotationen, ergibt sich ein hilfreiches Werkzeug um Schnittstellen zu dokumentieren, Programmanalysen zu unterstützen und nicht zuletzt typbezogenen Laufzeitfehlern vorzubeugen. Klassische Verfahren der Typinferenz basieren auf statischen oder dynamischen Programmanalysen, jeweils einhergehend mit einschlägigen Problemen: mangelnde Präzision aufgrund fehlender Typeinschränkungen dynamischer Sprachen und der in statischen Analysen angewandten Abstraktionen oder der unvollständigen Abdeckung im Fall dynamischer Analysen. Methoden zur Typinferenz auf Grundlage tiefer Lernverfahren bieten einen aktuellen, alternativen Ansatz der vielversprechende Ergebnisse liefert.

Der große Teil der zur Typinferenz mittels tiefer Lernverfahren veröffentlichten Forschungsarbeiten betrachtet dynamische Programmiersprachen. Für die Sprache Python, die auch im Zentrum unseres Beitrags steht, wurden eine Reihe von Verfahren vorgeschlagen, so auch *Type4Py* [4]. In Type4Py werden zwei rekurrente neuronale Netze mit über 500.000 Programmbeispielen aus ca. 200.000 Python-Dateien öffentlicher Repositorien trainiert, um die charakteristischen Muster von Typannotationen anhand von auftretenden Bezeichnern und Programmstrukturen zu lernen. Beide Netze definieren dann in einem Modell einen Vektorraum, in dem ähnliche Typen zusammengehörige Gruppen bilden und sich die Typannotationen zu ungesehenen Programmbeispielen mit einer Ähnlichkeitssuche vorhersagen lassen. Die Autoren von Type4Py konnten zeigen, dass ihr Verfahren im Vergleich mit verwandten Ansätzen bessere Ergebnisse bei der Typvorhersage liefert [4].

Tabelle 1. Datensatz *CrossDomainTypes4Py* (Aufteilung in 70%, 10% und 20% Trainings-, Validierungs- und Evaluationsdaten; seltene Typen mit <100 Vorkommen)

	Webprogrammierung			Wissenschaftliches Rechnen		
Repositorien	3.129			4.783		
Python-Dateien	166.505			470.011		
	Train.	Valid.	Eval.	Train.	Valid.	Eval.
Programmbeispiele	251.064	27.987	61.978	476.768	56.854	148.732
Typen, darunter	7.588	1.195	8.475	14.973	2.218	14.960
... häufige Typen	232	158	192	363	252	332
... seltene Typen	7.356	1.037	8.283	14.610	1.966	14.628

2 Domänen und Datensatz *CrossDomainTypes4Py*

Die Anwendung und Entwicklung von Lernverfahren, so auch für die Typinferenz, erfolgt im Allgemeinen unter der Annahme, dass sich die statistischen Verteilungen in den Merkmalen der für das Training genutzten Daten und der vorherzusagenden Daten nicht unterscheiden. In der Praxis können aber Probleme bei Verletzung dieser unter dem Schlagwort *independent and identically distributed (iid)* bekannten Annahme auftreten. In diesem Beitrag untersuchen wir die Verteilungen der für die Typinferenz mit Type4Py relevanten Programmmerkmale unter Berücksichtigung unterschiedlicher Anwendungsdomänen. Im Fokus steht die Frage, inwiefern sich das auf einer Domäne erlernte Modell auch für Vorhersagen auf einer anderen Domäne nutzen lässt. Type4Py wurde dabei als ein aktueller Repräsentant tiefer Lernverfahren für die Typinferenz ausgewählt.

Um diese Frage zu beantworten, wird zunächst ein entsprechender Datensatz benötigt, der neben einer großen Zahl an Programmbeispielen und Typannotationen auch Informationen zu den zugehörigen Domänen bereitstellt. Vergleichbare Datensätze, wie der ursprünglich zum Training von Type4Py genutzte *ManyTypes4Py*-Datensatz [3], enthalten keine Informationen zu Anwendungsdomänen. Mittels Repository Mining durchsuchen wir zu diesem Zweck systematisch öffentliche Repositorien nach Python-Dateien, die einerseits eine Abhängigkeit zur Programmbibliothek *mypy* aufweisen, da wir in diesem Fall von vorhandenen Typannotationen ausgehen können. Andererseits filtern wir zusätzlich nach Abhängigkeiten zu den Bibliotheken *Flask* und *NumPy*. Diese zwei Bibliotheken sehen wir jeweils für die Domänen Webprogrammierung und wissenschaftliches Rechnen als kennzeichnend. Anschließend werden Duplikate entfernt, dies betrifft zum einen Doppelungen aufgrund von gleichzeitigen Abhängigkeiten zur Bibliothek Flask und zur Bibliothek NumPy und zum anderen Dateiduplikate. Eine weitere Vorverarbeitung analog [3] dient der Extraktion relevanter Programmmerkmale (Bezeichner, Symbolsequenzen, usw.) und Normalisierung von Typannotationen: Entfernen von `Any` und `None`, Auflösen von Typaliasen, Qualifizierung und Beschränkung der Verschachtelungstiefe generischer Typen. Kennzahlen für den sich ergebenden und öffentlich verfügbaren Datensatz *CrossDomainTypes4Py* [1] mit über einer Million Programmbeispielen sind in Tabelle 1 angegeben.

3 Experimente zur Domänenabhängigkeit

Zunächst untersuchen wir, ob es Unterschiede in den statistischen Verteilungen der zur Typinferenz mit Type4Py genutzten Programmmerkmale, insbesondere natürlich auch der vorkommenden Typen, zwischen den Domänen Webprogrammierung und wissenschaftliches Rechnen gibt. Tatsächlich fällt etwa bei Betrachtung der zehn häufigsten Typen unseres Datensatzes auf, dass die Typen `str`, `Optional[str]` und `dict` vermehrt in der erstgenannten Domäne vorkommen, wohingegen `int`, `float` und `numpy.ndarray` häufiger in der letztgenannten auftreten. Gleichzeitig können wir beobachten, dass beide Domänen lediglich 3.755 Typen gemeinsam haben, bei insgesamt 15.177 beziehungsweise 27.611 Typen. Der Grund hierfür ist in der schiefen Verteilung der Typen zu suchen, mit vielen weniger häufig auftretenden Typen, die oft projekt- oder domänenspezifisch sind.

Um die Auswirkung der Verteilungsunterschiede auf die Typinferenz mit Type4Py zu beurteilen, wird ein Modell mit Programmbeispielen aus der Domäne Webprogrammierung trainiert und anschließend mit Programmbeispielen der Domäne wissenschaftliches Rechnen evaluiert. Ein zweites Modell wird mit Programmbeispielen aus der Domäne wissenschaftliches Rechnen sowohl trainiert als auch evaluiert. Beim Vergleich der Vorhersagequalität der zwei Modelle sehen wir eine Verringerung des F1-Werts für die exakte Typvorhersage. Auch wenn wir, anstatt des auf unterschiedlichen Domänen trainierten und evaluierten Modells, ein auf dem ursprünglichen ManyTypes4Py-Datensatz [3] trainiertes und auf den Programmbeispielen zum wissenschaftlichen Rechnen evaluiertes Modell betrachten, können wir eine Verringerung des F1-Werts um bis zu 14,15 beobachten. Werden unbekannte Typen bei der Evaluation vernachlässigt, das heißt Typen die in den Evaluations- aber nicht in den Trainingsdaten vorkommen, ergibt sich zwar eine erhebliche Verbesserung der Vorhersagequalität, der Unterschied in der Typinferenz mit Type4Py in unterschiedlichen Domänen lässt sich aber dadurch nicht vollständig erklären. Weitere Experimente bestätigen diese Beobachtungen.

Literatur

[1] Gruner, Bernd ; Heinze, Thomas S. ; Brust, Clemens-Alexander: *CrossDomainTypes4Py: A Python Dataset for Cross- Domain Evaluation of Type Inference Systems*. http://dx.doi.org/10.5281/zenodo.5747024. Version: Januar 2022

[2] Gruner, Bernd ; Sonnekalb, Tim ; Heinze, Thomas S. ; Brust, Clemens-Alexander: Cross-Domain Evaluation of a Deep Learning-Based Type Inference System. In: *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*, IEEE, 2023, 158–169

[3] Mir, Amir M. ; Latoskinas, Evaldas ; Gousios, Georgios: ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-based Type Inference. In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, IEEE, 2021, 585–589

[4] Mir, Amir M. ; Latoskinas, Evaldas ; Proksch, Sebastian ; Gousios, Georgios: Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, 2022, 2241–2252

Java-TX Eclipse-Plugin

Daniel Holle[1,2] und Martin Plümicke[1,3]

[1] Duale Hochschule Baden-Württemberg, Campus Horb, Department of Computer Science Florianstraße 15, D–72160 Horb, Germany, https://www.dhbw-stuttgart.de/horb/forschung-transfer/forschungsschwerpunkte/typsysteme-fuer-objektorientierte-programmiersprachen
[2] d.holle@hb.dhbw-stuttgart.de
[3] pl@dhbw.de

Zusammenfassung. Java-TX is a language based on Java 8. It extends the core of Java with a global type inference scheme. This means that, in most cases, the types can be left out in the source code. This is greatly reducing the programmer effort, especially in complex scenarios that involve multiple generics and lambda functions. One problem with this approach however, is that the types are no longer present in the source code to be inspected. So an effort has to be made in order to visualize the sometimes multiple types that have been substituted. This paper introduces a plugin which extends the Eclipse IDE with a new text editor explicitly made for Java-TX. Its most important feature is the visualization of types and the ability to automatically place the concrete types into the source code.

1 Einleitung

Java-TX [9] ist eine Programmiersprache basierend auf Java 8. Sie erweitert Java um eine globale Typinferenz. Das heißt, Typen werden automatisch an der richtigen Stelle eingesetzt und müssen nicht erst deklariert werden. Manchmal können allerdings mehrere Lösungen valide sein. In diesem Fall werden automatisch Überladungen von Methoden erstellt. Für weitere Details zu Java-TX ist auf Abschnitt 6 verwiesen.
Da es im Sourcecode alleine nicht ersichtlich ist, welche Typen letztendlich eingesetzt wurden, muss auf andere Weise eine Visualisierung erstellt werden. Moderne IDEs lassen sich über verschiedene Schnittstellen erweitern, um Support für andere Programmiersprachen bereitzustellen. Im Folgenden soll eine Implementierung eines solchen Plugins für die Eclipse-IDE vorgestellt werden. Diese Arbeit basiert auf den Arbeiten [5,10].

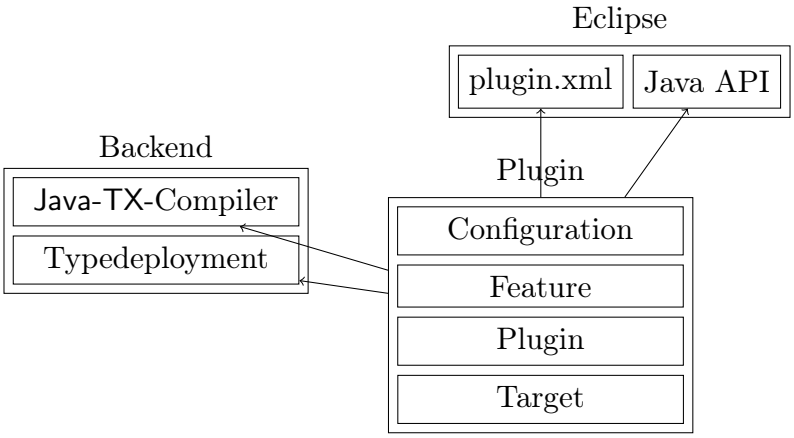
2 Motivation und Problemstellung

Im Wesentlichen sollen zwei Features umgesetzt werden. Zunächst soll mit Hilfe von Markern die generierten Generics[7] und andere Typen im Sourcecode visualisiert werden. Dann soll es mittels Kontextmenü möglich sein, die generierten

Typen an der richtigen Stelle einzusetzen, falls diese weiter eingegrenzt werden sollen. Besonders bei komplizierten Typen ist es nicht immer sinnvoll, diese Schritte von Hand durchzuführen. Bei komplexen Beispielen soll die vom Compiler eingesetzte Nebenläufigkeit eingesetzt werden um die Ergebnisse anzuzeigen sobald eine Lösung vorhanden ist. Dieser Vorgang soll auch im User-Interface abgebrochen werden können.

3 Aufbau des Plugins und verwendeter Schnittstellen

Das Plugin besteht aus zwei Teilen. Zunächst gibt es den Java-TX-Compiler, welcher das Backend des Plugins darstellt. Hier implementert ist die Typinferenz und die Überladung von Methoden, sowie das Generieren von Generics. Kommuniziert wird über das Package `typedeployment`. Ebenfalls wichtig sind die Offsets der Typannotationen, damit die Marker an der richtigen Stelle eingesetzt werden. Fehlermeldungen bei der Kompilation werden auch an das Plugin weitergegeben.
Das Frontend besteht aus einem Eclipse-Plugin mit eigenem Text-Editor für das Bearbeiten von Java-TX Programmen. Die Eclipse-API wurde verwendet um eigene Marker zu erstellen, sowie für das Kontextmenü und den verwendeten Editor.



3.1 Projektstruktur

Das Plugin ist in mehrere Segmente aufgeteilt. Jedes dieser Segmente besitzt eine eigene Mavenkonfiguration und bildet ein eigenständiges Eclipseprojekt.

Configuration Hier werden die M2E (Maven for Eclipse) Einstellungen gespeichert. M2E ist für das Generieren von Eclipseprojekten aus der Mavenkonfiguration verantwortlich. Aus diesem Grund wird für die Bearbeitung des Plugins auch Eclipse als IDE eingesetzt.

Weiterhin werden hier auch die transitiven Abhängigkeiten des Java-TX Compilers in einer separaten pom-Datei spezifiziert. Für die Verarbeitung von Eclipse-spezifischen Abhängigkeiten wird das Tycho Plugin verwendet. Eclipse basiert auf dem OSGi-Modell. OSGi ist eine Schnittstelle die es erlaubt Abhängigkeiten zwischen Modulen zu modellieren, welche auf der JVM ausgeführt werden. Die hierzu verwendeten Module werden außerhalb von Maven von sogenannten Software Repositories importiert. Die Liste dieser Abhängigkeiten wird in einer XML-Datei gespeichert welche im Target-Segment angelegt ist.
Da Plugins für Eclipse signiert werden sollten, wird das dafür verantwortliche Mavenplugin „maven-jarsigner-plugin“ importiert. Die Signierung erfolgt in Java üblicher Weise mit jarsigner und wird von Maven automatisch ausgeführt.

Feature Um Plugins auf dem Eclipse Market Place zu veröffentlichen muss eine Featuredefinition erstellt werden. Dabei kann ein Plugin aus verschiedenen Features bestehen oder aber ein Feature aus mehreren Plugins. Die Features werden dann unter dem Wurzelverzeichnis der Software Repositories gelistet. Ein Feature kann eine eigene Lizenz besitzen, welche vom Endbenutzer akzeptiert werden muss bevor das Feature installiert wird.

Plugin Im Plugin Segment wird der eigentliche Quellcode des Plugins gespeichert. Es gibt ein `src` Verzeichnis in dem die Java Dateien angelegt sind. In der Datei `plugin.xml` werden die einzelnen Erweiterungspunkte definiert, welche über die Eclipse-API implementiert werden. Eine weitere wichtige Datei ist das Manifest `MANIFEST.MF`. Im Manifest wird unter anderem der Classpath angelegt. Die genaue Definition unterscheidet sich deutlich von normalen Java Archiven, was mit dem OSGi-Modell zusammenhängt. Einer der speziellen Keys ist beispielsweise `Require-Bundle` in dem auf die verschiedenen Module der Eclipse-API verwiesen wird.

Target In diesem Segment befindet sich die Target-Definition. Diese enthält wie bereits erwähnt die Definitionen für sämtliche Eclipse-spezifischen OSGi-Module. Die Module werden von verschiedenen Software Repositories importiert.

3.2 Eclipse API

Um das Plugin in die Eclipse Umgebung zu integrieren werden mehrere Extensions definiert. Eine Extension agiert hierbei als die Definition einer Schnittstelle zwischen dem Plugin und Eclipse. Damit Eclipse beispielsweise die verwendete Klasse für einen Editor findet, wird diese in der Datei `plugin.xml` hinterlegt.

Konkret werden folgende Extensions definiert:

```
JavaTX Editor:
<extension
```

```
    point="org.eclipse.ui.editors">
    <editor
        class="typinferenzplugin.editor.JavEditor">
        ...

Typmarker:
<extension
    point="org.eclipse.core.resources.markers">
    ...

Annotation in der Editorleiste:
<extension
    point="org.eclipse.ui.editors.annotationTypes">
    ...

Aussehen des Markers im Editor:
<extension
    point="org.eclipse.ui.editors.markerAnnotationSpecification">
    ...

Kontextmenü für das Einsetzen von Typen:
<extension
    point="org.eclipse.ui.menus">
    <menuContribution
        class="typinferenzplugin.editor.RightClickMenu">
        ...
```

Der `class` Parameter agiert als Einstiegspunkt für die jeweilige Extension. Der `point` steht für ein Package innerhalb von Eclipse, welches durch die Extension erweitert werden soll.

4 Features

4.1 Editor

Für das Editieren von Java-TX-Dateien (`.jav`) wird ein eigener Texteditor bereitgestellt. Dieser basiert auf dem in Eclipse internen Texteditor und verhält sich demnach auch so. Der Editor ist zuständig für das Generieren der Typmarker und der Fehlermarker. Ein Typmarker wird immer an einer Stelle eingefügt, wo der Compiler einen oder mehrere Typen einsetzen kann. Das können in generischen Methoden die generierten Generics sein, oder bei nicht generischen Methoden ein oder mehrere konkrete Typen welche überladen werden. Die Fehlermarker werden eingefügt, falls der Compiler eine Fehlermeldung erzeugt.
Außerdem wird die Mausposition erfasst, und der für das Kontextmenü angeklickte Marker ausgewählt. Wenn die geöffnete Datei gespeichert wird, ruft der Editor den Java-TX Compiler auf um die Klasse neu zu generieren.
Ebenfalls implementiert für den Editor ist die Codevervollständigung. Diese zeigt, insofern der Typ bekannt ist, bei Aktivierung die möglichen Felder und Funkionen an, welche für diesen Typ implementiert sind.

4.2 Outline

Die Outline zeigt die Struktur der Java-TX-Klassen. Ähnlich wie auch bei der Java-Outline werden hier Klassen, Methoden und Felder mit den dazugehörigen Typen angezeigt. Die Outline folgt hierbei einer Baumstruktur, in der die einzelnen Elemente hierarchisch angeordnet werden. Für Java-TX bedeutet das, dass hier entweder die Platzhaltertypen angezeigt werden, oder aber konkrete Typen wenn diese eingesetzt wurden. Wie bei dem Kontextmenü für Marker, kann man hier auch die Typen einsetzen lassen.

4.3 Kontextmenü

Das Kontextmenü listet einen oder mehrere Typen, die für einen Typmarker eingesetzt werden können. Es wird bei einem Rechtsklick auf einen der Marker im Editor angezeigt.

5 Implementierung

Die Implementierung besteht aus zwei Teilen. Als erstes gibt es das Frontend, welches mit der Eclipse-API interagiert und die Benutzeroberfläche bereitstellt. Von hier wird der Compiler aufgerufen, der eine im Java-TX-Editor geöffnete Datei kompiliert. Die Schnittstelle, die das Resultat an das Eclipse-Plugin weitergibt, befindet sich im Paket `de.dhbwstuttgart.typedeployment`.
Die Idee hinter der Aufgabenteilung zwischen dem Java-TX-Compiler Projekt und dem Eclipse-Plugin ist, dass gemeinsame Logik, die für verschiedene Anwendungen basierend auf Java-TX benötigt wird, zentral bereitgestellt wird. So kann in Zukunft zum Beispiel ein neues Plugin geschrieben werden, welches für eine andere Entwicklungsumgebung benutzt wird. Denkbar wären auch weitere Anwendungen zur statischen Analyse des Quelltextes. Ein weiterer Punkt ist, dass das Plugin nur mit einer Schnittstelle agiert und deshalb nicht selbst in die Datenstruktur des Compilers eingreifen muss. Die Kupplung zwischen den beiden Projekten wird also auf ein Minimum verringert. So kann der Compiler angepasst werden und neue Funktionen implementiert werden, ohne dass das Plugin neu geschrieben werden muss.

5.1 Typedeployment

Hier gibt es drei Klassen, welche den Hauptteil der Arbeit erledigen. Die Datenstruktur welche letztendlich an das Eclipse-Plugin weitergegeben wird, ist ein Set aus `TypeInsertPoint` innerhalb der Klasse `TypeInsert`. Ein `TypeInsertPoint` besteht aus einem ANTLR-Token, welches den Offset innerhalb des Quellcodes speichert und einem String welcher das einzusetzende Ergebnis enthält. Wenn mehrere Marker an der selben Stelle stehen, besitzen sie das selbe Token. Das ist bei der Einsetzung relevant, wenn einer der Marker ausgewählt werden muss. Eclipse legt mehrere Marker, wenn sie sich an der selben Stelle

befinden, zusammen. Jeder Punkt besitzt außerdem einen Typ aus dem Enum `KindOfTypeInsertPoint`. Hier gibt es normale Inserts, also solche wo innerhalb einer Klasse Typen eingesetzt werden. Außerdem gibt es Inserts, welche den generischen Parametern entsprechen. Hierbei wird zwischen Klassen und Methoden unterschieden.
Jeder `TypeInsertPoint` besitzt eine Methode um das Ergebnis in den Quelltext einzusetzen. Übergeben wird dazu der besagte Quelltext als String und dieser wird verändert zurückgegeben. Hierfür wird einfach der vorher bestimmte Text an der richtigen Stelle eingesetzt. Dafür wird die Position des Tokens verwendet.
Erstellt werden die Punkte von der Klasse `TypeInsertFactory`. Diese generiert für die generischen Parameter von Klassen und Methoden die jeweils dazugehörigen Punkte zusammen mit dem Text welcher eingesetzt wird. Für die Generierung der Punkte innerhalb des Quelltexts wird das Visitor-Pattern eingesetzt. Hierfür werden zwei `ASTWalker` implementiert. Der erste kümmert sich um die Klassendefinitionen innerhalb einer Quelldatei und ruft wiederum den zweiten Visitor `TypeInsertPlacerClass` auf, welcher für die einzelnen Methoden und Felder die verschiedenen Punkte generiert.
Der Text eines Punktes wird auch durch einen Visitor `TypeInsertToString` erstellt, dieser ist diesmal ein `ResultSetVisitor`. Dieser wird mit den verschiedenen Klassen von Typen welche der Compiler kennt, aufgerufen.

`RefType`: Ein Referenztyp, erbt von `java.lang.Object`.

`GenericRefType`: Instanz eines generischen Parameter, definiert durch `<T>` innerhalb einer Klasse oder Methode.

`SuperWildcardType`: Typ der Form `? super A`, wobei ein Supertyp von A gemeint ist.

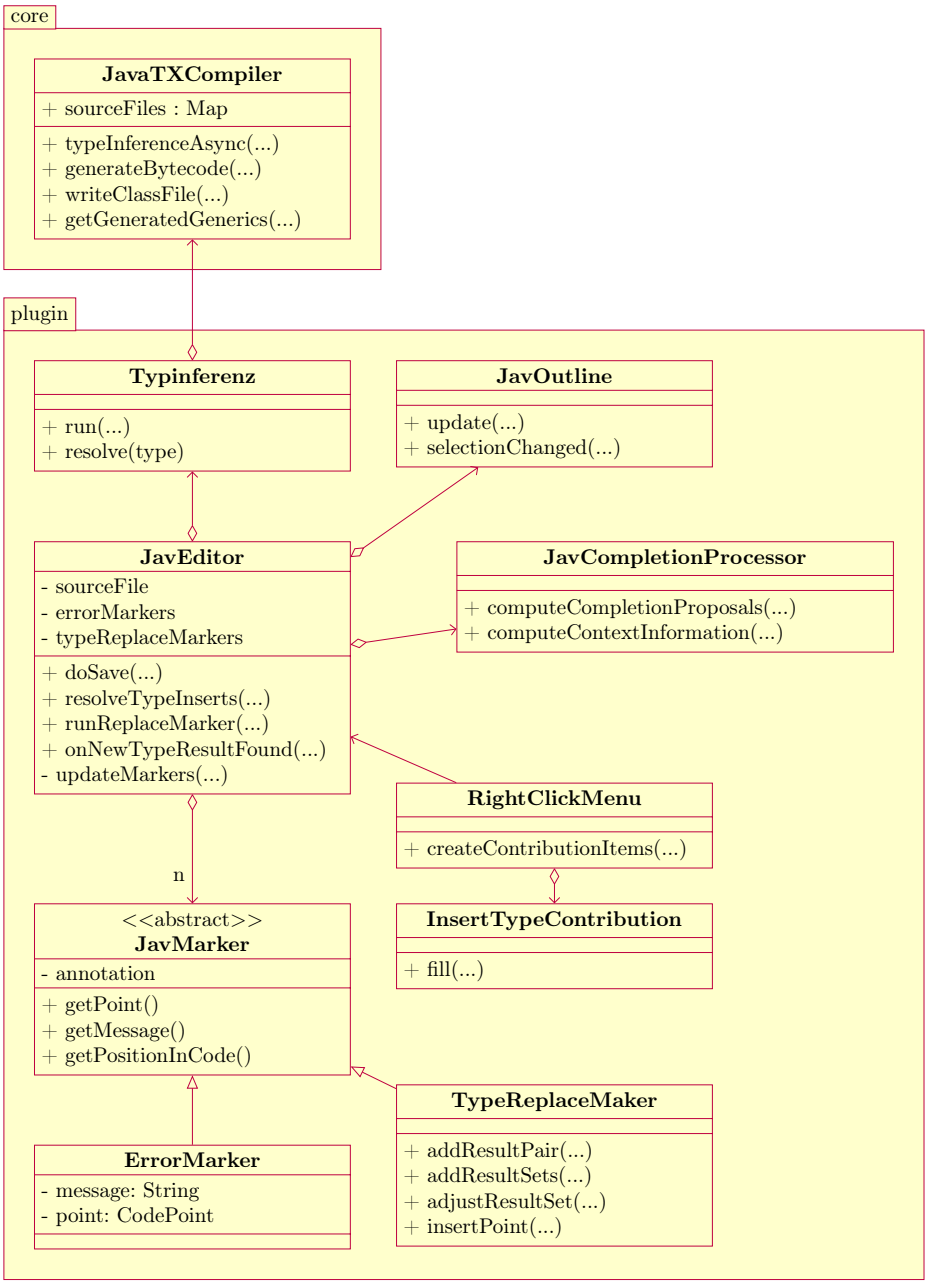
`ExtendsWildcardType`: Typ der Form `? extends A`, wobei ein Subtyp von A gemeint ist

`TypePlaceholder`: Typ der im Quelltext weggelassen wurde und durch die Typinferenz ausgefüllt wird. Übrige Constraints mit Typplatzhaltern werden als Generics realisiert, das heißt im Endeffekt wird jeder Typplatzhalter aufgelöst.

Für jeden dieser Typen wird die dazugehörige Textrepräsentation erstellt. Für die Referenztypen werden beispielsweise der Name und die generischen Typen innerhalb von spitzen Klammern angefügt.

5.2 Klassendiagramm des Plugins

Dieses Diagramm ist vereinfacht und stellt nur die wichtigsten Klassen bzw. Methoden dar.



5.3 Editor

Die Klasse `JavEditor` ist der Dreh- und Angelpunkt des Plugins. Hier werden die verwendeten Marker gespeichert. Ebenso wird die Outline, sowie über einen Adapter der Vervollständigungsprozessor `JavCompletionProcessor`. Kommuniziert mit dem Backend wird über die Klasse `Typinferenz`, welche ebenfalls vom Editor instanziiert wird. Bei einem Speichern wird der Zustand zurückgesetzt und die Typinferenz von neuem ausgeführt.
Der Editor wird als `UnifyResultListener` implementiert, welcher von der Typinferenz aufgerufen wird, wenn ein neues Resultat zu Verfügung steht. Dies ist nötig, da die Typinferenz nebenläufig arbeitet und daher auf einem separaten Thread läuft. Dies ist vom Plugin her gesehen sinnvoll, da so nicht die Benutzeroberfläche hängt, wenn die Typinferenz läuft.
Die Methode `runReplaceMarker` wird aufgerufen, wenn einer der Marker eingesetzt werden soll. Diese Funktion wird von dem jeweiligen Menü aus aufgerufen. Verwaltet wird hier der Zustand der einzelnen Marker, da die Position und die Validität sich jeweils verändert, wenn ein Marker eingesetzt wird.
Die Methode `resolveTypeInserts` wird vom Vervollständigungsprozessor aufgerufen, um die konkreten Typen, welche für einen TypePlaceholder eingesetzt werden, zu bestimmen.
Ebenfalls vom Editor ausgeführt wird die Bestimmung der Mausposition im Code, da so festgestellt werden kann, ob das Rechtsklickmenü für einen Marker angezeigt werden soll. Diese Lösung ist nicht ganz ideal, aber da von Eclipse keine API bereitgestellt wird, welche ein Rechtsklickmenü für einen Marker darstellt, musste dieser Workaround implementert werden. In einer vorherigen Version wurde das Menü über die Quickfix-Funktion von Eclipse für einen Marker angezeigt. Da diese Quickfixes nun aber in verschiedenen Untermenüs oder über eine Tastenkombination aufgerufen werden müssen, wurde zwecks Benutzerfreundlichkeit die Lösung mit dem Rechtsklickmenü angewendet.

5.4 Outline

Die Outline stellt die für eine Java-TX-Klasse verwendete Struktur da. Hier sieht man gut die jeweiligen Overloads für die verschiedenen Funktionen, sowie die verwendeten Typplatzhalter und deren Namen. Ebenfalls implementert wird hier ein Rechtsklickmenü, welches ähnlich zu dem im Editor verwendeten funktioniert. Registriert wird die Outline nicht wie sonst üblich über die `plugin.xml` Datei sondern es wurde ein Adapter verwendet. Wenn im Editor die Methode `getAdapter` mit der Klasse `IContentOutlinePage` aufgerufen wird, wird eine neue Instanz der Outline zurückgegeben. Implementiert wurde die Outline durch eine Subklasse des `OutputGenerator` aus dem Compiler. Hierbei handelt es sich um einen Vistor, welcher den Syntaxbaum abläuft und eine Textrepräsentation erstellt. Die Eclipse-API stellt bereits eine Möglichkeit bereit, eine Baumstruktur als Menü darzustellen. Diese Möglichkeit wird hier genutzt, und jeweils der Text für die einzelnen Elemente erstellt.

5.5 Vervollständigungsprozessor

Der `JavCompletionProcessor` wird jeweils im Editor aufgerufen, wenn das Shortcut zur Vervollständigung benutzt wird, oder aber ein Punkt für die Feld und Funktionsaufrufe getippt wird. Die Idee dahinter ist, dass das Programmieren mit Java-TX vereinfacht wird, da nicht immer klar ist, welcher Typ an welcher Stelle verwendet wird. So sieht man auf einen Blick, welche Funkionen oder Felder im Kontext eines Symbols valide sind. Die Implementierung arbeitet mit Reflections, es wird also die verwendete Klasse geladen. Diese wird durch den Editor mit der Methode `resolveTypeInserts` bereitgestellt. Zurückgegeben wird eine Liste mit Klassennamen, welche an der Eingabeposition eingesetzt werden. Im dargestellten Menü kann man dann die Methode oder das Feld auswählen, welches man eingefügt haben will. Momentan funktioniert die Vervollständigung nur teilweise, gerade wenn mehrere Methoden überladen werden, wird nicht immer der richtige Kontext ausgesucht. Hier müsste also noch nachgebessert werden.

5.6 Rechtsklickmenü

Das Rechtsklickmenü im Editor ist ähnlich implementert, wie das Menü für die Outline. Das Menü wird seperat über die `org.eclipse.ui.menus` Schnittstelle registriert und hier wird außerdem die Stelle an der das Menü angezeigt wird auf `popup:#TextEditorContext` gesetzt, und dadurch wird das Menü nur für den Texteditor angezeigt. Durch den Editor wird die momentan genutzte Mausposition ermittelt, und daraus die Cursorposition berechnet. Das `RightClickMenu` generiert, falls der Editor dem JavEditor entspricht, eine `InsertTypeContribution` welche von `ContributionItem` erbt. Diese fügt dann mit Hilfe der Methode `fill` eine oder mehrere MenuItems an den Anfang des Kontextmenüs. Der Editor befüllt eine Liste von `TypeInsertMarker` die sich auf der Mausposition befinden.

5.7 Marker

Für das Plugin werden zwei Typen von Markern definiert. Zum einen der Fehlermarker `ErrorMarker` und der Typmarker `TypeReplaceMaker`. Beide erben von der abstrakten Klasse `JavMarker`. Hierbei ist zu beachten, dass `JavMarker` lediglich für die Implementierung des Plugins verwendet werden. Die eigentliche Marker-Instanz die an das Plugin gesendet wird befindet sich im Feld `annotation` welches vom Typ `Annotation` ist. Diese Annotationen werden durch die `plugin.xml` Datei definiert, wo auch das Aussehen und das Icon der Marker festgelegt wird. Die Marker werden durch die Funktion `placeMarkers` der Editor Klasse gesetzt. Diese wird asynchron ausgeführt, wieder um ein Hängen der Entwicklungsumgebung zu verhindern. Immer wenn die Typinferenz ein neues Ergebnis berechnet, werden die Marker neu gesetzt. Auch wenn die Datei gespeichert wird, werden die Marker neu generiert. Ein Problem mit den Markern in der momentanen Version ist, dass diese nur ein Zeichen breit sind. Es wäre sicherlich besser, wenn die Marker das gesamte Symbol auf das sie angewendet

werden einnehmen. Da das Token bereits diese Information hat müsste es relativ einfach sein, diese Änderung vorzunehmen.

6 Java-TX

In Java gibt es bisher keine *globale Typinferenz* (auslassen aller Typinformationen und automatische Berechnung zur Übersetzungszeit), keine *echten Funktionstypen* und kein vollumfängliches *Pattern Matching*. Java-TX (TX steht für *Type eXtended*) wendet sich diesen drei in Java noch fehlenden Eigenschaften zu. Das Herzstück von Java-TX ist die Typinferenz.

6.1 Globale Typinferenz

Globale Typinferenz meint im Gegensatz zu lokaler Typinferenz, dass alle Typen eines Java Programms weggelassen werden können und diese zur Übersetzungszeit inferiert werden. Das heißt, dass Java die statische Typeigenschaft behält. Dies gilt insbesondere auch für rekursive Lambda-Ausdrücke. Für den Typinferenzalgorithmus verweisen wir auf [6]. Drei Beispiele, die die Mächtigkeit der Typinferenz zeigen geben wir hier an.
Das erste Beispiel ist die Fakultät, definiert als rekursiver Lambda-Ausdruck.

```
class Faculty {
    fact[4] = (x) -> {
            if (x == 1) {
                return 1;
            }
            else {
              return x * (fact.apply(x-1));
            }
      };
  }
```

Der Typinferenzalgorithmus berechnet für das Attribut `fact` den echten Funktionstyp `Fun1$$<Integer,Integer>` [5].
Das zweite Beispiel zeigt die zusätzliche Möglichkeiten, die die Typinferenz für das Overloading bietet.

```
class OL     {
   m(x) { return x + x; }

   m(x) { return x || x; }
}

class OLMain   {
```

[4] In Java-TX kann im Gegensatz zum Java–Standard einem Attribut direkt ein Lambda-Ausdruck zugewiesen werden. In Standard Java ist dies nur über einen Konstruktor möglich.
[5] Echte Funktionstypen werden in Abschnitt 6.2 näher betrachtet

```
        main(x) {
                var ol;
                ol = new OL();
                return ol.m(x);
        }
}

System.out.println(ol.main(2));
System.out.println(ol.main(2.0));
System.out.println(ol.main("applied␣to␣a␣string␣"));
System.out.println(ol.main(true));
```

In der Klasse `OL` wird eine überladene Methode `m` für die Addition und eine Methode `m` für das boole'sche *oder* deklariert. In der Funktion `main` der Klasse `OLMain` wird `m` der Klasse `OL` aufgerufen. Dies ergibt nun je nach Argument entweder die eine oder die andere Methode. Dementsprechend ist `main` nun für Werte der Typen `Integer`, `Double`, `String` und `Boolean` aufrufbar.
Das dritte Beispiel zeigt, dass der Typinferenzalgorithmus auch Typen mit Wildcards bestimmt.

```
public class MatrixOP extends Vector<Vector<Integer>> {
  mul = (m1, m2) -> {
    var ret = new MatrixOP();
    var i = 0;
    while(i < m1.size()) {
      var v1 = m1.elementAt(i);
      var v2 = new Vector<Integer>();
      var j = 0;
      while(j < v1.size()) {
        var erg = 0;
        var k = 0;
        while(k < v1.size()) {
          erg = erg + v1.elementAt(k)
                    * m2.elementAt(k).elementAt(j);
          k++; }
        v2.addElement(erg);
        j++; }
      ret.addElement(v2);
      i++;}
    return ret;}
```

Abb. 1. Matrix–Multiplikation

In der Klasse `MatrixOP` (Abb. 1), die eine Erweiterung von `Vector<Vector<Integer>>` ist, wird ein Attribut `mul` deklariert. `mul` wird ein Lambda-Ausdruck zugeordnet, der die Matrix–Multiplikation berechnet. Als Typ für `mul` wird berechnet:

```
mul: Fun2$$<Vector<? extends Vector<? extends Integer>>,
            Vector<? extends Vector<? extends Integer>>,
            MatrixOP>
```

6.2 Integration von echte Funktionstypen und functional interfaces

In Version 8 von Java [4] wurde die Realisierung von Lambda-Ausdrücken mit Functional Interfaces als Target Types realisiert. Die Nachteile dieses Ansatzes haben wir in [8] dargestellt. Weiterhin haben wir dort beschrieben wie wir Funktionstypen analog zu Scala in Java-TX eingeführt haben.
Der Ansatz in Java-TX ist echte Funktionstypen einzuführen, gleichzeitig aber das Target Typing mit Functional Interfaces zu erhalten und beide Ansätze zu integrieren.
Dazu haben wir in Java-TX zwei Bündel von speziellen Functional Interfaces ergänzt:

```
FunN$$<-A1, ... , -AN, +R> {
    R apply(A1 arg1, ... AN argN);
}

FunVoidN$$<-A1, ... , -AN> {
    R apply(A1 arg1, ... AN argN);
}
```

Die Interfaces haben im Gegensatz zu sonstigen Typen in Java *Declaration-site Varianz*. Das heißt, dass bereits bei der Deklaration festgelegt wird, welche Parameter kovariant und welche kontravariant sind. '-' steht dabei für Kontravarianz und '+' für Kovarianz. Weiterhin sind für die Interfaces Wildcards als Parameter nicht zulässig. Lambda-Ausdrücke werden durch diese Interfaces explizit typisiert.
Als Beispiel betrachten wir aus JavaFX die Methode `setOnAction` der Klasse `javafx.scene.control.Button`. Sie erwartet als Argument einen `javafx.event.EventHandler`. Das Interface `javafx.event.EventHandler` ist ein Functional Interface:

```
interface EventHandler<T extends javafx.event.Event>
{
    void handle(T event);
}.
```

Wir definieren zunächst einen explizit getypten Lambda-Ausdruck:

```
Fun1Void$$<ActionEvent> helloworld =
    event -> System.out.println("Hello␣World!");
```

Diesen geben wir nun als Argument der Methode `setOnAction` mit:

```
Button btn = new Button ();
btn.setOnAction(helloworld);
```

Das Beispiel zeigt die Integration der beiden Konzepte *Typisierung von Lambda-Ausdrücken durch echte Funktionstypen* und *Target Typing von Lambda-Ausdrücken*. Dem Argument der Methode `setOnAction` mit Target Type `EventHandler` wird die explizit getypte Variable `helloworld`, belegt mit einem Lambda-Ausdruck, zugeordnet.

6.3 Pattern–Matching

Pattern–Matching wird derzeit in Java-TX implementiert. Erste Überlegungen zur Realisierung wurden in [11] angestellt. In Java werden seit der Version 14 zum Teil als sogenannte *Preview–Features* Pattern–Matching Ansätze für `instanceof`- und `switch`-Statements realisiert. [3,2]. Unterstützend wurden dabei *sealed classes* and *record*–Typen eingeführt.
In Java-TX soll Pattern–Matching wie in funktionalen Programmiersprachen mit Typinferenz gekoppelt werden. Dazu eigenen sich besonders die *Record Patterns* [1], die für Version 21 angekündigt sind.
Folgendes Beispiel zeigt den Ansatz. Seien die Datenstrukturen definiert:

```
record Point(int x, int y) {}

enum Color { RED, GREEN, BLUE }

record ColoredPoint(Point p, Color c) {}

interface Shape {}
record Rectangle(ColoredPoint upperLeft,
                 ColoredPoint lowerRight) implements Shap {}
record Circle(ColoredPoint c, int r) implements Shape {}
```

Die Methode, die eine Figur drehen soll, kann mit Hilfe von Pattern Matching wie folgt deklariert werden:

```
Shape rotate(Shape shape, double angle) {
  return switch (shape) {
    case Circle c -> c;
    case Rectangle r -> r.rotate(angle);
```

Pattern Matching ermöglicht es die Matching Variablen `c` and `r` auf der rechten Seite bei der Ausführung zu benutzen.
Des weiteren ist es möglich geschachtelte Pattern zu definieren. Als Beispiel dafür wird in der folgenden Methode der Mittelpunkt einer Figur bestimmt.

```
Point center(Shape shape) {
  switch (shape) {
  case Rectangle(ColoredPoint(Point pt1, Color color1),
                 ColoredPoint(Point pt2, Color color2)
                 -> center(pt1, pt2);
  case Circle(ColoredPoint(Point pt,Color color),int r)->pt;
}
```

Die Erweiterung in Java-TX soll zunächst die Nutzung von Pattern Matching ähnlich wie in funktionalen Sprachen ohne Typdeklarationen der Matching Variablen ermöglichen:

```
Point center((shape) {
  switch (shape) {
  case Rectangle(ColoredPoint(pt1,color1),
                 ColoredPoint(pt2,color2)->center(pt1,pt2);
  case Circle(ColoredPoint(pt,color), r) -> pt;
}
```

Dies wird in Java–21 mit Hilfe des `var`–Schlüsselworts auch möglich sein. Weiterhin soll Pattern Matching auch in Methodenköpfen möglich werden:

```
Point center(Rectangle(ColoredPoint(pt1, color1),
                       ColoredPoint(pt2, color2)) {
  return center(pt1, pt2);
}

Point center(Circle(ColoredPoint(pt,color), r)) {
  return pt;
}
```

7 Ausblick

Da das momentan implementierte Plugin nur für eine einzige Entwicklungsumgebung funktioniert, nämlich Eclipse, in der Praxis allerdings verschiedene Entwicklungsumgebungen verwendet werden, wäre es wünschenswert für diese auch ein Plugin zu entwickeln. Da das Projekt modular aufgebaut wird und der Java-TX-Compiler die eigentliche Implementierung der Features bereitstellt, muss dieser hierfür nur geringfügig angepasst werden.
Eine in den letzten Jahren immer häufiger verwendete API ist das von Microsoft entwickelte Language Server Protocol[6]. Es ist ein größtenteils Sprachagnostisches Framework, welches für verschiedene Entwicklungsumgebungen verwendet wird. Die Basis ist hierbei ein JSON-RPC mit dem über Sockets oder Pipes Befehle von der Entwicklungsumgebung an das Backend des Plugins gesendet werden. Dieses kann also weiterhin in Java geschrieben werden. Mit diesem Framework könnte man also ein Plugin schreiben, welches größtenteils unabhängig von der Entwicklungsumgebung funktioniert. Hierdurch könnten also andere Entwicklungsumgebungen wie Visual Studio Code, IntelliJ IDEA oder sogar Emacs unterstützt werden. Ein Nachteil ist allerdings, dass Funktionen wie das Rechtsklickmenü nicht direkt unterstützt werden. Es müsste also eine mit dem LSP kompatible Lösung gefunden werden, wie beispielsweise durch Refactorings. Durch das LSP könnten auch sogenannte Typinlays (also im Sourcecode angezeigter, generierter Text) realisiert werden. Diese würden sich gerade für Java-TX besonders eigenen, wenn nur ein Typ berechnet wurde.

[6] `https://microsoft.github.io/language-server-protocol/`

Ein weiters Feature, welches noch implementiert werden sollte ist das Syntaxhighlighting. Hier könnte eine auf Java basierte Grammatik eingesetzt werden, wie sie auch im Java-TX-Parser verwendet wird. Angepasst werden müssten nach jetzigem Stand nur die optionalen Typen.

Literatur

1. Bierman, G.: Jep 440: Record patterns (2023), http://openjdk.java.net/jeps/440, updated: 2023/08/28 16:51
2. Bierman, G.: Jep 441: Pattern matching for switch (2023), http://openjdk.java.net/jeps/441, updated: 2023/08/28 16:51
3. Goetz, B.: Jep 394: Pattern matching for instanceof (2020), http://openjdk.java.net/jeps/394, updated: 2022/06/10 16:12
4. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java® Language Specification. The Java series, Addison-Wesley, Java SE 8 edn. (2014)
5. Hake, T., Stresing, C.: Erweiterung des Eclipse Plugins zur Unterstützung der Arbeit mit dem Typinferenz Compiler. BA Stuttgart, Studienarbeit (2008), (in german)
6. Plümicke, M.: More type inference in Java 8. Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers **8974**, 248–256 (2015). https://doi.org/10.1007/978-3-662-46823-4_20
7. Plümicke, M., Holle, D.: Principal generics in Java–TX. In: Noll, T., Fesefeldt, I. (eds.) Programmiersprachen und Grundlagen der Programmierung, 22. Kolloquium, KPS'23. Aachener Informatik-Berichte (AIB), Aachen (September 2023)
8. Plümicke, M., Stadelmeier, A.: Introducing Scala-like function types into Java-TX. In: Proceedings of the 14th International Conference on Managed Languages and Runtimes. pp. 23–34. ManLang 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3132190.3132203, http://doi.acm.org/10.1145/3132190.3132203
9. Plümicke, M., Zink, E.: Java-TX: The language. INSIGHTS – Schriftenreihe der Fakultät Technik 01/2022, DHBW Stuttgart (2022), https://www.dhbw-stuttgart.de/forschung-transfer/technik/schriftenreihe-insights
10. Stadelmeier, A.: Java type inference as an eclipse plugin. In: 45. Jahrestagung der Gesellschaft für Informatik, Informatik 2015, Informatik, Energie und Umwelt, 28. September - 2. Oktober 2015 in Cottbus, Deutschland. pp. 1841–1852 (2015), http://subs.emis.de/LNI/Proceedings/Proceedings246/article18.html
11. Trumpfheller, L.: Erweiterung des Java-TX-Compilers um Pattern Matching im Stil funktionaler Programmiersprachen. DHBW Mannheim, Studienarbeit (2023), (in german)

Optimierung von Random-Access Listen in Haskell

Frank Huch, Maja Reichert

Institut für Informatik
Christian-Albrecht-Universität zu Kiel

In der modernen Programmierung sind dynamische Listen eine zentrale Datenstruktur. Neben der Möglichkeit auf die Elemente effizient über den Index zuzugreifen, könnne diese auch dynamisch verlängert oder verkürzt werden. In Java stehen sie der/dem Programmierer*in als ArrayList zur Verfügung und in Python sind sie die einzige eingebaute arrayartige Datenstruktur.

In der Funktionalen Programmierung sind Listen ebenfalls eine zentrale Datenstruktur. Allerdings erlauben diese nur einen recht ineffizienten (linearen) Zugriff auf ein Element an einem bestimmten Index. Okasaki hat als Alternative eine funktionale Datenstruktur für Random-Access-Lists vorgestellt, mit welcher alle Zugriffe und Veränderungen in logarithmischer Laufzeit in der Länge der Liste möglich sind. Diese Implementierung basiert auf Ketten von größer werdenden blattmarkierten Binärbäumen. Wegen der komplexen Struktur sind für den Zugriff mehrere (logarithmische) Operationen notwendig, wodurch die absolute Laufzeit z.T. weit hinter der bei gewöhnlichen Haskell-Listen zurück bleibt. Dies hängt auch damit zusammen, dass in der Datenstruktur nur zwei Fälle unterschieden werden, welche aber jeweils im Speicher als ein Speicherwort repräsentiert werden und somit viel Speicher verbrauchen.

Wir präsentieren eine Optimierung dieser Struktur, welche es ermöglicht dynamische Arrays kompakter und bis zu doppelt so effizient umzusetzen. Die Idee ist die Verwendung von mehr Fallunterscheidungen, so dass die Repräsentation kompakter wird. Diese Idee geht damit einher, dass man in der Implementierung anstelle von Binärzahlen andere Zahlensysteme, wie z.B. Hexadezimalzahlen verwendet.

Die Optimierung erfolgt recht schematisch, so dass es sich im wesentlichen um Boilerplate-Code handelt. Diesen erzeugen wir mit Hilfe von Template-Haskell automatisch und können so skalieren, wie sehr wir die Datenstruktur komprimieren wollen. Der generierte Code wird hierbei allerdings immer größer, so dass sich ein Optimum ergibt, welches dann praktisch genutzt werden sollte.

Towards Improved Test-to-Code Traceability via Mutation Testing

Kerstin Jacob[1] and Gerald Lüttgen[1]

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

Software traceability is the ability to connect different artifacts created during the development of a software system [SZ05], where test-to-code traceability links are especially important. Firstly, they aid impact analysis by showing the tests that are likely affected by a code change and vice versa. They assist developers to keep test and code in sync, thereby reducing the rate of test failures [WK22], and are also employed in regression test suite optimization to determine which tests to execute after a code change [RD09]. Secondly, test-to-code tracelinks are beneficial in the context of program comprehension, where tests serve as a valuable documentation artifact for the code, which shows how parts of a system are supposed to be used [LFO08]. When tests are linked to – or automatically generated from – a software specification, test-to-code tracelinks also help one to establish links from the specification to code units.

In practice, test-to-code tracelinks are rarely maintained manually, posing a need for automatic recovery strategies. Even though candidate tracelinks can be determined from code coverage information, many tests invoke functions that are not considered to be amongst the functions under test, such as functions that initialize the state of an object, or helper functions, getters and setters [WK22]. Based on this observation [RD09] proposed six approaches for automatic tracelink recovery, which have been widely adopted and extended since (see, e.g., [Qus+14], [KTV18], and [WK22]). These approaches exploit naming conventions, lexical analysis, static call graphs, and information from version control systems, and operate either at method level, linking production code methods to test methods, or at class level, linking production and test classes. However, many such approaches rely on developer discipline or require strong assumptions on the test structure.

This talk introduces a novel approach to automatically recovering test-to-code tracelinks, which is based on code mutation. It first generates mutants of the production code and then, in the spirit of mutation testing [JH11], links a test to a method in the production code if the respective test kills a mutant in the method. We evaluate our approach on four large open-source projects (GSON, JFreeChart, CommonsIO and CommonsLang) and, when compared to existing tracelink recovery methods [WK22], obtain promising results already with a basic set of mutation operators. Unlike these recovering methods, our approach is able to establish tracelinks at statement level, which is beneficial especially for program comprehension.

References

[JH11] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Trans. Software Eng.* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[KTV18] András Kicsi, László Tóth, and László Vidács. "Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability". In: *6th IEEE/ACM Intl. Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE, 2018*. ACM, 2018, pp. 8–14. DOI: 10.1145/3194104.3194106.

[LFO08] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. "Traceability Management for Impact Analysis". In: *2008 Frontiers of Software Maintenance*. Beijing, China. IEEE, 2008, pp. 21–30. DOI: 10.1109/FOSM.2008.4659245.

[Qus+14] Abdallah Qusef et al. "Recovering Test-to-Code Traceability Using Slicing and Textual Analysis". In: *J. Syst. Softw.* 88 (2014), pp. 147–168. DOI: 10.1016/j.jss.2013.10.019.

[RD09] Bart Van Rompaey and Serge Demeyer. "Establishing Traceability Links between Unit Test Cases and Units under Test". In: *13th European Conference on Software Maintenance and Reengineering, CSMR 2009*. IEEE, 2009, pp. 209–218. DOI: 10.1109/CSMR.2009.39.

[SZ05] George Spanoudakis and Andrea Zisman. "Software Traceability: A Roadmap". In: *Handbook of Software Engineering and Knowledge Engineering* 3 (2005). DOI: 10.1142/9789812775245_0014.

[WK22] Robert White and Jens Krinke. "TCTracer: Establishing test-to-code traceability links using dynamic and static techniques". In: *Empir. Softw. Eng.* 27.3 (2022), p. 67. DOI: 10.1007/s10664-021-10079-1.

The Constraint-Logic Object-Oriented Language Muli

Herbert Kuchen and Hendrik Winkelmann

Universität Münster, kuchen@uni-muenster.de
https://www.wi.uni-muenster.de/de/institut/pi/

Abstract. Muli is an extension of Java by logic variables and a built-in search mechanism. Unbound logic variables in conditions of conditional statements lead to branching of the computation. For each branch, corresponding constraints are added to a constraint store. If the accumulated constraints become unsatisfiable, the corresponding branch is discarded. Muli supports several search strategies such as depth-first search, breadth-first search, and iterative deepening. It allows logic variables not only to represent basic values but also objects and arrays.

Extended Abstract

There are quite a few approaches to combine different programming paradigms. For instance, functional logic programming languages such as Curry [HKMN95] and Babel [KLMR90,MKLR90] combine features from functional and logic programming. Scala [OR14] combines object-oriented and functional programming. Also, current versions of object-oriented languages such as Java include more and more features from functional programming. Higher-order functions and streams are the most prominent ones. There are also attempts trying to combine object-oriented and (constraint) logic programming. For instance, tuProlog [DOR05] enables accessing Java features from Prolog. However, it does not integrate the two paradigms seamlessly. Both paradidgms remain clearly separated. Another approach to combine constraint logic and object-oriented programming is Oz [RBD+03]. It has a logic programming flavor where object orientation is simulated on top of the core language. It does not support the symbolic representation of objects and arrays.

Our approach is the **Mu**enster **L**ogic **I**mperative Programming Language Muli [DK23,WK22]. It extends the object-oriented programming language Java by logic variables as known from the logic programming language Prolog and by encapsulated search as known from the functional logic programming language Curry [HKMN95]. Muli achieves a seamless integration of constraint-logic and object-oriented programming. When the condition of an if-statement or of a loop contains unbound logic variables, this may lead to a branching computation. For each branch, a corresponding constraint is added to a constraint store. If the accumulated constraints of a branch are no longer satisfiable, the branch is discarded. Syntactically, the only addition to Java is the keyword `free`, which

can be used in variable declarations, e.g. in `int x free;`. It indicates that the corresponding variable is an unbound logic variable as in Prolog. In contrast to Prolog, its value may be changed by assignments. This also holds for object fields and array contents. Thus, side-effects are allowed as in Java.

Muli supports several search strategies when dealing with the corresponding search space. In particular, depth-first search, breadth-first search, and iterative deepening can be chosen.

Logic variables and the mentioned search are only allowed in encapsulated search regions. Thus, outside of these search regions, Muli behaves just like Java. This enables non-symbolic computations to be executed efficiently, while providing the built-in search mechanisms where needed. Test-case generation is an example of an application where this combination of symbolic and concrete computing is extremely helpful [WTK22].

While the first versions of Muli only allowed logic variables of basic types, recent extensions now also allow logic variables to represent objects [DWK21] and arrays [WK23]. Class hierarchies are hereby properly taken into account.

The following example solves the well-known NP-complete Knapsack Problem. Notice the use of the boolean logic variable `take` in line 19. It determines whether the considered item should be put into the knapsack or not. The constraints generated by the condition of the if-statement in line 20 ensure that an item is only considered, if it still fits into the knapsack. Finally, the check in line 23 makes sure that a set of selected items is only accepted as a solution, if the capacity of the knapsack is fully used. Notice that the formulated Muli code does not explicitly implement a search algorithm, but leaves the solution to the included constraint solver.

```
1  public class Knapsack {
2      static Item[] items = {new Item("bread",750,10),
3                                ...
4                             new Item("water",1000,30)};
5
6      public static class Item {
7          String item;
8          int weight;
9          int benefit;
10         Item(String it, int w, int b){
11             item = it; weight = w; benefit = b;}
12     }
13
14     public static ArrayList<Item> fillKnapsack(int capacity){
15         int weight = 0;
16         ArrayList<Item> result = new ArrayList<Item>();
17         for (Item item : items) {
18             if (weight == capacity) break;
19             boolean take free;
20             if (weight + item.weight <= capacity && take) {
21                 result.add(item);
22                 weight += item.weight;}}
23         assume(weight == capacity);
24         return result;
25     }
26 }
```

The method `fillKnapsack` can then be used in an encapsulted search region which delivers a list of solutions. These solutions can then be processed e.g. one

by one in a loop. The considered version of the Knapsack Problem ignores the benefits of the items.

References

[DK23] Jan C. Dageförde and Herbert Kuchen. Applications of muli: Solving practical problems with constraint-logic object-oriented programming. In Pedro López-García, John P. Gallagher, and Roberto Giacobazzi, editors, *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems - Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday*, volume 13160 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2023.

[DOR05] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm java-prolog integration in tuprolog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[DWK21] Jan C. Dageförde, Hendrik Winkelmann, and Herbert Kuchen. Free objects in constraint-logic object-oriented programming. In Niccolò Veltri, Nick Benton, and Silvia Ghilezan, editors, *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, pages 14:1–14:13. ACM, 2021.

[HKMN95] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS*, volume 95, pages 95–107, 1995.

[KLMR90] Herbert Kuchen, Rita Loogen, Juan José Moreno-Navarro, and Mario Rodríguez-Artalejo. Graph-based implementation of a functional logic langugage. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 432 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 1990.

[MKLR90] Juan José Moreno-Navarro, Herbert Kuchen, Rita Loogen, and Mario Rodríguez-Artalejo. Lazy narrowing in a graph machine. In Hélène Kirchner and Wolfgang Wechler, editors, *Algebraic and Logic Programming, Second International Conference, Nancy, France, October 1-3, 1990, Proceedings*, volume 463 of *Lecture Notes in Computer Science*, pages 298–317. Springer, 1990.

[OR14] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014.

[RBD+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming: the oz experience. *Theory Pract. Log. Program.*, 3(6):715–763, 2003.

[WK22] Hendrik Winkelmann and Herbert Kuchen. Constraint-logic object-oriented programming on the java virtual machine. In Jiman Hong, Miroslav Bures, Juw Won Park, and Tomás Cerný, editors, *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*, pages 1258–1267. ACM, 2022.

[WK23] Hendrik Winkelmann and Herbert Kuchen. Constraint-logic object-oriented programming with free arrays of reference-typed elements via symbolic aliasing. In Hermann Kaindl, Mike Mannion, and Leszek A. Maciaszek, editors, *Proceedings of the 18th International Conference on Evalu-*

ation of Novel Approaches to Software Engineering, ENASE 2023, Prague, Czech Republic, April 24-25, 2023, pages 412–419. SCITEPRESS, 2023.

[WTK22] Hendrik Winkelmann, Laura Troost, and Herbert Kuchen. Constraint-logic object-oriented programming for test case generation. In Jiman Hong, Miroslav Bures, Juw Won Park, and Tomás Cerný, editors, *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*, pages 1499–1508. ACM, 2022.

Towards a Holistic Case for Comparing Language Product Line Approaches

Thomas Kühn[1]

Software Engineering and Programming Languages Group
Martin-Luther University Halle-Wittenberg
D-06120 Halle, Germany
thomas.kuehn@informatik.uni-halle.de

Abstract. In recent years, research on language product line (LPL) engineering has emerged. Building on the ideas of modular compiler construction and software product line engineering (SPLE), LPLs enable language users to choose and pick language features from which a corresponding compiler, interpreter and/or integrated development environment is composed. While several LPL engineering approaches showcased their general applicability to individual cases, most only considered composition at the concrete or abstract syntax level, avoiding the complexity of code generation. Thus, it is currently impossible to adequately compare different LPL engineering approaches. To remedy this, I aim to establish a suitable, holistic language family of fully functional compilers. Moreover, I aim to provide a method for comparing different LPL approaches using the goal question metric approach. In this paper, I present initial ideas and work-in-progress results.

Keywords: Compiler Construction · Software Product Lines · Language Product Lines · Empirical Evaluation Method

1 Introduction

Computer scientists in the field of compiler construction generally agree that the development of programming languages is inherently complex. Moreover, as most of the compiler is generated by compiler construction tools (or language workbenches [16]), reuse is done opportunistically by copying definitions, grammar fragments, attribute grammar equations or translation rules. This opportunistic reuse is more akin to recycling, than to the notion of a reusable component, i.e., *"a reusable unit encapsulating a potentially incomplete language definition [comprising] the realization of syntax and semantics of a (software) language."* [6, p. 243]. To improve reuse in compiler construction, researchers initially focused on modular compiler constructions [18, 23, 35, 38]. Still, many established compiler construction tools lack means to create reusable (language) components. This changed in the past decades, as compiler construction tools and language workbenches improved providing means for systematic reuse of language components, e.g.[5, 10, 12, 17, 47, 49]. Building on the ideas of modular compiler construction and systematic reuse, more recently, researchers coined

the term *language product line* (LPL), e.g. [11, 27, 51], to denote the systematic development of a set of compilers/interpreters for a family of languages by composing reusable language components. Adopting ideas from *software product line* (SPL) engineering, language users should be able to choose and pick needed language features[1] from which a corresponding compiler/interpreter and *integrated development environment* (IDE) is constructed. As a result, several LPL approaches have been published indicating their applicability and functional suitability by showcasing their application to individual language families. Naturally, the presented approaches differ in scope and level of abstraction; i.e., while many approaches focus on composing the concrete or abstract syntax only, e.g. [26, 28, 29, 51], few include the intermediate or machine code generation, for instance, [6]. Moreover, while some approaches consider families of general purpose languages, e.g. [27, 48], it is currently impossible to systematically compare the different LPL engineering approaches.

To remedy this, I aim to design an evaluation method to compare different LPL engineering approaches employing a suitable language family as a common case.[^1.0] To achieve this, I must answer the following research questions:
(RQ1) What comprises a holistic family of programming languages for comparing LPL engineering approaches?
(RQ2) What are suitable goals, questions, and metrics for evaluating an LPL engineering approach?

In essence, RQ1 recognizes that many approaches employ either limited programming languages or limit the scope of the LPL to generate code of a host programming language. While LPLs of current programming languages are undoubtedly realistic cases employing a large variety of language concepts, the huge effort required to create a corresponding LPL makes their application for a comparative case study infeasible. Similarly, a custom DSL or the family of state-machine languages [9, 43, 45] is unsuitable as they often lack language concepts found in programming languages, e.g., coercion, recursion or nested scopes. While a small sample language might not be realistic, it might still be suitable if it incorporates the same challenging language concepts. Consequently, I will consider a family of holistic programming languages that is small yet able to *"illustrate the fundamental problems of compiler construction, but avoids the uninteresting complications"* [50, p.319], namely the family of *LAX* languages. While a suitable language family as a common case is imperative, RQ2 acknowledges that a suitable evaluation method for comparing LPL engineering approaches must define which aspects of an LPL's development and implementation are measured and how these aspects provide evidence for the quality of the LPL engineering approach under study. Consequently, I will employ the *goal question metric* (GQM) approach [7] to define comparison goals, evaluation questions, and corresponding metrics. Answering both questions, permits me to establish a suitable evaluation method for comparing LPL engineering approaches.

Please note that I present initial ideas and work-in-progress results.

[1] Following Vacchi and Cazzola [44], language features are either language constructs, e.g., *if then else*, or language concepts (without concrete syntax), e.g., *recursion.*

2 Background

Henceforth, I provide a brief overview of the main concepts and history of modular compiler construction, software product line (SPL) engineering, and language product line (LPL) engineering. Furthermore, I will discuss contemporary approaches for comparing compiler construction tools and language workbenches.

2.1 Modular Compiler Construction

While modular and incremental development of software were initially emphasized as crucial properties by Liskov [31] and Basil and Turner [4], respectively, the idea of modular development of compilers was initially discussed by Ganzinger [18]. He proposed a method for modular descriptions of compilers based on ideas from modular algebraic specifications, abstract datatypes, and attribute grammars. While rudimentary, he introduced *compiler modules* that decorate nodes of parse trees with semantic information. Later, Kastens and Waite [23] expand on this idea by introducing *attribution modules* that permit the modular and reusable specification of attribute grammars utilizing inheritance. These could be employed to modularize the different phases of the compilation process, e.g., parsing, semantic analysis, and code generation, to permit the modular specification of a compiler. Mernik and Žumer [35] recognized that while compilers are usually modularized along the compilation phases, these modules *"cannot be easily reused or extended in other language specifications"* [35, p. 2]. Building on the ideas of Kastens and Waite [23] and Ganzinger [18], they argue that languages should be extended along language features. They argue that this can be achieved by extending all compilation phases using attribute grammars and inheritance in a concerted effort. In essence, all extensions contributing to a specific language feature form a *compiler module*. Based on these ideas, a plethora of approaches emerged that go beyond the limitation of inheritance based extensions by means of, e.g., *model-based language development* [46], grammar inheritance and embedding [25, 38], or generalized *language components* [44]. The latter two will be discussed in more detail in subsection 2.3.

Please note that modular compiler verification, e.g. [36], is beyond the scope of this paper.

2.2 Software Product Line Engineering

Software engineers striving for ever-increasing modularity and reusability, led to the research area of *software product line* (SPL) engineering [40]. In this area, notions of product variants, product families, and product lines are adopted from the automotive industry and applied to software development. According to Northrop [37], an SPL *"is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."* In essence, an SPL is a family of software systems whose commonalities and differences are managed by means of features, such

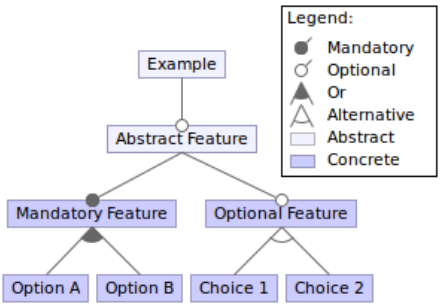


Fig. 1: An illustrative example of a feature model [28].

that given a feature selection, a variant of the software system can be derived from a common set of software modules (or components).

To represent the variability of an SPL, Kang et al. [22] introduced the notion of feature models. As illustrated in Figure 1, a feature model is a tree whose nodes are features.[2] In detail, a feature can have multiple children, whereas each child can be either *Optional* or *Mandatory*. Moreover, an *Or* group (filled pie slice) denotes that at least one of the child features must be selected, whereas an *Alternative* group (white pie slice) denotes that only exactly one of the child features must be selected. Furthermore, *Concrete* features are mapped to implementation artifacts, while *Abstract* features only improve the tree's readability. Last but not least, a feature model can be supplemented by a list of predicate logic expressions with features as atoms, to allow for specifying arbitrary cross-tree constraints. Given a feature model, a feature configuration denotes a set of selected concrete features. In turn, a feature configuration is considered valid if it fulfills the constraints represented by the feature model and cross-tree constraints. The feature configuration $\{\mathsf{Mandatory}, \mathsf{OptionA}\}$ would be a valid feature configuration of the feature model in Figure 1, as it selects the mandatory feature and at least one of its children.

Recently, researchers in the area of SPL engineering highlighted the linguistic distinction between variability in space and variability in time [2]. The former refers to the existence of a software system in different **variants** at the same time [40, Def. 4-6]. In contrast, the latter refers to the existence of different **versions** of the software system valid at different times [40, Def. 4-5]. Besides these, Ananieva et al. [2] describe variability in space and time to refer to the existence of a software system in different **variants and versions** valid at the same time.

2.3 Language Product Line Engineering

Inspired by SPL engineering and the advances in modular compiler construction, researchers proposed *language product line* (LPL) engineering [51] to per-

[2] In this paper, I will employ the notation used within FeatureIDE [33].

mit the feature-oriented development of compilers or interpreters for families of *domain-specific languages* (DSLs), e.g., [14, 28, 29, 51], and *general purpose programming languages* (GPLs), e.g., [8, 13].[3] While researchers focus on language variants, practitioners struggle with language versions that lead to ever-increasing complexity in compilers when required to support all previous language versions as well. Regardless of the cause for variability, LPL engineering approaches, e.g., [30, 44], structure a family of languages along *language features*. A language feature is any user-relevant language construct, e.g., `while` loops or `switch case`, and language concept, e.g., recursion or exceptions. Similar to an SPL, given a selection of language features, an LPL derives a compiler or interpreter for a language variant.[4] In contrast to SPL, however, there exist two fundamentally different LPL development approaches [26]. *Top-down* LPLs design a feature model of the language family first and implement corresponding language components afterward. By contrast, *bottom-up* LPLs implement annotated language components first and derive the feature model from the annotations and components' dependencies. While *bottom-up* LPLs are more flexible, due to the complexity and high coupling of language components, the derived feature model might be overly restrictive, requiring the renaming of open non-terminals to derive a viable language variant [27].

2.4 Contemporary Comparisons of Compiler Construction Tools and Language Workbenches

While the various LPL engineering approaches highlighted in subsection 2.3 employed case studies illustrate their applicability, only a few employ a common family of languages. Although many approaches employ a DSL to specify state machines as an illustrative case, e.g. [9, 43, 45], their solutions are not suitable for comparison as each approach only focuses on showcasing functional suitability and applicability. Thus far, only Kühn and Cazzola [26] have presented a comparative study of two different LPL engineering approaches, comparing a top-down and a bottom-up LPL approach. Employing role-based extensions to a modularized Java parser and printer, this comparison only compares the syntax analysis phase. As a result, none of these cases cover all aspects of a compiler, from syntax analysis down to code generation. When considering language workbenches themselves, Erdweg et al. [16] present a qualitative comparison of language workbenches, focusing on their capabilities. A quantitative comparison of different language workbenches was presented by Erdweg et al. [15], who tasked experts with implementing a small language for questionnaires. While this was the largest language workbench comparison, Kelly [24] correctly criticized their results for only comparing lines of code and coverage of capabilities. Kelly [24] argues that neither lines of code, user satisfaction, development time nor development costs are suitable metrics for comparing language implementations.

[3] Méndez-Acuña et al. [34] provides a comprehensive systematic literature review of LPL engineering approaches.

[4] Note that many LPL approaches additionally generate the IDE for the language variant, as well [17].

Aside from language workbenches, Ortin et al. [39] compared two parser generators in a compiler construction course, still focusing on the syntax analysis phase and measuring development time and user satisfaction. Again, these comparisons did not consider all aspects of a compiler and employed questionable metrics. While the comparison of compiler construction tools (or language workbenches) is rarely considered, many researchers have compared different compilers for the same language, e.g., [1, 41], yet most of them focus on the performance of the compiler and/or the generated machine code. While performance is an objective measurement, I argue that compilation performance is not a critical property of LPL engineering approaches, at least not at this stage of their maturity.

3 Towards a Case Study for Language Product Line Engineering

As I previously established, contemporary comparisons of LPL approaches lack a common programming language as a case as well as suitable properties measured for comparison. Thus, to design an evaluation method[5] for the comparison of LPL engineering approaches, we require a suitable family of programming languages as a case as well as a method for the analysis of individual LPL realizations based on the *goal question metric* (GQM) paradigm [7].

3.1 Towards LAX as a Language Product Line

Selection Criteria As existing case studies for comparing LPL engineering approaches limit themselves by excluding machine code generation, I aim to identify other programming languages, from which a suitable family of programming languages can be derived. To select a suitable programming language candidate, I consider the following selection criteria:

i) The language shall comprise the typical language constructs of programming languages, e.g., primitive types, expressions, control structures, variables, functions, or language concepts, e.g., type checking, coercion, name resolution, and recursion.
ii) The language shall be statically typed and reject any program with errors in its syntax and semantics.
iii) The language shall be compiled to machine code, e.g., RISC machine code, rather than source code or object code, to consider all phases of a compiler.
iv) It should be feasible for a single researcher to implement the language in a limited time frame.
v) The language specification shall be concise, complete, and publicly available.

In essence, these criteria ensure that the selected family of languages includes the fundamental problems of compiler construction (i), requires checking both a

[5] This evaluation method would be correctly classified as case study according to the empirical standards of software engineering [42].

program's syntax and semantics (ii), and copes with the generation of machine code (iii). In addition to that, I maintain that for a language to be implemented for a case study, researchers must be able to feasibly realize its compiler, i.e., it must be manageable to implement the compiler in a month (v). Last but not least, the specification of the language should be short, easy to understand, and freely available.

LAX as a Programming Language Candidate Using these selection criteria, I have selected the Sample Programming Language (LAX), designed by Waite and Goos [50], as a suitable candidate. In accordance with the five selection criteria, (i) LAX was specifically designed to encompass the fundamental concepts of a programming language without uninteresting complications. The language constructs range from primitive types and expressions via records and arrays to pointers and functions, whereas language concepts include, among others, coercion, recursion, and nested scopes. Second, (ii) LAX has a simple static type system and explicitly specifies which errors must be detected during semantic analysis. Third, (iii) LAX's semantics is still simple enough to be easily translatable to RISC machine code, i.e., MIPS [21]. This, in turn, requires LPL approaches to consider code generation. Fourth, (iv) LAX is a concise language that should be easy to implement, especially, for researchers in the field of compiler construction. While the development time might vary, I assume that a LAX compiler could be developed by one person in a month. Last but not least, (v) the language specification of LAX is concise, complete, and publicly available [50, Appendix A]. The upshot of all this is that LAX is a suitable candidate if a language family can be derived from it.

Now, the reader might object that LAX is not an object-oriented language. While I concede that LAX does not support inheritance or polymorphic dispatch, the language features of instantiable records and procedure/function pointers are at the core of every object-oriented language, albeit rarely visible. The challenges for machine code generation, however, remain the same.

Incremental Compiler Construction of LAX When designing a family of languages from an existing programming language, I followed a hybrid LPL approach instead of a pure top-down or bottom-up approach [Cazzola16i]. In particular, instead of decomposing an existing LAX compiler into language components, I followed a reactive SPL approach [3, Cha. 2.4], starting from a feature-minimal LPL and incrementally adding new language features. Fortunately, this process was aided by the existence of an incrementally developed LAX compiler, initially developed by W. Zimmermann in 2015.

Applying and teaching incremental compiler construction, as proposed by Basil and Turner [4], W. Zimmerman devised a corresponding practical course at Martin-Luther University Halle-Wittenberg.[6] Within this course, students

[6] The module description is available here: https://studip.uni-halle.de/dispatch.php/shared/modul/description/8b1a2667b293f0e4d61b7a524277e842/?display_language=EN

Table 1: Overview of the increments for implementing LAX derived from the practical course on compiler construction by W. Zimmermann.

#	Introduced Language Features	#	Introduced Language Features
0.0	expressions with integer constants	0.23	`switch case` expressions
0.1	integer addition `+`	0.24	`while` loops
0.2	integer subtraction `-`	0.25	`for` loops
0.3	integer multiplication `*`	0.26	declaration of `record` types
0.4	integer division `/`	0.27	field access of `record` types
0.5	unary `+` (integer)	0.28	object instantiation `new`
0.6	unary `-` (integer)	0.29	pointer types `ref`
0.7	expressions in brackets	0.30	pointer comparison `==`
0.8	declaration of constants and read access	0.31	dereference `^` operator
0.9	declaration lists	0.32	variables with 1-dimensional `array` types and array access
0.10	datatype `boolean`	0.33	variables with n-dimensional `array` types and array access
0.11	comparison operators `<` and `>`	0.34	general `array` types
0.12	equality operator `=`	0.35	`procedure`s without parameters and proc. call
0.13	declaration of variables and read access	0.36	one proc. parameter with atomic type and proc. call
0.14	assignment	0.37	one proc. parameter with composite type and proc. call
0.15	assignment lists	0.38	multiple proc. parameters and proc. call
0.16	datatype `real`, operators, and coercion	0.39	function with atomic return type
0.17	real division (with `/`)	0.40	function with composite return type
0.18	blocks	0.41	procedure/function pointer, assignment and comparison
0.19	`if then else` expressions	0.42	deproduction of procedure/function pointer
0.20	boolean negation `not`	0.43	argument passing for procedure/function pointers
0.21	conjuction `and` with short evaluation	0.44	procedure/function pointer as return types
0.22	disjunction `or` with short evaluation	0.45	unconditional jumps `goto`

are tasked to incrementally develop a compiler for LAX employing the compiler construction tool *Eli* [20]. Initially, the students are provided with a working language version 0.0 with all Eli specifications and auxiliary functions written in C. During the course, they implement a new language version each week by implementing a language feature into the previous language versions. The course follows the iteration plan, outlined in Table 1. These increments were meticulously planned and refined by W. Zimmerman, such that later increments require minimal changes to the existing implementation. Although most students only reach language version 0.23, for each language version there exists a reference implementation that was incrementally developed as well as a staggering amount of 708 980 096 positive and negative test cases programmatically generated for each increment.

I employed both the iteration plan (cf. Table 1) and a file diffing tool to investigate commonalities and differences between the 46 language versions. To eliminate the implicit dependency from a language version to all previous versions, I determined to which of the previous language versions a dependency exists. The resulting partial order of increments represents both the dependencies among language features from which I derive both a feature model and candidates for language components from which a reference LAX LPL can be implemented.

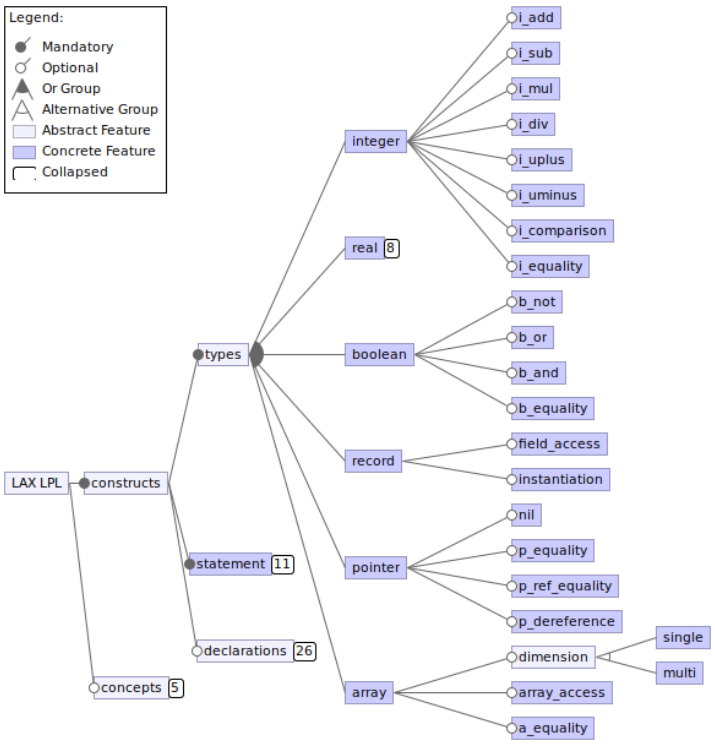


Fig. 2: Feature model for the types of the family of LAX languages without cross-tree constraints.

Variability Model of the Family of LAX Languages Based on the analysis of the dependencies among the language features, I was able to design a feature model for the family of LAX programming languages. The resulting feature model is depicted in Figure 2 and Figure 3. The feature model encompasses 78 concrete and 8 abstract features. In detail, Figure 2 highlights that a LAX variant must at least select one of the types. Once a type is selected, the corresponding literals, semantics, and machine code generation should be added. The child features of each of the types denote operators that can occur in an expression of that type. For brevity, I omitted the 8 child features of *real*, as this datatype has the same operators as the *integer* feature. In contrast, Figure 3 emphasizes the remaining language constructs and concepts. The language constructs cover different kinds of statements and declarations. The former covers both conditional expressions and loops, whereas the latter covers declarations for the corresponding types, e.g., variable, reference, or procedure declarations. As the feature model was not expressive enough to capture all identified feature dependencies, I added 14 cross-tree constraints, listed in Figure 4. The constraints (1)–(6), for instance, ensure that if a comparison operator, a conditional

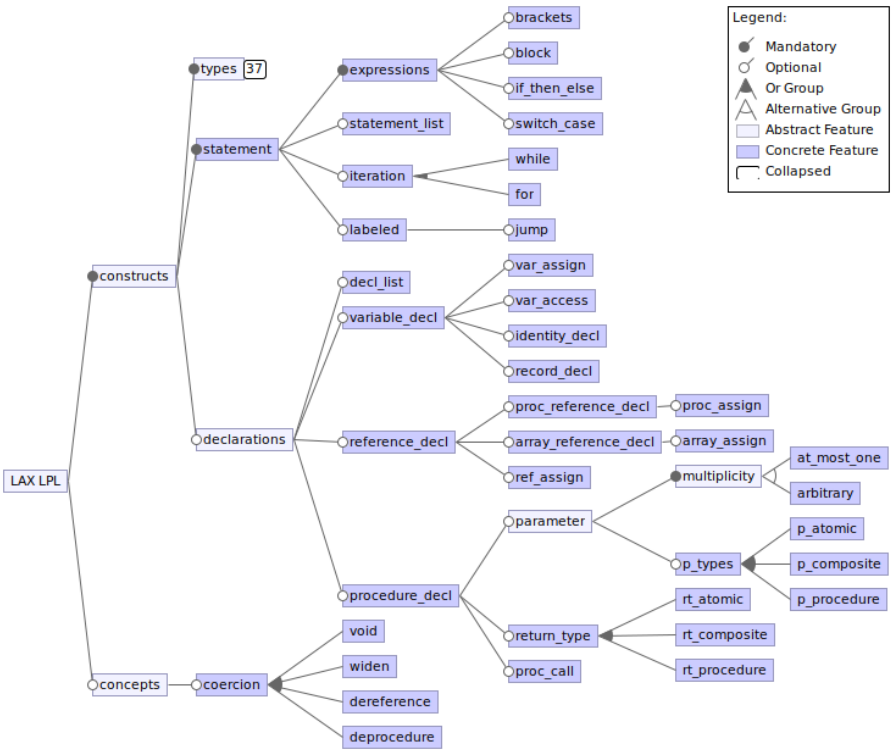


Fig. 3: Feature model for the language constructs and concepts of the family of LAX languages without cross-tree constraints.

expression, or a loop statement is selected, then the required primitive datatype is selected as well. Similarly, the constraints (8–12) ensure that the types exist if a corresponding declaration is selected. In sum, the variability described by the feature model and cross-tree constraints is extensive, yielding more than 500000 valid feature configurations.[7] The feature model not only covers the versions of LAX, discussed in section 3.1, but also a plethora of language specializations, e.g., LAX with functions yet only the *boolean* datatype and corresponding operators and expressions. Consequently, LAX version 0.45 corresponds to a feature complete variant of the family of LAX languages, namely the original LAX programming language.

Towards a Reference Implementation of a LAX Language Product Line Up to this point, I have only captured the feature model and candidates for language components, but not a decomposition of the language. While one

[7] This is an estimation computed by FeatureIDE [33].

$$\text{i\_comparison} \vee \text{i\_equality} \Rightarrow \text{boolean} \quad (1)$$
$$\text{r\_comparison} \vee \text{r\_equality} \vee \text{a\_equality} \Rightarrow \text{boolean} \quad (2)$$
$$\text{p\_equality} \vee \text{p\_ref\_equality} \Rightarrow \text{boolean} \quad (3)$$
$$\text{if\_then\_else} \vee \text{while} \Rightarrow \text{boolean} \quad (4)$$
$$\text{switch\_case} \Rightarrow \text{integer} \quad (5)$$
$$\text{for} \Rightarrow \text{integer} \quad (6)$$
$$\text{instantiation} \vee \text{dereference} \Rightarrow \text{pointer} \quad (7)$$
$$\text{block} \Rightarrow \text{declarations} \quad (8)$$
$$\text{record} \Leftrightarrow \text{record\_decl} \quad (9)$$
$$\text{proc\_reference\_decl} \Rightarrow \text{procedure\_decl} \quad (10)$$
$$\text{reference\_decl} \Rightarrow \text{pointer} \quad (11)$$
$$\text{array\_reference\_decl} \Rightarrow \text{array} \quad (12)$$
$$\text{widen} \Rightarrow \text{integer} \wedge \text{real} \quad (13)$$
$$\text{deprocedure} \Rightarrow \text{return\_type} \quad (14)$$

Fig. 4: Additional cross-tree constraints for the feature model shown in Figure 2 and Figure 3.

might think that this suffices as a case study, for a suitable quantitative evaluation method, a reference implementation is required. This will permit the generation of the different language variants, programs for these language variants, as well as corresponding machine code. This is a considerable undertaking, especially, as I intend to avoid the application of an existing language workbench suitable for LPLs, e.g. [5, 17]. To mitigate the language development effort and the introduction of bias, I intend to reuse most of the Eli specifications and implementations of the incrementally developed LAX version (cf. section 3.1). Moreover, I will employ one of the first variability implementation techniques, i.e., preprocessor annotations [3, Cha. 5.3]. While this technique does not permit the creation of language components, it can be uniformly employed to implement variability throughout all Eli-specific files and auxiliary implementations written in C. I can simply employ the C-preprocessor to derive a compiler for the language variant by defining the involved language features. I aim to complete the development of the reference implementation of the LAX LPL by the end of next year.

Besides that, I focus on the extension of the existing black-box test suite generated for the different language versions. As it is infeasible to exhaustively test all valid language configurations, I am to adapt the test suite to generate test cases for specific language features in a given language context, i.e., a valid language configuration. In detail, I will annotate the language features present in each generate test case as well as whether it is a positive or negative test for this feature. This, in turn, permits me to employ SPL testing methods, such as

sample based testing that is based on language features [3, Cha. 10.3.2] or delta-based testing that is based on language increments [32]. The resulting black-box test suite mirrors the variability of the LAX LPL to permit the creation of tailored positive and negative test cases.

3.2 Goals, Questions, and Metrics for Comparing Language Product Line Engineering Approaches

The family of LAX language represents the common case that should be realized using the various LPL engineering approaches. However, to evaluate each individual realization and subsequently compare different realizations, suitable measurements must be defined beforehand. Instead of blindly picking measurements, I followed the *goal question metric* (GQM) approach [7], to define the goals of the evaluation, the evaluation questions, and corresponding measurements (or metrics) taken from an LPL realization.

The GQM approach was promoted by Caldiera and Rombach [7] to improve the collection of empirical data in software engineering research. Their approach emphasizes that the evaluation of a system under study should be driven by **goals**, **questions**, and **metrics**. Goals clearly state the particular goal of the evaluation wrt. the system under study. Assigned to each goal are questions that characterize the viewpoint of the system under study to achieve the evaluation goal. In turn, for each question, the approach selects a number of metrics that subjectively or objectively quantify a property of the system under study relevant to answering the evaluation question. Employing this approach ensures a top-down, goal-oriented evaluation of the systems under study [7].

For the proposed comparison of the LPL realizations of the LAX language family, I propose the following GQM plan:

- **G1:** Variability coverage of the LPL realization
  - **Q1:** Does the LPL realization correctly derive compilers for all language variants of the LAX family?
    - **M1:** Fraction of valid configurations for which a compiler could be derived.
    - **M2:** Sum of false included language features in derived compilers.
- **G2:** Syntactic validity of derived compilers
  - **Q2:** Does the derived compiler only accept programs belonging to a language variant?
    - **M3:** Fraction of accepted (semantic correct) programs belonging to the language variant.
    - **M4:** Fraction of rejected programs of the language variant with injected syntax errors.
    - **M5:** Number of unselected language features that are accepted by the derived compiler when occurring in a program of another language variant.
- **G3:** Semantic validity of derived compilers
  - **Q3:** Does the derived compiler correctly check the static semantics of programs belonging to a language variant?

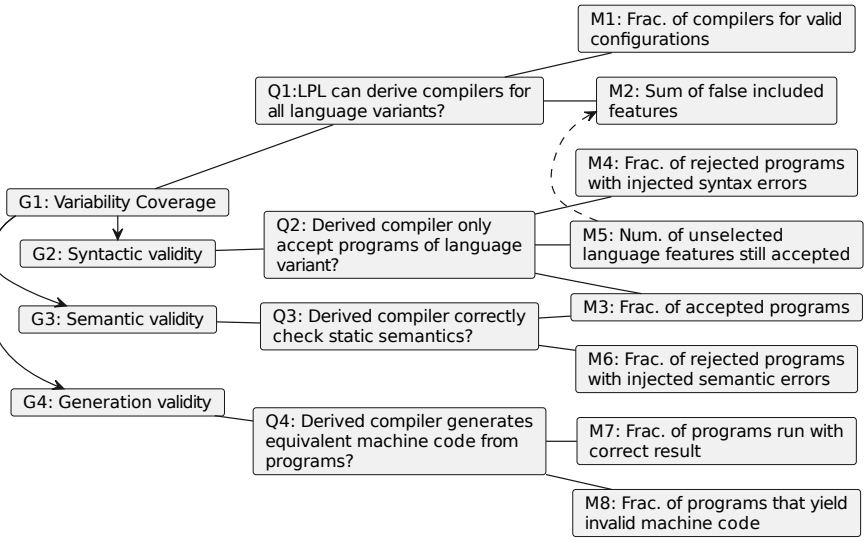


Fig. 5: Overview of GQM plan for the evaluation of LPL realizations.

    - ∗ **M6:** Fraction of rejected programs of the language variant with injected semantic errors.
- **G4:** Generation validity of derived compilers
  - **Q4:** Does the derived compiler generate equivalent machine code for programs belonging to a language variant?
    - ∗ **M7:** Fraction of programs of the language variant whose generated MIPS code yields the correct result.
    - ∗ **M8:** Number of programs of the language variants that when compiled yield invalid MIPS machine code.

This plan, summarized in Figure 5, focuses on four main evaluation goals, whereas the first goal investigating whether the LPL realization covers the variability of the feature model (G1) leads to the three remaining goals investigating the correctness of the derived compiler for a particular language variant wrt. the syntax analysis (G2), semantic analysis (G3) and finally machine code generation (G4). For each goal, a corresponding evaluation question considers the programs of a language variant (with and without errors). Simply put, I will employ the test suite for the LAX LPL to derive suitable test samples to answer each evaluation question. Finally, the metrics M1 to M8 represent objective measurements that ensure that the LPL realization (under study) correctly realizes the family of LAX programming languages. Note that the metric M2 accumulates the results of metric M5.

Please note that I intentionally left out objective performance metrics, e.g., measuring the compiler derivation time, compilation performance or performance of the resulting programs, as I regard this as an unimportant property,

especially at the current maturity of LPL engineering approaches. Granted, these measurements could be easily included into the GQM plan, including counting the number of generated MIPS instructions. Similarly, I avoided the temptation to compare the lines of code or implementation time of LPL realizations. Following Kelly [24] argument, these are unsuitable for quantitative comparisons unless an objective metric can be found.

4 Conclusion

In this paper, I have presented my preliminary results towards the creation of an evaluation method for comparing LPL engineering approaches. I aimed to both identify a suitable language family as a common case and design a quantitative evaluation method for such a comparison. I have identified LAX as a suitable candidate for a holistic family of programming languages (RQ1). For this candidate, I have investigated the dependencies among language features to derive a feature model for the family of LAX programming languages. In addition, I discussed how I intend to develop a reference implementation of a LAX LPL using Eli and the C-preprocessor. Afterward, I have proposed a method for comparing LPL realizations based on the goal question metric (GQM) approach (RQ2). In detail, I selected four evaluation goals and corresponding evaluation questions that aim to evaluate whether the LPL and derived compilers for each language variant correctly implement the family of LAX programming languages. To answer the evaluation questions, I provided eight metrics that all consider programs of individual language variants. Although I argued that both the case and the GQM plan are suitable, I concede that objects, inheritance, and polymorphism are language features missing in LAX and that the presented GQM plan neglects all indicators for performance and implementation effort.

Consequently, in the future, I will investigate the language features of Sather-K [19] to define reusable language components for objects, inheritance, and polymorphism that could be added to LAX. Moreover, I study the literature looking for information theoretic metrics for code complexity as an objective measure of development effort. Last but not least, once the reference implementation of the LAX LPL is completed, I will start evaluating existing LPL engineering approaches, e.g. [5, 17].

Bibliography

[1] Åkesson, J., Ekman, T., Hedin, G.: Implementation of a modelica compiler using jastadd attribute grammars. Science of Computer Programming 75(1-2), 21–38 (2010)

[2] Ananieva, S., Greiner, S., Kühn, T., Krüger, J., Linsbauer, L., Grüner, S., Kehrer, T., Klare, H., Koziolek, A., Lönn, H., Krieter, S., Seidl, C., Ramesh, S., Reussner, R., Westfechtel, B.: A conceptual model for unifying variability in space and time. In: 24th ACM Conference on Systems and Software Product Line (SPLC'20): Volume A-Volume A. SPLC '20, Association for

Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3382025.3414955

[3] Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Sof@inproceedingslochau2012incremental, title=Incremental model-based testing of delta-oriented software product lines, author=Lochau, Malte and Schaefer, Ina and Kamischke, Jochen and Lity, Sascha, booktitle=Tests and Proofs: 6th International Conference, TAP 2012, Prague, Czech Republic, May 31–June 1, 2012. Proceedings 6, pages=67–82, year=2012, organization=Springer tware Product Lines: Concepts and Implementation. Springer (2013)

[4] Basil, V.R., Turner, A.J.: Iterative enhancement: A practical technique for software development. IEEE Transactions on Software Engineering SE-1(4), 390–396 (1975)

[5] Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: Systematic composition of independent language features. Journal of Systems and Software 152, 50–69 (2019)

[6] Butting, A., Wortmann, A.: Language Engineering for Heterogeneous Collaborative Embedded Systems, pp. 239–253. Springer International Publishing, Cham (2021), https://doi.org/10.1007/978-3-030-62136-0_11

[7] Caldiera, V.R.B.G., Rombach, H.D.: The goal question metric approach. Encyclopedia of Softw. Eng. pp. 528–532 (1994)

[8] Cazzola, W., Olivares, D.M.: Gradually Learning Programming Supported by a Growable Programming Language. IEEE Transactions on Emerging Topics in Computing 4(3), 404–415 (Sep 2016), special Issue on Emerging Trends in Education

[9] Crane, M.L., Dingel, J.: UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In: 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05). pp. 97–112. Lecture Notes in Computer Science 3713, Springer, Montego Bay, Jamaica (2005)

[10] Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Melange: a Meta-Language for Modular and Reusable Development of DSLs. In: Di Ruscio, D., Völter, M. (eds.) 8th International Conference on Software Language Engineering (SLE'15). pp. 25–36. ACM, Pittsburgh, PA, USA (Oct 2015)

[11] Delaware, B., Cook, W., Batory, D.: Product lines of theorems. In: 2011 ACM international conference on Object oriented programming systems languages and applications. pp. 595–608 (2011)

[12] Ekman, T., Hedin, G.: The JastAdd System — Modular Extensible Compiler Construction. Science of Computer Programming 69(1-3), 14–26 (Dec 2007)

[13] Erdweg, S., Giarrusso, P.G., Rendel, T.: Language Composition Untangled. In: Sloane, A.M., Andova, S. (eds.) 12th Workshop on Language Description, Tools, and Applications (LDTA'12). ACM, Tallinn, Estonia (Mar 2012)

[14] Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: Library-Based Syntactic Language extensibility. In: 26th ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA'11). pp. 391–406. ACM, Portland, Oregon, USA (Oct 2011)
[15] Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R.c., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G.H., van der Woning, J.: The state of the art in language workbenches. In: 6th International Conference on Software Language Engineering (SLE'13). pp. 197–217. Springer International Publishing (2013)
[16] Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van del Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. Computer Languages, Systems and Structures 44, 24–47 (Dec 2015)
[17] Favalli, L., Kühn, T., Cazzola, W.: Neverlang and featureide just married: Integrated language product line development environment. In: 24th ACM Conference on Systems and Software Product Line (SPLC'20): Volume A-Volume A. pp. 1–11 (2020)
[18] Ganzinger, H.: Increasing modularity and language-independency in automatically generated compilers. Science of Computer Programming 3(3), 223–278 (1983)
[19] Goos, G.: Sather-k – the language. Software-Concepts and Tools 18(3), 91–109 (1997)
[20] Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M.: Eli: A complete, flexible compiler construction system. Communications of the ACM 35(2), 121–130 (1992)
[21] Kane, G., Heinrich, J.: MIPS RISC architectures. Prentice-Hall, Inc. (1992)
[22] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
[23] Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Informatica 31, 601–627 (1994)
[24] Kelly, S.: Empirical comparison of language workbenches. In: 2013 ACM Workshop on Domain-Specific Modeling. p. 33–38. DSM '13, ACM (2013), https://doi.org/10.1145/2541928.2541935
[25] Krahn, H., Rumpe, B., Völkel, S.: Monticore: a framework for compositional development of domain specific languages. International journal on software tools for technology transfer 12, 353–372 (2010)
[26] Kühn, T., Cazzola, W.: Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In: 20th International Software Product Line Conference (SPLC'16). pp. 50–59. ACM, Beijing, China (19th-23rd of Sep 2016)

[27] Kühn, T., Cazzola, W., Olivares, D.M.: Choosy and Picky: Configuration of Language Product Lines. In: Botterweck, G., White, J. (eds.) 19th International Software Product Line Conference (SPLC'15). pp. 71–80. ACM, Nashville, TN, USA (20th-24th of Jul 2015)
[28] Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., Aßmann, U.: A Metamodel Family for Role-Based Modeling and Programming Languages. In: 7th International Conference Software Language Engineering (SLE'14). pp. 141–160. Lecture Notes in Computer Science 8706, Springer, Västerås, Sweden (Sep 2014)
[29] Leduc, M., Degueule, T., Combemale, B.: Modular language composition for the masses. In: Proceedings of 11th International Conference on Software Language Engineering (SLE'18),. p. 47–59. SLE 2018, ACM, New York, NY, USA (2018), https://doi.org/10.1145/3276604.3276622
[30] Liebig, J., Daniel, R., Apel, S.: Feature-Oriented Language Families: A Case Study. In: Collet, P., Schmid, K. (eds.) 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13). ACM, Pisa, Italy (Jan 2013)
[31] Liskov, B.H.: A design methodology for reliable software systems. In: December 5-7, 1972, fall joint computer conference, part I. pp. 191–199 (1972)
[32] Lochau, M., Schaefer, I., Kamischke, J., Lity, S.: Incremental model-based testing of delta-oriented software product lines. In: Tests and Proofs: 6th International Conference, TAP 2012, Prague, Czech Republic, May 31–June 1, 2012. Proceedings 6. pp. 67–82. Springer (2012)
[33] Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G.: Mastering Software Variability with FeatureIDE. Springer (2017)
[34] Méndez-Acuña, D., Galindo, J.A., Degueule, T., Combemale, B., Baudry, B.: Leveraging software product lines engineering in the development of external dsls: A systematic literature review. Computer Languages, Systems & Structures 46, 206–235 (2016)
[35] Mernik, M., Žumer, V.: Incremental Programming Language Development. Computer Languages, Systems and Structures 31(1), 1–16 (Apr 2005)
[36] Müller-Olm, M.: Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction, vol. 1283. Springer-Verlag, Berlin Heidelberg (1997)
[37] Northrop, L.: Software product lines essentials. software engineering institute (2008), https://resources.sei.cmu.edu/asset_files/presentation/2008_017_001_24246.pdf
[38] Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: International Conference on Compiler Construction. pp. 138–152. Springer (2003)
[39] Ortin, F., Quiroga, J., Rodriguez-Prieto, O., Garcia, M.: Evaluation of the use of different parser generators in a compiler construction course. In: Information Systems and Technologies. pp. 338–346. Springer International Publishing, Cham (2022)
[40] Pohl, K., Böckle, G., van Der Linden, F.J.: Software product line engineering: foundations, principles and techniques. Springer Science & Business Media (2005)

[41] Poorhosseini, M., Nebel, W., Grüttner, K.: A compiler comparison in the risc-v ecosystem. In: 2020 International Conference on Omni-layer Intelligent Systems (COINS). pp. 1–6. IEEE (2020)
[42] Ralph, P., Baltes, S., Bianculli, D., Dittrich, Y., Felderer, M., Feldt, R., Filieri, A., Furia, C.A., Graziotin, D., He, P., Hoda, R., Juristo, N., Kitchenham, B.A., Robbes, R., Méndez, D., Molleri, J., Spinellis, D., Staron, M., Stol, K., Tamburri, D.A., Torchiano, M., Treude, C., Turhan, B., Vegas, S.: ACM SIGSOFT empirical standards. CoRR abs/2010.03525 (2020), https://arxiv.org/abs/2010.03525
[43] Tratt, L.: Domain Specific Language Implementation Via Compile-Time Meta-Programming. ACM Trans. Progr. Lang. Syst. 30(6), 31:1–31:40 (Oct 2008)
[44] Vacchi, E., Cazzola, W.: Neverlang: A Framework for Feature-Oriented Language Development. Computer Languages, Systems & Structures 43(3), 1–40 (Oct 2015)
[45] Vacchi, E., Cazzola, W., Pillay, S., Combemale, B.: Variability Support in Domain-Specific Language Development. In: Proceedings of 6th International Conference on Software Language Engineering (SLE'13). pp. 76–95. Lecture Notes on Computer Science 8225, Springer, Indianapolis, USA (27th-28th of Oct 2013)
[46] Völter, M.: Language and IDE Modularization and Composition with MPS. In: Lämmel, R., Saraiva, J.a., Visser, J. (eds.) 4th International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11). pp. 383–430. Lecture Notes in Computer Science 7680, Springer, Braga, Portugal (Jul 2011)
[47] Völter, M., Pech, V.: Language Modularity with the MPS Language Workbench. In: 34th International Conference on Software Engineering (ICSE'12). pp. 1449–1450. IEEE, Zürich, Switzerland (Jun 2012)
[48] Völter, M., Ratiu, D., , Schätz, B., Kolb, B.: mbeddr: an Extensible C-Based Programming Language and IDE for Embedded Systems. In: 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12). pp. 121–140. ACM, Tucson, AZ, USA (Oct 2012)
[49] Wachsmuth, G.H., Konat, G.D.P., Visser, E.: Language Design with the Spoofax Language Workbench. IEEE Software 31(5), 35–43 (Sep/Oct 2014)
[50] Waite, W.M., Goos, G.: Compiler construction. Springer Science & Business Media (1995)
[51] Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U.: Vml*–a family of languages for variability management in software product lines. In: Software Language Engineering: Second International Conference, SLE 2009. pp. 82–102. Springer, Denver, CO, USA (2010)

A Navigation Logic for Recursive Programs with Dynamic Thread Creation

Roman Lakenbrink[1], Markus Müller-Olm[1], Christoph Ohrem[1], and Jens Gutsfeld[1]

University of Münster
{roman.lakenbrink,markus.mueller-olm,christoph.ohrem,jens.gutsfeld}@uni-muenster.de

Abstract. Dynamic Pushdown Networks (DPNs) are a model for multithreaded programs with recursion and dynamic creation of threads. In this talk, we propose a temporal logic called Navigation Temporal Logic (NTL) for reasoning about the call- and return- as well as thread creation behaviour of DPNs. Using tree automata techniques, we investigate the model checking problem for the novel logic and show that its complexity is not higher than that of LTL model checking against pushdown systems despite a more expressive logic and a more powerful system model. The same holds true for the satisfiability problem when compared to the satisfiability problem for a related logic for reasoning about the call- and return-behaviour of pushdown systems. Overall, this novel logic offers a promising approach for the verification of recursive programs with dynamic thread creation.

Model Checking of Asynchronous Hyperproperties

Jens Gutsfeld, Markus Müller-Olm, and Christoph Ohrem

University of Münster
{jens.gutsfeld,markus.mueller-olm,christoph.ohrem}@uni-muenster.de

Abstract. Hyperproperties [3] have received increasing attention in the last decade due to their importance e.g. for security analyses. As traditional specification logics like LTL are unsuitable for the specification of hyperproperties, new *hyperlogics* were developed. Past hyperlogics like HyperLTL [2] have been developed with a synchronous semantics in which different traces are compared lockstepwise. While this approach may be suitable for the verification of hardware systems, this is not the case for software systems whose behaviour is inherently asynchronous. For example, when checking information-flow policies on concurrent programs, traces might only be required to be equivalent up to stuttering [5] and thus matching observation points on different traces are not perfectly aligned. Another drawback of past approaches to model checking of hyperproperties is that they have focussed on finite models which cannot capture many programs suitably due to the lack of a representation for the call stack.

In this talk, we present two temporal logics for asynchronous hyperproperties in order to address these problems. First, we present the temporal fixpoint calculus H_μ [4], the first fixpoint calculus that can systematically express hyperproperties in an asynchronous manner and at the same time subsumes the existing hyperlogic HyperLTL. We investigate the model checking problem for the logic on finite models which is in general highly undecidable due to the high expressive power of the logic. As a remedy, we propose approximate analyses that induce natural decidable fragments. Secondly, we present the logic mumbling H_μ, a sublogic of H_μ, that handles asynchronicity via a mechanism to identify relevant positions on traces. While the new logic is more expressive than a related logic presented recently by Bozzelli et al. [1], we obtain the same complexity of the model checking problem for finite state models. Beyond this, we solve the model checking problem of the logic for pushdown models by introducing a concept called *well-alignedness* that requires the traces under consideration to have a similar call-return behaviour. This approach thus not only complements the decidable fragments of H_μ, it also provides one of the first approaches to the verification of hyperproperties on infinite state systems.

References

1. Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of HyperLTL. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science,*

LICS 2021, Rome, Italy, June 29 - July 2, 2021, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470583.

2. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.
3. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010. URL: https://doi.org/110.3233/JCS-2009-0393, doi:10.3233/JCS-2009-0393.
4. Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. doi:10.1145/3434319.
5. Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*, page 29. IEEE Computer Society, 2003. doi:10.1109/CSFW.2003.1212703.

Improving Rust Mutation Testing using Static and Dynamic Program Analysis

Samuel Pilz

Compilers and Languages
Institute of Information Systems Engineering
TU Wien
e1327391@student.tuwien.ac.at

Abstract. Mutation testing is a powerful software testing method in which the program under test is seeded with artificial faults that are considered to be possible programming errors and should be discovered by a high-quality test suite. The cost of mutation testing is one of the most critical issues for its practical applications. In this thesis, we present the tool `muttest` for mutation analysis of Rust programs that improves quality and performance of mutation analysis compared to related systems.
This thesis proposes several methods for leveraging Rust language features and results of static analysis to implement mutation operators while preventing the generation of invalid mutations. Moreover, dynamic program analysis is used to perform weak mutation analysis and its results are utilized to improve the performance of strong mutation analysis. By relying only on stable features of Rust, we ensure best-possible compatibility of `muttest` with future versions of Rust. The experimental evaluation in this thesis shows that `muttest` has competitive performance and produces a high-quality mutation analysis report.

Keywords: software correctness · software testing · mutation testing · weak, strong mutation analysis · static, dynamic program analysis · Rust

Principal generics in Java–TX

Martin Plümicke[1,2] and Daniel Holle[1,3]

[1] Duale Hochschule Baden-Württemberg, Campus Horb, Department of Computer Science, Florianstraße 15, D–72160 Horb, Germany, https://www.dhbw-stuttgart.de/horb/forschung-transfer/forschungsschwerpunkte/typsysteme-fuer-objektorientierte-programmiersprachen
[2] pl@dhbw.de
[3] d.holle@hb.dhbw-stuttgart.de

Abstract. Java-TX (TX standing for **T**ype e**X**tended) is an extension of Java. The predominant new features are global type inference and real function types for lambda expressions. Since that time the type inference algorithm consists of two steps, the tree traversing and the type unification. During traversing of the abstract syntax tree type constraints are built. These type type constraints are solved by the type unification. The result of the type unification is a set of pairs of a set of remaining constraints consisting only of two type variables and a general type unifier. Since that time the remaining constraints are not considered in detail. In this paper we consider these constraints. The constraints become to bound type variables of the corresponding classes and its methods, respectively. The goal is that the classes and the methods become a principal type, respectively. In a first step each type variable is mapped to a class or a method. After that some bounds are added which are induced by method calls. Finally cycles and infima of the constraints are erased as they are not allowed in Java.

1 Introduction

Since version 1.5 the programming language Java has been extended by incorporating many features from functional programming languages. Version 1.5 [2] saw the introduction of generics. Generics are known as parametric polymorphism in functional programming languages.
In Java 8 [3] lambda expressions were introduced, but not real function types. The types of lambda expressions are defined as target types, which are functional interfaces (essentially interfaces with one method). In Java-TX we added real function types [6]
Local type inference was introduced in the versions five, seven, and ten. In Java 5.0 the automatic determination of parameter instance was introduced. In Java 7 the diamond operator was introduced. In Java 10, finally, the var keyword for types of local variables was introduced [1]. In Java-TX local type inference is enlarged to global type inference [5].
Our type inference algorithm consists since that time of two funtions *tree traversing* and *type unification*.

In this paper we add a third function *generate generics*. The basic idea is to transfer the remaining type variables of the type unification to type parameters (gererics) of the classes and the methods, respectively.
The paper is organizes as follows: In Section 2 we give a short summary of the type inference algorithm. In section 3 the function *generate generics* is introduced. Finally, we close with a summary and an outlook.

2 Global type inference for Java-TX

Global type inference allows us to leave out all type annotations. As in functional programming languages like Haskell, the compiler similarly determines a principal typing, such that Java-TX is statically typed as original Java. Let us first consider a simple example.

Example 1. Let the class `Fac` be given.

```
import java.lang.Integer;

class Fac {
    getFac(n){
        var res = 1;
        var i = 1;
        while(i<=n) {
            res = res * i;
            i++;
        }
        return res; }
}
```

It is the simple iterative implementation of the factorial function. The return and the argument type of `getFac` are left out. The type inference algorithm has to infer the types. The types are determined by the declaration of `res` and the operator *. In order to reduce the complexity of the type inference algorithm for overloaded operators in the same way as for overloaded methods, only types are inferred which are explicitly imported by the keyword **import**. Therefore in `getFac` only the type `java.lang.Integer` is inferred.

2.1 The type inference algorithm

The input of the type inference algorithm is the abstract syntax tree of the corresponding Java-TX class, without type annotations:

$$\mathsf{Class}(\, cl, \mathsf{extends}(\, ty\,), \overline{f}, \mathsf{Method}(\, m_1, \overline{v_1}, bl_1\,) \ldots \mathsf{Method}(\, m_n, \overline{v_n}, bl_n\,)\,)$$

The *result of Java-TX type inference* is fully typed Java-TX-class:

$$\begin{aligned}\mathsf{Class}(\, cl, \overline{gen_{cl}}, \mathsf{extends}(\, ty\,), \overline{fty\ f},\\ \mathsf{Method}(\, \overline{gen_1}, rty_1, m_1, \overline{v_1 : ty_1}, bl_1\,)\\ \ldots\\ \mathsf{Method}(\, \overline{gen_n}, rty_n, m_n, \overline{v_n : ty_n}, bl_n\,)\,)\end{aligned}$$

The type inference algorithm consists of three steps: *tree traversing*, *type unification*, and *generate generics*.
In this section we summarize briefly the first two steps. For more details we refer to [5,4,7]. The main topic of this paper is the third step *generate generics* which we consider in the next section.

Tree traversing: In a traversing of the abstract syntax tree, a type is mapped to each node of the methods' statements and expressions. If the corresponding types are left out, a type variable is mapped as type placeholder. Otherwise, the known type is mapped.
During the traversing a set of type constraints $\{\,\overline{ty \lessdot ty'}\,\}$ is generated. The constraints represent the type conditions as defined in the Java specification [3]. For more details see the function **TYPE** in [5].

Type unification: For the set of type constraints $\{\,\overline{ty \lessdot ty'}\,\}$ general unifiers (substitution) σ are demanded, such that

$$\overline{\sigma(\,ty1\,) \leq^* \sigma(\,ty'\,)}.$$

The result of the type unification is a set of pairs

$$(\{\,\overline{(T \lessdot T')}\,\}, \sigma),$$

where $\{\,\overline{(T \lessdot T')}\,\}$ is a set of remaining constraints consisting of two type variables and σ is a general unifier.
The type unification algorithm is given in [4,7]. There we proved that the unification is indeed not unitary, but finitary, meaning that there are finitely many general unifiers.

Let us consider the application of the type inference algorithm to the factorial example (Example 1). Note that we omit the `import` statements for the sake of readability in the further examples.

Example 2. First, we present the essential type variables which are mapped to nodes of the method `getFac`:

```
class Fac {
  N getFac(O n) {
    P res = 1;
    R i = 1;
     while((i::R) <= (n::O))::T  {
        (res::P)=((res::P)*(i::R))::U;
        (i::R)++;
      }
      return(res::P);
  }
}
```

The generated constraints are

$\{\,(P \lessdot N), (U \lessdot P),$
$(O \lessdot \texttt{java.lang.Number}), (R \lessdot \texttt{java.lang.Number}), (\texttt{java.lang.Boolean} \doteq T),$
$(\texttt{java.lang.Integer} \doteq U), (R \lessdot \texttt{java.lang.Integer}), (P \lessdot \texttt{java.lang.Integer})\,\}$

The result of type unification is given as:

$$\begin{aligned}\{\,(\emptyset, [&(U \mapsto \texttt{java.lang.Integer}),\\ &(P \mapsto \texttt{java.lang.Integer}), (R \mapsto \texttt{java.lang.Integer}),\\ &(O \mapsto \texttt{java.lang.Integer}), (N \mapsto \texttt{java.lang.Integer}),\\ &(T \mapsto \texttt{java.lang.Boolean})]\,\}\end{aligned}$$

In this example, no constraints consisting only of type variables remain. Furthermore, there is only one general unifier.
If we instantiate the type variables by the determined types, we get:

```
class Fac {
  Integer getFac(Integer n) {
    Integer res = 1;
    Integer i = 1;
    while((i::Integer) <= (n::Integer))::Boolean{
      (res::Integer) =
        ((res::Integer) * (i::Integer))::Integer;
      (i::Integer)++;
    }
    return(res::Integer);
  }
}
```

The set of remaining constraints which consist only of type variables is empty. If this set would not be empty, then type parameters (generics) of the class or of its method would be generated. We consider this in Section 3.

3 Generalized type variables

In a similar way as in type inference of functional programming languages, free type variables which are not instanced by other types after type inference are generalized to generics. In comparison to functional programming languages, in Java subtyping leads to a more powerful generalization mechanism.
Keeping in mind the result of the type unification (Sec. 2.1)

$$\{\,(\{\,\overline{T_1 \lessdot T_1'}\,\}, \sigma_1), \quad \ldots, \quad (\{\,\overline{T_n \lessdot T_n'}\,\}, \sigma_n)\,\}.$$

given as a set of pairs of

- remaining constraints consisting only of pairs of type variables $\{\,\overline{T_i \lessdot T_i'}\,\}$ and a
- most general unifier σ_i.

```
class TPHsAndGenerics {

    id = x -> x;

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

    m(a, b){
        return b; }

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

```
class TPHsAndGenerics {
    Fun1$$<UD, ETX>
        id = (DZP x) -> x;

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

    AI m(AM a, AN b){
        return b; }

    AA m2(AB a, AD b){
        AE c = m(a,b);
        return a; }
}
```

Fig. 1. Class `TPHsAndGenerics` before and after tree traversing

In the previous section we considered the unifiers. In this section we shall consider the remaining constraints. In the existing type inference algorithm of functional programming languages without subtyping (e.g. Haskell or SML) the remaining type variables are generalized such that any type can be instantiated if the function is used.

Following up this idea, the remaining type variables become bound type parameters of the class and its methods, respectively, where the left-hand side of a constraint is a type parameter and the right-hand side is its bound. Additionally, due to the Java restrictions of type parameters, some type parameters have to be collected to one new type parameter.

This section is structured as follows: After a motivating example, we present an apportioning of the type variables to the class and its methods. We then reduce the respective set of type variables such that the Java restrictions of type variables are fulfilled.

Let us start with a motivating example.

Example 3. On the left side in Fig. 1 a Java-TX program is given. The identity function is mapped to the field `id`. In the method `id2` the identity function is called. In the method `m2` the method `m` is called.

The application of the tree traversing step is presented on the right side where we leave out inner type variables. The result of the type unification is $\{ (cs, [\,]) \}$ where the remaining set of constraints of the type is:

$$cs = \{ \mathtt{AD} \lessdot \mathtt{AN}, \mathtt{AN} \lessdot \mathtt{AI},\ \mathtt{V} \lessdot \mathtt{UD},\ \mathtt{AI} \lessdot \mathtt{AE},\ \mathtt{AB} \lessdot \mathtt{AM}, \mathtt{DZP} \lessdot \mathtt{ETX},\ \mathtt{UD} \lessdot \mathtt{DZP} \}.$$

3.1 Family of generated generics

We divide up the set of remaining constraints cs by transferring it to bound type variables of the class and each method of the class, respectively. Therefore we build a family of generated generics FGG where the index set is given as the class name and its method names.

Definition 1 (Family of generated generics). *The* family of generated generics *is defined as*

$$FGG = (FGG_{in})_{in \in CLM},$$

where

$$CLM = \{\, cl \,\} \cup \{\, m \mid m \text{ is method in } cl \,\}$$

is the index set of the class name and its methods' names.
Let cs be a set of remaining constraints as result of the type unification. cs is transferred to the family of generated generics FGG where, the set of generated generics of the class FGG_{cl} *are given as:*

- *all type variables of the fields with its bounds, including the initializers*
- *the closure of all bounds of type variables of the fields with its bounds, and*
- *all unbound type variables of the fields and all unbound bounds with* `Object` *as bound.*

The set of generated generics FGG_m *of its methods m: are given, respectively, as:*

- *the type variables of the method m with its bounds, where the bounds are also type variables of the method,*
- *all type variables of the method m with its bounds, where the bounds are type variables of fields and*
- *all unbound type variables of the method* `m` *and all type variables of* `m` *which bounds are not type variables of* `m` *with* `Object` *as bound.*

We present the family of generated generics for the class `TPHsAndGenerics` from Example 3

Example 4. The set of remaining constraints

$$cs = \{\, \texttt{AB} \lessdot \texttt{AA}, \texttt{AD} \lessdot \texttt{AN}, \texttt{AN} \lessdot \texttt{AI}, \texttt{V} \lessdot \texttt{UD}, \texttt{AI} \lessdot \texttt{AE}, \texttt{AB} \lessdot \texttt{AM}, \texttt{DZP} \lessdot \texttt{ETX}, \texttt{UD} \lessdot \texttt{DZP} \,\}$$

of the class `TPHsAndGenerics` results in the family of generated generics.
The set of generated generics $FGG_{\texttt{TPHsAndGenerics}}$ **of the class:**

- Type variables of the fields with its bounds:
$$\{\, \texttt{UD} \lessdot \texttt{DZP} \,\}$$
- Closure of all bounds of type variables of the fields with its bounds:
$$\{\, \texttt{DZP} \lessdot \texttt{ETX} \,\}$$

```
class TPHsAndGenerics
        <UD extends DZP, DZP extends ETX, ETX> {

    Fun1$$<UD, ETX> id = x -> x;

    <V extends UD> ETX id2(V x) {
        return id.apply(x);}

    <AM, AN extends AI, AI> AI m(AM a, AN b){
        return b;}

    <AA, AB extends AA, AD, AE> AA m2(AB a, AD b){
        AE c = m(a,b);
        return a;}
}
```

Fig. 2. Generated generics of the class `TPHsAndGenerics`

– All unbound type variables of the fields and all unbound bounds with `Object` as bound:

$$\{\, \texttt{ETX} \lessdot \texttt{Object} \,\}$$

The set of generated generics $FGG_{\texttt{id2}}$:

– All pairs where the bounds are type variables of fields:

$$\{\, \texttt{V} \lessdot \texttt{UD} \,\}$$

The set of generated generics $FGG_{\texttt{m}}$:

– The type variables of the method m with its bounds, where the bounds are also type variables of the method:

$$\{\, \texttt{AN} \lessdot \texttt{AI} \,\}$$

– All unbound type variables of the method m with `Object` as bound:

$$\{\, \texttt{AM} \lessdot \texttt{Object}, \texttt{AI} \lessdot \texttt{Object} \,\}$$

The set of generated generics $FGG_{\texttt{m2}}$:

– The type variables of the method m with its bounds, where the bounds are also type variables of the method:

$$\{\, \texttt{AB} \lessdot \texttt{AA} \,\}$$

– All unbound type variables of the method `m` and all type variables of `m` which bounds are not type variables of `m` with `Object` as bound:

$$\{\, \texttt{AD} \lessdot \texttt{Object}, \texttt{AE} \lessdot \texttt{Object} \,\}$$

The mapping of the family to the class and its methods in the Java-TX program is presented in Fig. 2, where the bounds `Object` are left out.

This is not yet a correct Java-program. In the method m2 the type AD of the second parameter b of the method-call of m must be a subtype of the type AE of the local variable c, as in the method m the argument type AN is subtype of the return type AI. We address this problem by extending the family of generated generics to the completed family of generated generics.

Definition 2 (Completed family of generated generics). *Let cs be the remaining constraints after unification and FGG be the family of generated generics. The* completed family of generated generics *CFGG is defined as*

- $CFGG_{cl} = FGG_{cl}$
- $CFGG_m$ *coresponds to* FGG_m *where* $T_i \lessdot \texttt{Object}$ *is substituted by* $T \lessdot R$ *for each method call*

$$\texttt{rec} :: \texttt{cl.m}'(\overline{\texttt{e} :: ty}) :: rty$$

in the method m with the signature $\overline{ty}' \to rty'$ *where* $T \in \mathsf{TVar}(\, ty_i \,)$, $T' \in \mathsf{TVar}(\, ty_i' \,)$, $R' \in \mathsf{TVar}(\, rty' \,)$, $R \in \mathsf{TV}(\, rty \,)$, $T \lessdot T' \leq^* R' \lessdot R$ *is in the transitive closure of cs, and* $T' \leq^* R'$ *is in the transitive closure of* $CFGG_{m'}$,

Example 5. In the completed family of generated generics of the class TPHsAndGenerics in the set of generated generics $FGG_{\texttt{m2}}$ the bound of AD is changed from Object to AE:

$$\{\, \texttt{AD} \lessdot \texttt{AE} \,\}$$

as the method m is called in m2:

$$\texttt{AE c = m}(\, \texttt{a} :: AB, \texttt{b} :: AD \,),$$

$\texttt{AD} \leq^* \texttt{AN} \leq^* \texttt{AI} \leq^* \texttt{AE}$ is in the transitive closure of cs, and $\texttt{AN} \leq^* \texttt{AI}$ is in the transitive closure of $FGG_{\texttt{m}}$.

The algorithm to build the completed family of generated generics (Def. 1) corresponds to Def. 1.

Algorithm 1 (Completed family of generated generics)

Input: *Family of generated generics FGG*
Output: *Completed family of generated generics CFGG*

Foreach element FGG_m *of the family of generated generics determine the completed family of generated generics* $CFGG_m$ *with the following algorithm:*

1. *If* $CFGG_m$ *is not initialized*
   - ***Initialize*** $CFGG_m = FGG_m$
   - *Mark any recursive method call of m in the method m*
2. *For each not marked method call*

$$\texttt{rec} :: \texttt{cl.m}'(\overline{\texttt{e} :: ty}) :: rty$$

in the method m with the signature $\overline{ty}' \to rty'$ *and foreach* $T \in \mathsf{TVar}(\, ty_i \,)$, $T' \in \mathsf{TVar}(\, ty_i' \,)$ *with* $T \lessdot T'$ *element of the transitive closure of cs* {

– *Mark any method call* m' *in* m
– $CFGG_{m'}$ *is the result of the recursive call of the algorithm for* $FGG_{m'}$
– *If* $R' \in \mathsf{TVar}(\, rty' \,)$, $R \in \mathsf{TV}(\, rty \,)$, $T \lessdot T' \leq^* R' \lessdot R$ *is in the transitive closure of* cs, *and* $T' \leq^* R'$ *is in the transitive closure of* $CFGG_{m'}$
add $(T \lessdot R')$ ***to*** $CFGG_m$**.**

}

It is obvious that the algorithm terminates after all method calls are marked.

Lemma 1. *The algorithm 1 is correct.*

Proof. A relationship between an input type variable and an output type variable cannot be derived from a (mutual) recursive call of the method. Therefore on the one hand direct recursive calls are leaved out, induced by the marking during initialization.
One other hand for indirect recursive calls the algorithm to determine $CFGG_m$ is called twice if the methods m and m' are mutual recursive. In the second determination of $CFGG_m$ the first method call of m' is leaved out as it is marked.

We give an example where two methods are mutual recursive.

Example 6. Let the class `Mutual` with the inferred types be given.

```
class Mutual {
    Pair<BB,DD> m1(B x, C y) {
        D y2 = m2(x, y).snd();
        return new Pair<>(id(x),y2);
    }

    Pair<HH,GG> m2(F x, G y) {
        H x2 = m1(x, y).fst();
        return new Pair<>(x2, id(y));
    }

    I id(J x) {
        return x;
    }

}
```

Furthermore, the set of remaining constraints is given as
$cs = \{\, \mathtt{B} \lessdot \mathtt{J}, \mathtt{J} \lessdot \mathtt{I}, \mathtt{I} \lessdot \mathtt{BB}, \mathtt{B} \lessdot \mathtt{F}, \mathtt{C} \lessdot \mathtt{G}, \mathtt{G} \lessdot \mathtt{J}, \mathtt{I} \lessdot \mathtt{GG}, \mathtt{F} \lessdot \mathtt{B}, \mathtt{G} \lessdot \mathtt{C}, \mathtt{GG} \lessdot \mathtt{D}, \mathtt{BB} \lessdot \mathtt{H} \,\}$

The family of generated generics is given as:
$FGG_{\mathtt{m1}} = \{\, \mathtt{B} \lessdot \mathtt{Object}, \mathtt{C} \lessdot \mathtt{Object}, \mathtt{D} \lessdot \mathtt{DD}, \mathtt{DD} \lessdot \mathtt{Object}, \mathtt{BB} \lessdot \mathtt{Object} \,\}$
$FGG_{\mathtt{m2}} = \{\, \mathtt{F} \lessdot \mathtt{Object}, \mathtt{G} \lessdot \mathtt{Object}, \mathtt{H} \lessdot \mathtt{HH}, \mathtt{HH} \lessdot \mathtt{Object}, \mathtt{GG} \lessdot \mathtt{Object} \,\}$
$FGG_{\mathtt{id}} = \{\, \mathtt{J} \lessdot \mathtt{I}, \mathtt{I} \lessdot \mathtt{Object} \,\}$

Determination of the completed family of generated generics:

1. $CFGG_{m1} = FGG_{m1}$
2. Method call $m2$ with $\overline{ty} = (\mathtt{B}, \mathtt{C})$ and $\overline{ty}' = (\mathtt{F}, \mathtt{G})$
   Mark the method call $m2$
   1. $CFGG_{m2} = FGG_{m2}$
   2. Method call $m1$ with $\overline{ty} = (\mathtt{F}, \mathtt{G})$ and $\overline{ty}' = (\mathtt{B}, \mathtt{C})$
      Mark the method call $m1$
      1. $CFGG_{m1}$ is initialized
      2. Method call $m2$ is marked
         Method call id with $\overline{ty} = \mathtt{B}$ and $\overline{ty}' = \mathtt{J}$
         Mark the method call id
         1. $CFGG_{id} = FGG_{id}$
         2. no method call

         As $\mathtt{B} \lessdot \mathtt{J} \lessdot \mathtt{I} \lessdot \mathtt{BB} \in cs$ and $\mathtt{J} \lessdot \mathtt{I} \in CFFG_{id}$
         add $(\mathtt{B} \lessdot \mathtt{BB})$ **to** $CFGG_{m1}$

      As $\mathtt{F} \lessdot \mathtt{B} \lessdot \mathtt{BB} \lessdot \mathtt{H} \in cs$ and $\mathtt{B} \lessdot \mathtt{BB} \in CFFG_{m1}$
      add $(\mathtt{F} \lessdot \mathtt{H})$ **to** $CFGG_{m2}$

      Method call id with $\overline{ty} = \mathtt{G}$ and $\overline{ty}' = \mathtt{J}$
      Mark the method call id
      1. $CFGG_{id}$ is initialized
      2. no method call

      As $\mathtt{G} \lessdot \mathtt{J} \lessdot \mathtt{I} \lessdot \mathtt{G}' \in cs$ and $\mathtt{J} \lessdot \mathtt{I} \in CFFG_{id}$
      add $(\mathtt{G} \lessdot \mathtt{GG})$ **to** $CFGG_{m2}$

   As $\mathtt{C} \lessdot \mathtt{G} \lessdot \mathtt{HH} \lessdot \mathtt{D} \in cs$ and $\mathtt{G} \lessdot \mathtt{HH} \in CFFG_{m2}$
   add $(\mathtt{C} \lessdot \mathtt{D})$ **to** $CFGG_{m1}$

   Method call id is marked

This leads to the result:

```
class Mutual  {
  <B extends BB, BB, C extends D, D extends DD, DD>
  Pair<BB,DD> m1(B x, C y) {
      D y2 = m2(x, y).snd();
      return new Pair<>(id(x),y2);
    }

  <F extends H, H extends HH, HH, G extends GG, GG>
  Pair<HH,GG> m2(F x, G y) {
      H x2 = m1(x, y).fst();
      return new Pair<>(x2, id(y));
    }

  <J extends I, I> I id(J x) {
      return x;
  }
}
```

In the following section we transform the completed family of generated generics to Java generics of the class and its methods, respectively.

3.2 Java-conforming binary relation of type parameters

We have to reduce the completed family of generated generics:

- The subtyping hierachy is an ordering. Therefore cyles have to eliminated.
- Java allows no multiple inheritance. Therefore infima have to be eliminated.
- As `Object` is a supertype of each class-type sometimes suprema can leaved out. In these cases during the call `Object` as supremum is inferred.
- The argument types of methods are contravariant und return types are covariant. Therefore some type arguments and return types can be equalized without loosing the principalty of the type.

In the following two sections we transform the family of generated generics to Java class and method generics, respectively. First we eleminate cyles. After that we eliminate infima and give some simplifications.
The set of remaining constraints cs as well as any element of the completed family of generated generics $CFGG_m$ are arbitrary binary relations.
There are two conditions in Java that all members of the completed family of generated generics have to fulfill:

- The reflexive and transitive closure of its have be a partial ordering as the subtyping relation in any Java program is a partial ordering.
- Two different elements have no common infimum as multiple inheritance is prohibited in Java.

Furthermore there are some inferred type variables that can be leaved out without hampering the prinipality of the types.
The general approach is to equalize type variables by a surjective map h that preserves the subtype relation:
For $T \leq^* T'$

$$h(T) \leq^* h(T')$$

holds true.

Removing cyles Considering the following lemma:

Lemma 2. *The reflexive and transitive closure of any binary relation is a partial ordering if it contains no cycle.*

Proof. A partial ordering is binary relation with the properties *reflexivity*, *transitivity*, and *antisymmetry*. The first two properties are given as we consider a reflexive and transitive closure. Therefore a reflexive and transitive closure is no partial ordering only if the property antisymmetry is not given. Antisymmetry is given if and only if there are no cycles.

This means that we have to eliminate cycles. We will do this by a surjective mapping of connected type variables to a new type variable.
First, we shall consider an example.

Example 7 (Cycle). Let the class `Cycle` be given:

```
class Cycle {
    m(x, y) {
        y = x;
        x = y;
    }
}
```

For the inferred method parameter `m(L x, M y)` we get

$$CFGG_{\mathtt{m}} = \{\, (L \lessdot M), (M \lessdot L) \,\}$$

But

```
<L extends M,M extends L> void m(L x, M y) {...}
```

is not a correct declaration.

We shall now present an algorithm which eliminate cycles.

Algorithm 2 (Remove cycles)

Input: *A member of the completed family of generated generics C.*
Output: *An adapted member of the family of generated generics C' and a surjective mapping h that describes the adaption of C.*
Postcondition: *For $T \leq^* T' \in C$ and $T' \leq^* T \in C$ holds true $h(T) = h(T')$ and for $T \leq^* T' \in C$ and $T' \leq^* T \notin C$ holds true $T \leq^* T' \in C'$*

For any $(T \lessdot K \lessdot G \lessdot ... \lessdot T)$ in C:
- *Substitute all type variables of the cycle with a new type variable X in C.*
- *Remove all constraints with elements of the cycle from C.*
- *In h all removed type variables of the cycle are mapped to X.*
- *For any element $U \lessdot Cy$ and $Cy \lessdot U'$ with an element of the cycle $Cy \in \{\, T, K, G, \ldots, \}$: the constraints $U \leq^* X$ and $X \leq^* U'$ are added.*

In the following examples, we apply the algorithm to the classes `Cycle` (Example 7).

Example 8. Applying the algorithm to the class `Cycle` we get the surjective mapping h with

$h(L) = X$
$h(M) = X$

and the adapted class:

```
class Cycle {
    <X> void  m(X x, X y) {
        y = x;
        x = y;
    }
}
```

Eliminate infima und further simplifications The following example presents the problem.

Example 9 (Infimum). Let the class `Infimum` be given:

```
class Infimum {
    m( a::A , b::B , c::C ) {
        b = a;
        c = a;
    }
}
```

For the inferred method parameter `m(L x, M y, N x)` we get

$$CFGG_{\mathtt{m}} = \{\, (A \lessdot B), (A \lessdot C), (B \lessdot \mathtt{Object}), (C \lessdot \mathtt{Object}) \,\}$$

But

```
<A extends B, A extends C, B, C> void  m(A x, B y, C z) {...}
```

is not a correct declaration as multiple inheritance is prohibited.

Before we solve this problem we have to consider some other properties. For the elimination of infima we have to differ two cases: Type variables of types of argument and return types on the one hand and type variables of types of inner nodes on the other hand.
We will start with the consideration of type variables of types of argument and return types.

Definition 3 (Type variables in covariant and contravariant position). *A type variable of an argument of a function/method is in* contravariant position. *A type variable of a return type of a function/method is in* covariant position.

In the following we write for a type variable in contravariant position $R^{(-)}$ and for a type variable in covariant position $T^{(+)}$.
From the property that for function types

$$(ty' \to rty) \leq^* (ty \to rty')$$

holds true

$$ty \leq^* ty' \text{ and } rty \leq^* rty'.$$

follows for elements of the members of the completed family of generated generics:

A type variable in contravariant position is a subtype of a type variable in covariant position ($T^{(+)} \lessdot T'^{(-)}$). The subtype $T^{(+)}$ has to be maximal and the supertype $T'^{(-)}$ has to be minimal in a principal type. Therefore these type variables can be equalized.

Two type variables in contravariant positions ($T^{(-)} \lessdot T'^{(-)}$) have to be maximal in a principal type. Therefore, $T^{(-)}$ and $T'^{(-)}$ can be equalized.

Two type variables in covariant positions ($T^{(+)} \lessdot T'^{(+)}$) have to be minimal in a principal type. Therefore, $T^{(+)}$ and $T'^{(+)}$ can be equalized.

A type variable in covariant position is a subtype of a type variable in contravariant position ($T^{(-)} \lessdot T'^{(+)}$). In the principal type $T'^{(+)}$ has to be maximal and $T^{(-)}$ has to be minimal. Therefore these type variables cannot be equalized.

In the following we give some examples which represents the different constellation which have to be considered.

Example 10. Let the class `Id` be given.

```
class Id {
     R id(A x) { return x:A; }
}
```

The completed family of generated generics are given as

$$CFGG_{id} = \{\, A^{(-)} \lessdot R^{(+)} \,\}.$$

As in in this example the case *type variable in contravariant position is a subtype of a type variable in covariant position* is given the type variables can be equalized and the simplified program is given as

```
class Id {
     <A> A id(A x) { return x; }
}
```

The next example represents the case that a covariant type variable is a subtype of a contravariant type variable.

Example 11. Let the class `Recursion` be given.

```
class Recursion {
    R m( a::A , b::B ) {
      if (cond) b = m(a, b);
      else return a;
  }
}
```

$$CFGG_m = \{\, A^{(-)} \lessdot R^{(+)}, R^{(+)} \lessdot B^{(-)}, B^{(-)} \lessdot \texttt{Object} \,\}$$

The constraint is $R^{(+)} \lessdot B^{(-)}$ remains as if in the case that a *type variable in covariant position is a subtype of a type variable in contravariant position* the type variables cannot be equalized. The type variables $A^{(-)}$ and $R^{(+)}$ are equalized.

The result program is given as

```
class Recursion {
   <R extends B, B> R m(R a, B b) {
      if (cond) b = m(a, b);
      return a;
   }
 }
```

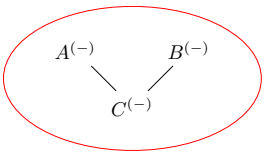
In the following we will see that there are more complex constellations with inifima and inner type variables which we have to solve, too.
First, we consider the elimination of inifima.

Eliminatation of inifima As said above it is necessary to eliminate all inifima, as infima are not allowed in Java subtyping hierachies. We consider examples that represents the different constellations.

I. All type variables are in contravariant position

Example 9 is an example for this case.

Example 12. As in Example 9 the three type variables $A^{(-)}, B^{(-)}, C^{(-)}$ are contravariant and *type variables in contravariant positions* can be equalized the result program is given as




```
class Infimum {
    <A> void  m(A a, A b, A c) {
         b = a;
         c = a;
     }
}
```

II. All type variables are in covariant position The following example shows a covariant infimum of covariant type variables.

Example 13. Let the classes `Triple` and `InfCovariant` be given.

```
class Triple<U, T, S> {
    U a;
    T b;
    S c;

    U fst() { return a; }
    T snd() { return b; }
    S trd() { return c; }
}

class InfCovariant {
```

```
    R m() {
      return new Triple<>(m().trd(), m().trd(), m().trd());
    }
}
```

The *type unification* function determines $R =$ `Triple<`$R1$`,` $R2$`,` $R3$`>` The completed family of generated generics is given in Fig. 3. The result is given as:

$$CFGG_m = \{\, R3^{(+)} \lessdot R1^{(+)},\; R3^{(+)} \lessdot R2^{(+)},\; R1^{(-)} \lessdot \texttt{Object},\; R2^{(-)} \lessdot \texttt{Object} \,\}$$

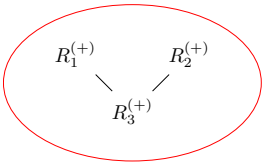


Fig. 3. All type variables are in covariant position

```
class InfCovariant {
    <R3> Triple<R3, R3, R3> m() {
        return new Triple<>m().trd(),m().trd(),m().trd());
    }
}
```

III. Covariant infimum of contravariant type variables The next example represents the case that a covariant type variable is a subtype of two contravariant type variable. Therefore we extend the Example 11.

Example 14. Let the class `RecursionInf` be given.

```
class RecursionInf {
    R m( a::A , b::B , c::C ) {
      if (cond) {
          b = m(a, b, c);
          c = m(a, b, c);
      else return a;
    }
}
```

The completed family of generated generics is given in Fig. 4. The constraints $R^{(+)} \lessdot B^{(-)}$ and $R^{(+)} \lessdot C^{(-)}$ should be remained as if in the case that a *type variable in covariant position is a subtype of a type variable in contravariant position* the type variables cannot be equalized. As above, the type variables $A^{(-)}$ and $R^{(+)}$ are equalized. Additionally, the type variables $B^{(-)}$ and $C^{(-)}$ have to be equalized to elinimate the infimum. This make the program indeed less principal but it becomes type correct. There is no more principal type correct solution.

The result program is given as

$$CFGG_m = \{\, A^{(-)} \lessdot R^{(+)},\; R^{(+)} \lessdot B^{(-)},\; R^{(+)} \lessdot C^{(-)},\; B^{(-)} \lessdot \texttt{Object},\; C^{(-)} \lessdot \texttt{Object} \,\}$$

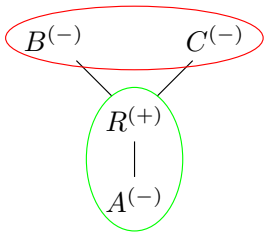


Fig. 4. Covariant infimum of contravariant type variables

```
class RecursionInf {
   <R extends B, B> R m(R a, B b, B c) {
     if (cond) {
       b = m(a, b);
       c = m(a, b);
      }
      return a;
   }
 }
```

IV. Inner type variables as Infima First we consider an example, where an inner type variable is an infimum of two contravariant type variables.

Example 15. Let the class `InnerInf` be given:

```
class InnerInf {
    void m( a::A , b::B ) {
        var i::I  = null;
        a = i;
        b = i;
    }
}
```

The completed family of generated generics is given in Fig. 5. All type variables

$$CFGG_m = \{\, I \lessdot A^{(-)},\; I \lessdot B^{(-)},\; A^{(-)} \lessdot \texttt{Object},\; B^{(-)} \lessdot \texttt{Object} \,\}$$

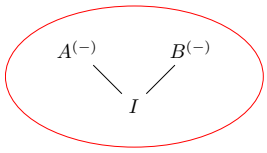


Fig. 5. Inner type variables as Infimum of two contravariant type variables

have to be equalized to elinimate the infimum. The result is given as:

```
class InnerInf {
```

```
    <A> void m(A a, A b) {
        A i = null;
        a = i;
        b = i;
    }
}
```

The next two examples shows the situation that the infimum is an inner type variable of type variables with different variance.

Example 16. Let us consider the following class:

```
class InfReturn {
    R  m( a::A  ) {
      var ret::I  = null;
      a = ret;
      return ret;
    }
 }
```

The completed family of generated generics is given in Fig. 6.

$$CFGG_m = \{\, I \lessdot A^{(-)},\ I \lessdot R^{(+)},\ A^{(-)} \lessdot \texttt{Object},\ R^{(+)} \lessdot \texttt{Object} \,\}$$

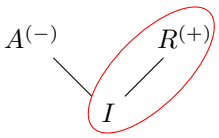


Fig. 6. Inner type variables as Infimum of type variables with different variance (I)

This family contains an infimum I. Therefore this have to be eliminated. We have to determine if the inner type variable I should be co- or contravariant. If we consider I as contravariant all type variables could be equalized as $I^{(-)} \lessdot R^{(+)}$, and $I^{(-)} \lessdot A^{(-)}$. This approach is less principal than the approach to consider I as covariant. This means $I^{(+)} \lessdot R^{(+)}$, and $I^{(+)} \lessdot A^{(-)}$. In this case only I and R are equalized. The result program is given as:

```
class InfReturn {
    <R extends A, A> R m(A a) {
        I ret = null;
        a = ret;
        return ret;
    }
}
```

In the following we extend the class such that two contravariant type variables are supertypes of the infimum.

Example 17. Let the extended class `InfReturn` be given.

```
class InfReturnII {
    R  m( a::A , b::B ) {
      var ret::I  = null;
      a = ret;
      b = ret;
      return ret;
    }
 }
```

The completed family of generated generics is given in Fig. 7.

$$
\begin{aligned}
CFGG_m = \{\, & I \lessdot A^{(-)}, \\
& I \lessdot B^{(-)}, \\
& I \lessdot R^{(+)}, \\
& A^{(-)} \lessdot \texttt{Object}, \\
& B^{(-)} \lessdot \texttt{Object}, \\
& R^{(+)} \lessdot \texttt{Object} \,\}
\end{aligned}
$$

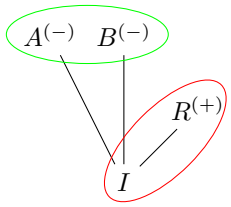


Fig. 7. Inner type variables as Infimum of type variables with different variance (II)

The approach to declare the infimum I as covariant leads to $I^{(+)} \lessdot R^{(+)}$, $I^{(+)} \lessdot A^{(-)}$, and $I^{(+)} \lessdot B^{(-)}$. In this case I and R are equalized, too. But in this example there is an new infimum R with $R^{(+)} \lessdot A^{(-)}$ and $R^{(+)} \lessdot B^{(-)}$, that has be eliminated, too.
As in Example 14 in both constraints the *type variables in covariant position are subtypes of a type variable in contravariant position.* Therefore, the type variables cannot be equalized.
To elinimate the infimum R the type variables A and B have to be equalized. As in Example 14 this make the program indeed less principal but it becomes type correct. There is no more principal type correct solution.
The result program is given as:

```
class InfReturnII {
   <R extends A, A> R m(A a, A b) {
        R ret= null;
        a = ret;
        b = ret;
        return ret;
    }
}
```

We close the section with a complex example with two infima and two suprema. We will see that the infima have to be eliminated and the suprema can be eliminated to siplify the example.

Example 18. Let the following class `Complex` be given:

```
class Complex {
    m( b::B ) {
        var c::C  = b;
        var d::D  = c;
        var e::E  = null;
        d = e;
        var r1::R1  = e;
        var f::F  = e;
        var g::G  = null;
        f = g;
        var r2::R2  = g;
        return new Pair<>(r1,r2);
    }
}
```

The completed family of generated generics is given in Fig. 8.

$$
\begin{aligned}
CFGG_{\mathtt{m}} = \{\ & B^{(-)} \lessdot C,\\
& C \lessdot D,\\
& E \lessdot D,\\
& D \lessdot \mathtt{Object},\\
& E \lessdot R1^{(+)},\\
& R1^{(+)} \lessdot \mathtt{Object},\\
& E \lessdot F,\\
& F \lessdot \mathtt{Object},\\
& G \lessdot F,\\
& G \lessdot R2^{(+)},\\
& R2^{(+)} \lessdot \mathtt{Object}\ \}
\end{aligned}
$$

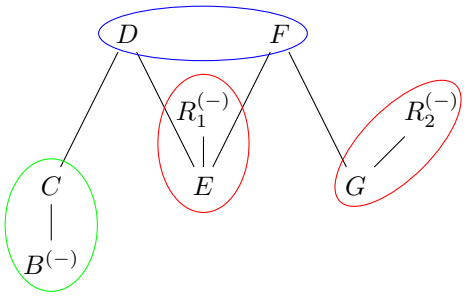


Fig. 8. Complex example

The program contains two infima E and G which have to be eliminated. Similar as in Example 16 and in Example 17 the type variables E and R_1 and G and R_2 are equalized, respectively. This eliminates the infimum G. Furthermore, the type variables B and C can be equalized. As B is in contravariant position it should be as maximal as possible. If additionally D would be equalized with B and C the program becomes less principal as E is no subtype of B but E would be a subtype of the equalized type variable. Finally, the infimum R_1 respectively E of D and F have to be eliminated. As neither R_1 nore R_2 should become a supertype of B the type variables D and F have to equalized. The result is given in Figure 9.
If we consider the result more in detail we will see that the supremum is an inner type variable. Therefore any upper bound can be instantiated during a method call of `m`. As `Object` is an upper bound of any set of `Java` types we can instantiate `Object` without loosing principality.
The simplified result is given as:

```
class Complex {

    <B extends F,
     R1 extends F,
     R2 extends F, F>
    Pair<R1, R2> m(B b) {
        B c = b;
        F d = c;
        R1 e = null;
        d = e;
        R1 r1 = e;
        F f = e;
        R2 g = null;
        f = g;
        R2 r2 = g;
        return new Pair<>(r1,r2);
    }
}
```

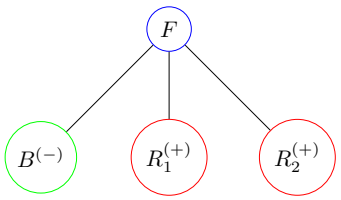


Fig. 9. Complex example

```
class Complex {
    <B, R1, R2> Pair<R1, R2> m(B x) {
        B c = b;
        Object d = c;
        R1 e = null;
        d = e;
        R1 r1 = e;
        Object f = e;
        R2 g = null;
        f = g;
        R2 r2 = g;
        return new Pair<>(b,e);
    }
}
```

4 Summary and Outlook

In this paper we presented first a short summary of the two existing functions of the Java-TX global type inference algorithm *tree traversing* and *type unification*. The main part is the new function *generated generics*. This functions takes the remaining type variables of the type unification and transfers them to type parameters (generics) of the classes and its methods, respectively, in a way such they become principal types.
This is done in three steps. First, the type variables are mapped to a class or a method, respectively. After that some additional bounds of the type parameters are generated induced by the method calls. The next step is to transform the

bounded type parameters to Java-conforming type parameters. Therefore first the cycles are removed. Second, the infima are eliminated. The elimination of the infima is a complex process. The main challenge is one the one hand indeed to eliminate the infima which means that the type can become less principal but on the other hand the challenge is to get the most principal type that is a correct Java type. Therefore we have to consider different constellations which have to be handled different. In this paper we considered many constellations and give corresponding examples, respectively.
In future work these constellations have to be summerized to an algorithm *eliminate infima* which become the last part of the function *generate generics*.

References

1. Goetz, B.: Jep 286: Local-variable type inference (2016), `http://openjdk.java.net/jeps/286`, updated: 2018/10/12 01:28
2. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification. The Java series, Addison-Wesley, 3rd edn. (2005)
3. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java® Language Specification. The Java series, Addison-Wesley, Java SE 8 edn. (2014)
4. Plümicke, M.: Java type unification with wildcards. 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers **5437**, 223–240 (2009). https://doi.org/10.1007/978-3-642-00675-3_15
5. Plümicke, M.: More type inference in Java 8. Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers **8974**, 248–256 (2015). https://doi.org/10.1007/978-3-662-46823-4_20
6. Plümicke, M., Stadelmeier, A.: Introducing Scala-like function types into Java-TX. In: Proceedings of the 14th International Conference on Managed Languages and Runtimes. pp. 23–34. ManLang 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3132190.3132203, `http://doi.acm.org/10.1145/3132190.3132203`
7. Steurer, F., Plümicke, M.: Erweiterung und Neuimplementierung der Java Typunifikation. In: Knoop, J., Steffen, M., y Widemann, B.T. (eds.) Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts. pp. 134–149. No. 482 in Research Report, Faculty of Mathematics and Natural Sciences, UNIVERSITY OF OSLO (2018), iSBN 978-82-7368-447-9, (in german)

A grammar centric approach to generate drag & drop subsets of programming languages*

Marcus Riemer[1,2] and Frank Huch[1]

[1] Christian-Albrechts-Universität, Kiel, Germany
[2] AKRA GmbH, Hamburg, Germany, marcus.riemer@akra.de

Abstract. We present a way to use two of the most typical tools of compiler construction, namely grammars and (syntax) trees, as the foundation to build a generator that can create drag & drop editors for programming or markup languages. Instead of text files, the generated editors operate on syntax trees and use special grammars to ensure the resulting tree is well formed. The editors can be generated for two different backends: Either our custom editor called *BlattWerkzeug* or *Blockly*, which is the de-facto standard for drag & drop editors and e.g. used by *Scratch*. If the grammar additionally defines the syntax of the language through terminal symbols, the syntax tree can immediately be represented as a properly formatted text document as well. The grammar we developed is conceptually based on XML Schema and is used to validate trees where each node corresponds to a block that is draggable in the user interface. This visual use case brings the important requirement of never confronting the user with a block that has no representation. To aid in this requirement, the grammar can define artificial nodes which do not appear in the tree but only aid in validation. We currently have prototypes for drag & drop subsets of SQL, JavaScript & HTML and are exploring design patterns to build intuitive editors, possibly involving the use of parser-parser combinators, as our current area of research.

* Supported by AKRA Hamburg

Fuzzing through the Prism of Programming Language Theory

Vasil Sarafov, Stefan Brunthaler

Munich Computer System Research Laboratory μCSRL,
Universität der Bundeswehr München,
Munich, Germany
vasil.sarafov@unibw.de, brunthaler@unibw.de
https://ucsrl.de

Abstract. In recent years fuzz testing has emerged as an effective testing strategy for finding bugs. Interestingly, the principles behind fuzzing — feeding a system with randomly generated input and observing the said system for negative behaviour — have shown success for both simple and complex hardware and software systems.
In this work we present an introduction to fuzz testing from the perspective of programming language theory. We reflect on how fuzzing fits the big picture of program analysis and illuminate its connections to formal methods.
Furthermore, we present our ongoing work into a theoretical model that seeks to succinctly explain fuzzing via Markov Decision Processes. We provide concrete examples for fuzzing's fundamental limits.
On the experimental side, we discuss implementation optimizations that seek to increase the fuzzing utilization of existing hardware. We complement the discussion with preliminary evaluation results.

Auffinden von Quellcode-Klonen in Zwischencodedarstellungen von Java Bytecode

André Schäfer[1], Thomas S. Heinze[2], and Wolfram Amme[1]

[1] Friedrich Schiller University Jena, Germany
{andre.schaefer,wolfram.amme}@uni-jena.de
[2] Cooperative University Gera-Eisenach, Gera, Germany
thomas.heinze@dhge.de

Abstract. Kopierter, wiederverwendeter und veränderter Programmcode ist ein häufiges Phänomen in der Softwareentwicklung. Da die daraus resultierenden Code-Klone oft nicht als solche gekennzeichnet sind und ihre korrekte Erkennung für die Softwarequalität und das Re-Engineering von entscheidender Bedeutung ist, wurden zahlreiche Techniken und Werkzeuge zum Auffinden von Code-Klonen entwickelt. Dies gilt auch für die Programmiersprache Java, obwohl die meisten Klon-Detektoren mit Java-Quellcode arbeiten. Nur ein Teil der Forschung befasst sich mit Code-Klonen in Java Bytecode und ein noch kleinerer Teil untersucht die Beziehung zwischen Java-Quellcode und Bytecode-Code-Klonen. In dieser Arbeit erweitern wir den quellcodebasierten Klon-Detektor StoneDetector auf Java Bytecode. Mit BigCloneBench als State-of-the-Art-Benchmark sind wir in der Lage, seine Effektivität bei der Erkennung von Quellcode-Klonen in Java Bytecode zu bewerten und zu analysieren. Wir berichten auch über Unterschiede in der Leistung der Klon-Erkennung für Stack-basierte und Register-basierte Repräsentationen von Java Bytecode. Die Ergebnisse zeigen, dass Quellcode-Klone in Java-Quellcode und Bytecode im Allgemeinen zwar unterschiedlich sein können, dass aber Quellcode-Klone in Java-Bytecode mit dem angepassten System mit hoher Wiedererkennung und Präzision erkannt werden können.

Why Rank-Polymorphism Matters

Sven-Bodo Scholz[1]

Radboud University, Nijmegen, Netherlands
SvenBodo.Scholz@ru.nl

Abstract. Rank-Polymorphism refers to programming language support for specifying functions that operate on arrays of arbitrary dimensionality (rank). This paper argues why this is an important language feature even though it may intuitively appear that in real-world applications data typically lends itself to rather low-dimensional index-spaces. By means of a few examples in the rank-polymorphic array programming language SaC we show how rank-polymorphism does not only allow for very concise generic specifications, but it also is conducive when expressing algorithmic variations for optimising algorithms with respect to parallelism pattern or temporal locality.

1 Introduction

Almost all programming languages support some data structure that is referred to as *array*. It usually is seen as a memory region containing several data items that can be directly accessed through consecutive indices. While these indices are traditionally restricted to whole numbers starting from zero or one, many languages nowadays support various different forms of indices, including not only enumerations but also arbitrary strings as indices, usually referred-to as *associative arrays*.

Furthermore, most languages allow for arbitrary nesting of data structures, including nested arrays. Although such nested arrays can be viewed as a way to represent multi-dimensional arrays, the nesting structure itself usually is not an integral part of the array data type itself. This aspect is what sets array programming languages apart. Array Programming Languages are designed around multi-dimensional arrays as predominant data structure where the index space is more complex, typically an n-fold product of indices which, by itself, can be seen as a one-dimensional array of indices.

Multi-dimensional arrays have a natural match with multi-dimensional vector spaces as they occur in many applications in physics and other natural sciences. Since the dimensions of a vector space can be described through a matrix of bases with corresponding rank, array languages typically refer to the number of dimensions as *rank*. The structure of any multi-dimensional array is integral part of such an array. It can be described by its rank and its *shape*. In its simplest form, the shape can be described by a vector of whole numbers that denote the size of the index space for all dimensions.

As soon as we step away from seeing multi-dimensional arrays as a nesting of one-dimensional ones but look at them as a structure with rank and shape, it becomes rather natural to generalise the programming capabilities around these structures. All that is needed to do so is a capability to define such arrays from a given rank and shape and to access their elements through shape-compliant indices. Once these capabilities are provided, we can write abstractions that are rank-polymorphic, i.e., abstractions that can process multi-dimensional arrays of all possible ranks.

While such level of abstraction is conceptually nice, it usually comes at a price in terms of runtime performance. There are many contributing factors: (1) the generality of rank-polymorphism significantly increases the complexity of type systems that are capable to statically identify out-of-bound accesses; (2) iteration over individual elements or certain subsets of elements can no longer be expressed through a fixed nesting of linear loops; (3) various compiler optimisations due to the generality of the required code either become impossible or at least much more difficult; the list goes on. These observations give rise to the question: *Do we really need this level of generality?*

If we look at typical applications that naturally lend themselves to multi-dimensional arrays, such as applications from image processing, natural sciences, machine learning, etc., the number of dimensions that arise from the applications themselves are typically rather small. While it is nice to be able to specify rank-polymorphic library functions, it may seem that a fixed set of dimensions up to a small upper limit, say 4 or 5, in practice might suffice.

As it turns out, this is the wrong way of looking at the question. Instead of asking *What applications require rank-polymorphism?* we should ask the question *What can rank-polymorphism add to our applications?*

This paper follows up on exactly that question. We look at three different applications each of which exposes a different aspect of the expressive power that arises from rank-polymorphism. We identify how rank-polymorphism can take a crucial role to capture more holistic perspectives on the transformation of data, how it can control parallelism, and how it can be key for advanced optimisations such as blocking.

As a vehicle for our exposition, we use the functional array programming language SaC. However, all our observations and results should carry over to any other array programming language with support for rank-polymorphism such as APL [1] or Accelerate [2].

Section 2 gives a brief characterisation of the implications that support for rank-polymorphism has on the required programming language support, be it as a DSL or a standalone language. Section 3 then introduces the key features of SaC (version 2.0) with a special emphasis on those design aspects of SaC that enable rank-polymorphism. The subsequent three sections demonstrate three different aspects of the power of rank-polymorphism at three different example applications. Section 4 looks at ways on how to program neural networks for deep learning. As it turns out, the connections between different layers quite naturally lead to deeply nested arrays of weights which can be conveniently looked at as

higher-dimensional arrays. Controlling the parallelism of an application through the shape of arrays is investigated in Section 5. At the example of `scan` operations, we show how various implementational variants that expose very different behaviours in terms of parallelisation and synchronisation can be captured by a single rank-polymorphic specification. Section 6 again leverages array shapes in order to express different runtime behaviours. This time, however, it concerns the temporal locality of a shape-recursive matrix multiply, which can be steered entirely by the shape of the argument matrices. Finally, we draw some conclusions in Section 7.

2 Rank-Polymorphism in Programming Languages

The key for supporting rank-polymorphism is the introduction of arrays that have an inherent notion of rank and shape. While the former specifies the number of independent indexing dimensions, the latter describes the exact range of legitimate indices. Furthermore, there needs to be support to enable a definition of functions whose domain allows for arrays of arbitrary rank and shape. These two requirements have several immediate implications on the design of languages that support rank-polymorphism.

2.1 Indexing Arrays

From the premise of dealing with an array of arbitrary rank and shape, element-selection requires a sequence of indices whose length matches the rank of the array to be selected from. If we want to capture such a sequence by a single variable, irrespective of the rank of the array we want to select from, that variable itself requires an array-like structure. Many array languages therefore use rank-one arrays as indices.

2.2 Iterating over Arrays

Similar to selections, a dimension-wise iteration through a static nesting of loops for rank-polymorphic programs is not possible in the context of rank-polymorphism. Instead, we need to either nest loops through explicit recursion within a loop body, or we need to provide some form of index generator that is parameterised by a given shape.

2.3 Defining Operators

Defining rank-polymorphic operators requires at least one built-in construct for generating an array of a given rank and shape. In several array languages, in particular the earlier ones like APL, there exists an entire set of built-in operators that are rank-polymorphic and that can be combined in order to generate more complex ones. Other systems such as Blitz [3] provide these basic operations through a shallowly embedded DSL, or through a set of built-in second-order operators such as in Futhark [4].

2.4 Types and Type Systems

There is a wide range of approaches towards typing languages that support multi-dimensional arrays. While APL and APL-like languages typically do not support any explicit notion of types, more recent languages do. Amongst these, the main differentiating criterion is the level of shape information that is captured by the type system. While it would be desirable to statically guarantee shape compliance for entire programs, in general, this requires the full power of dependent typing. Several approaches towards this have been proposed in the literature such as in Qube [5] or in Remora [6]. The inherent un-decidability in general has led to several less stringent type systems such as the one of SaC [7] or the one in Futhark [4]. While SaC injects dynamic checks if correctness remains unproven at compile time, Futhark restricts the set of legal programs to fixed ranks and it does not support recursion.

2.5 Nesting

Nesting in the context of multi-dimensional arrays takes a special role. We have to distinguish two cases: *homogeneous nesting* and *in-homogeneous nesting*. The former refers to a case where the individual sub-arrays of the outer array all have the same shape, whereas the latter allows them to be different.

The key observation is that there is an intuitive isomorphism between homogeneously nested arrays and a counterpart whose rank is simply the addition of the ranks of the two. For example, a vector of m elements, each of which is a vector of n elements can be viewed as a $m \times n$ *matrix*. Most array languages support array algebras that allow for switching arbitrarily between the homogeneously nested and the non-nested view.

In fact most array languages that aim at high-performance parallel computing do not even support in-homogeneous nesting at all while more traditional array languages such as APL typically do support nesting. The programming language Nial [8] takes the aspect of in-homogeneous nesting one step further; it considers all arrays as (in)-homogeneously nested.

3 Single Assignment C

As indicated by the name of SaC, it has a close similarity to the programming language C. In fact, most language constructs from C can be readily used in SaC and they have the same semantics as in C. The key difference between C and SaC is that SaC does not support any data structures from C other than scalar values. Instead, SaC supports multi-dimensional arrays as the only form of data structure. In SaC, all arrays are, at least conceptually, being passed by value. This allows C's scalar values to be considered arrays as well. Like in most array languages, they constitute arrays of rank 0 which have an empty shape. In the following, I provide a brief overview of the key features needed to understand the examples in this paper. For more details please do refer to [7, 9].

3.1 Arrays in SaC

In SaC, arrays can only have rectangular shapes. This choice allows shapes to be described by a single vector of upper bounds for indices whose length matches the rank of a given array. For example, an array of shape `[10,20]` describes a matrix whose first index can range from `0` to `9`, and whose second index ranges between `0` and `19`.

SaC only supports homogeneous nesting and always *implicitly* considers these higher-dimensional arrays. For example, the expression `[[1,2,3],[4,5,6]]` denotes a rank 2 array of shape `[2,3]`. In contrast, an expression of the form `[[1,2,3],[4,5]]` does not form a legal expression in SaC. Selections with fewer indices than the rank of an array return the corresponding hyper planes as if the array was nested. For our array of shape `[2,3]` from above this means that selecting at the index vector `[0]` results in the first row (`[1,2,3]`), while a selection with the index vector `[0,1]` returns the value `2`.

3.2 Function signatures in SaC

Function signatures in SaC very much look like function signatures in C. The key differences are that (i) SaC supports multiple, comma separated return values, and (ii) all argument types are composed of an element type, followed by a shape specification. As element type, all built-in C types are supported. For the shape specification, SaC supports a notion similar to pattern matching.

In their simplest form, we can specify a restriction to a fixed shape. For example, `int[2,3]` refers to rank 2 arrays of integers that have a shape of `[2,3]`. If we want to denote rank 3 integer arrays of arbitrary shape, we can either use `int[.,.,.]` or replace the dot symbols by variables, in case we want to refer to the sizes further (e.g., `int[m,.,n]`). Repeated use of the same variable requires equivalence in those shape components. For example, `double[n,n]` denotes quadratic matrices of double values.

To support rank-polymorphism, SaC allows for pattern within the shape specification that capture entire sequences of shape components. The most generic example is a specification of the form `int[*]`, which allows arbitrarily shaped arrays of integer values. Similar to the variables that can be used to match individual shape components, we can also use variables to match against entire ranges of shape components. To do so, we use a length specifier followed by the colon symbol followed by a variable that matches length-many shape components. The length component itself can either be a constant or a variable itself. For example, `int[3:shp]` will match integer arrays of rank 3 whose shape will be captured by the vector variable `shp`. Likewise, `int[d:shp]` matches arbitrary integer arrays and captures the rank in the variable `d` and the shape in the vector variable `shp`.

All these pattern can be combined allowing for sophisticated shape dependencies to be expressed. For example,

```
double[m:ishp] sel (int[n] iv, double[n:oshp,m:ishp] a)
```

can be used to describe selections into arbitrary arrays of doubles. Here we see that the length `n` of the index vector `iv` splits the shape of the array `a` into two parts and selects the corresponding hyperplane of rank `m` and shape `ishp`.

3.3 Defining arrays using tensor comprehensions

The central language construct of SaC that extends its C core is the tensor comprehension. It takes the form

`{` *<idx>* `->` *<expr>* [`|` *<range-spec>*] `}`

and it describes an array through a mapping from indices (*<idx>*) to values (*<expr>*). The shape of the result array can either be inferred from the way *<expr>* is defined or it can be specified explicitly through an optional range specification (*<range-spec>*).
In its simplest form, we can define a vector of 20 zeros by

```
{ [i] -> 0 | [i] < [20] }
```

.

Assuming a given array `arr` of shape `shp`, we can write

```
{ iv -> arr[iv] + 1 }
```

to define a new array of shape `shp` whose elements are the increments of the corresponding elements in `arr`. Note here, that this element-wise increment works for arbitrary shapes `shp` including empty shapes in case `arr` is a scalar value. If we specify

```
{ [i,j] -> arr[j,i] }
```

we require `arr` to be at least of rank 2 and we transpose the first two axes of `arr`. Finally, we can also create an array of a modified shape. For example, if we write

```
{ iv -> arr[iv] | iv < shp/2 }
```

we create an array that has the same elements as those in `arr` but whose shape has been halved along all axes. For more details on the semantics of tensor comprehensions in SaC, in particular the shape inference when a range specification is missing please see [10].

4 Generality Through Shapes

In this Section, we look at the expressive power that stems from the interplay between homogeneous nesting and rank-polymorphism. As an example, we look at one of the core functions needed when implementing convolutional neural networks (CNNs), one of the standard approaches to deep learning.

At the heart of CNNs lies a convolutional kernel. It takes some input data and blurs the data by computing weighted sums of neighbouring elements. When applying deep learning to images, these data are typically 2-dimensional, however, in general the data can be n-dimensional. To describe a convolution, we need a small array of weights that defines to what degree the corresponding neighbour elements should contribute to the blurring. In SaC, we can describe convolutions for n-dimensional inputs through the following function:

```
float[n:oshp] Convolve (float[n:ishp] in,
                        float[n:wshp] weights)
{
  oshp = ishp - wshp + 1;
  return { iv -> sum ({ jv -> weights[jv] * in[iv+jv] })
              | iv < oshp };
}
```

Here, we see that the output of such a convolution, while being of the same dimensionality `n`, is a bit smaller than the input as the outermost elements of the input data do not have the required neighbouring elements to apply the weighting. We also can see that for each resulting element at index position `iv`, we have to compute a sum over the products of all elements of the array of weights `weights` with the corresponding neighbours of the input `in`.

While this implementation is straight-forward, in practice, we often face the challenge that the CNN needs to apply the same convolution on an entire batch array of inputs. To add to the challenge, the input data often is interleaved as to allow for more efficient runtime performance. While this in traditional languages may require some non-trivial adjustments in offsets and index computations, in an array language such as SaC this can be achieved by two minor adjustments:

```
float[n:oshp,b:bshp] Convolve (float[n:ishp,b:bshp] in,
                               float[n:wshp] weights)
{
  oshp = ishp - wshp + 1;
  return { iv -> rsum (n, { jv -> weights[jv] * in[iv+jv] })
              | iv < oshp };
}
```

The most important change is the extension of the specified argument and return types. Here we add a pattern `b:bsh` to both of them. Note here, that this allows for arbitrary forms of batch applications, including the non-batched case itself! SaC's isomorphism between homogeneous nesting and higher-dimensional arrays allows the expression `{ jv -> weights[jv] * in[iv+jv] }` to be used unchanged. However, it now denotes an array of shape `wshp ++ bshp` since the selections `in[iv+jv]` result in arrays of shape `bshp` rather than just individual elements. Since the function `sum` that we used previously always reduces to individual elements, we have to adjust that reduction to ensure that we sum up arrays of shape textttbshp. This can be done by using a function `rsum` instead, which expects the number of ranks that it should sum over as a first parameter. Once that is done, the expression `rsum (n, { jv -> weights[jv] * in[iv+jv] })` computes an array of shape `bshp`, leading to the desired overall result.

The next challenge we face is the need to apply convolutions to an entire set of weights, yielding an entire array of convolved inputs as results. Furthermore, we typically want to add some bias to each of these convolutions. This can be expressed, using the same nesting-isomorphism:

```
float[m:mshp,n:oshp,b:bshp] MultiConv (float[n:ishp,b:bshp] in,
                                       float[m:mshp,n:wshp] weights,
                                       float[m:mshp] bias)
{
  return { iv -> Convolve (in, weights[iv]) + bias[iv]
              | iv < mshp };
}
```

We add another shape, this time around the weights. It is of rank `m` and of shape `mshp`. In the body of the function, we have a single tensor comprehension that maps the convolution `Convolve` over this shape, yielding an array of rank `m+n+b` as result.

This most interesting aspect of this little example is that the resulting code, despite only requiring less than 20 lines of code, is very generic. It allows for convolutions on input of arbitrary rank, supports batching without requiring it (iff `b=0`), and it supports multiple convolutions with different weights without requiring that either (iff `m=0`).

For more details on implementng full CNNs in SaC and a comparison of runtime performance with state of the art frameworks see [11].

5 Controlling Parallelism Through Shapes

In this Section, we show how array shapes can used in order to shape the way parallelism can unfold. The scan operation serves as running example. In its simplest form, we can define a scan of an `n` element vector `a` as another `n` element vector `b` whose elements are defined by $b_i = \sum_{j<i} a_j$. This definition can be translated into the following SaC function:

```
int[n], int scan(int[n] a) {
  return ({ [i] -> sum ({[j] -> a[j] | [j] < [i]}) | [i] < [n]},
            sum (a));
}
```

For convenience, we return the complete sum of `a` as a second return value.

While the correspondence to the mathematical formulation renders this an intuitive implementation, it has one major drawback. The inherently independent specification of all elements of the result, when compiled naively, leads to `n` parallel computations of the partial sums without sharing any of them.

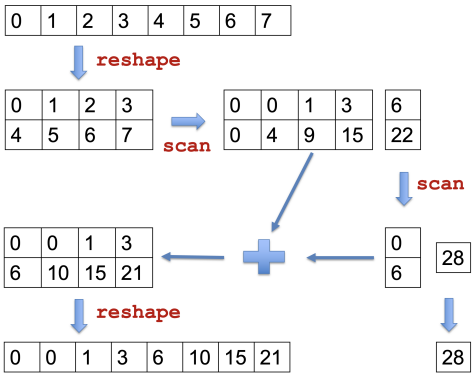
To achieve a maximum of sharing, we can specify a purely sequential version of the computation. We can do so in SaC by re-defining `a` element-wise, using a `for`-loop:

```
int[n], int scan(int[n] a) {
    s = 0;
    for (i = 0; i < shape(a)[0]; i++) {
        t = a[i];
        a[i] = s;
        s += t;
    }
    return (a, s)
}
```

This solution constitutes the other end of the spectrum of possible implementations. We do expose a maximum of sharing but expose no concurrency at all. Now let us try to target a middle-ground solution. The basic idea is that we cut the vector into sections of equal length (assuming here that that is possible). To describe this, we reshape the vector into a matrix so that the rows indeed constitute the chunks that we want to compute in parallel. Once this is done, we can apply our sequential `scan` function in parallel to each row. From the sums of the rows we then derive, again by means of a sequential scan, the values that

we need to add to each row in order to obtain the complete scan on the entire matrix.

At the example of the 8-element vector that contains the values 0 to 7, we can see how this transformation leads to the correct result:



This idea can be encoded in SaC as:

```
int[m,n], int scan(int[m,n] a) {
    a, ms = { iv -> scan (a[iv])| iv < [m]) };
    ms, s = scan (ms);
    a = a ^+ ms;
    return (a, s);
}
```

Note here, that this implementation makes use of SaC's support for ad-hoc overloading. It ensures that the recursive call actually refers to the previous, sequential implementation. Another aspect to observe is the use of the infix operation ^+. It refers to an addition of arrays with different rank, which builds on the expectation that the shape of the smaller rank array is a prefix of the shape of the larger-rank array; it replicates all the values of the smaller array to match the shape of the larger-rank array.

With this definition, the amount of parallelism and sharing can be controlled by the shape of the matrix. The more rows we have, the more parallelism we generate and the longer the row vectors, the more sharing we have. The only downside of this formulation is that the more rows we generate, the longer our sequential scan over the results of the parallel scans becomes. To avoid this sequential bottleneck but yet generate sufficient parallelism, we can recursively repeat our reshaping exercise. Instead of reshaping into a rank 2 array, we can reshape into a rank n array, again assuming that is possible. With this idea, we can now formulate our `scan` as:

```
int[n:shp,m], int scan(int[n:shp,m] a) {
    a, ms = { iv -> scan (a[iv])| iv < shp) };
    ms, s = scan (ms);
    a = a ^+ ms;
    return (a, s);
}
```

Note here that our initial parallel scan directly invokes a sequential scan on the last axis of the array `a`. This provides a large amount of parallelism on the first scan, and at the same time, renders the call `scan (ms)` a parallel scan again. As it turns out, this formulation generates the same pattern of parallel additions as Blelloch's algorithm in [12] does. However, by simply reshaping our data we can now create a wide range of parallel executions without changing the actual definition of the function `scan`. For a more detailed discussion and further specificational alternatives please see [13].

6 Controlling Blocking Through Shapes

Similarly to using array shapes and rank-polymorphic programming for steering concurrency, we can leverage this technique for multi-level blocking algorithms as well. As a prominent example, let us consider matrix multiplication. A direct translation of the mathematical definition of matrix multiplication can in SaC be written as:

```
float[m,p] matmul (float[m,n] a, float[n,p] b)
{
   return {[i,j] -> sum ({[k] -> (a[i,k] * b[k,j]) }) };
}
```

While this computes the correct results, it is well known that a naive code generation for it quickly inhibits good, let alone near-peak runtime performance. The reason lies in the poor cache locality when computing individual result elements independently. Instead, so-called blocking techniques are being used. The overall idea is to divide the matrices into matrices of sub-matrices. Then, we can re-arrange our addition of products into an outer product of matrix multiplies of the sub matrices. In SaC this means that we can block our matrices of rank 2 into matrices of rank 4. Considering a four-by-four matrix as example, this means we want to transform it into a matrix of shape `2,2,2,2]`:

shape [4,4]

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

shape [2,2,2,2]

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Such a transformation in SaC can be expressed as:

```
int[2:bshp,2:rshp] block (int[2] bshp, int[2:shp] a)
{
  rshp = shp / bshp;
  return { [i,j,k,l] -> a[rshp *[i,j] + [k,l]]
                      | [i,j,k,l] < bshp ++ rshp};
}
```

Once we have blocked matrices, we can overload the function `matmul` for the blocked case:

```
float[m,p,mm,pp] matmul (float[m,n,mm,nn] a, float[n,p,nn,pp] b)
{
   return {[i,j] -> rsum(1, {[k] -> matmul (a[i,k], b[k,j]) }) };
}
```

Notice here, that this looks almost identical to the rank 2 version. The key difference is that we do have a call to the rank 2 matrix multiply instead of the scalar multiplication. Consequently, that computation yields rank 2 arrays as results which requires us to, once again, replace the `sum` function by the more versatile rank-specific summation function `rsum`. Once this has been done, we can make `matmul` rank-polymorphic, enabling multiple levels of blocking. All that is needed to achieve that is a change in the signature of the function `matmul`:

```
float[m,p,+] matmul (float[m,n,+] a, float[n,p,+] b)
{
   return {[i,j] -> rsum(1, {[k] -> matmul (a[i,k], b[k,j]) }) };
}
```

Now, the recursive call to `matmul`, depending on the rank of `a` and `b`, either goes to the very same function definition or to the special case for rank 2 arrays.

Similar to the previous section, we can see that the rank-polymorphism enables us to define a shape generic version where the number and sizes of the blocking levels are encoded in the ishape of the data and **not** in the definition of the function `matmul` itself. As it turns out, using several levels of blocking actually enables the generation of code that is competitive with highly hand-optimised codes such as Intel's matrix multiply in the MKL library. For more details please see [14].

7 Conclusions

Rank-polymorphism proves to be more powerful than just enabling a reuse of functionality to arrays of arbitrary rank and shape. As it turns out, it is the key to encode sophisticated traversal pattern through the shape of data rather than through explicit code. In this paper, we showcase the benefits from this technique in two different contexts: the control of concurrency pattern in the context of `scan`, and the control of blocking pattern in the context of `matmul`. By liberating the algorithmic specification from the need to control these aspects we obtain a clear separation of concerns. An experimentation with different concurrency or blocking scheme turns into a reshaping of data and not into a re-write of the algorithm itself. This benefits code clarity and, with it, code maintenance as well. Furthermore, it opens the door to formal correctness proves of such algorithms as shown in [14].

These observations give rise to many new questions: Can this blocking technique be applied in the context of other algorithms from linear algebra? Can it be used to cover other code optimisation aspects? if so, which ones? Can we expand this technique to cases where we cannot find shape divisors? Would this require a more sophisticated notion of shapes? Can the proposed generalisations be introduced through systematic code transformations rat her than manual re-writes? It seems that the cases mentioned in this paper only constitute the tip of the iceberg.

References

1. Adin D. Falkoff and Kenneth E. Iverson. The design of apl. *IBM Journal of Research and Development*, 17(4):324–334, 1973.
2. Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14, 2011.
3. Todd L Veldhuizen. Arrays in blitz++. In *International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230. Springer, 1998.
4. Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.
5. Kai Trojahner and Clemens Grelck. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming*, 78(7):643–664, 2009.
6. Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*, pages 27–46. Springer, 2014.
7. Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(6):1005–1059, 2003.
8. Janice Glasgow, Michael Jenkins, Carl McCrosky, and Henk Meijer. Expressing parallel algorithms in nial. *Parallel Computing*, 11(3):331–347, 1989.
9. Clemens Grelck and Sven-Bodo Scholz. Sac—a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34:383–427, 2006.
10. Sven-Bodo Scholz and Artjoms Šinkarovs. Tensor comprehensions in SaC. In *Proceedings of the 31st Symposium on the Implementation and Application of Functional Programming Languages*, IFL '19, New York, NY, USA, 2019. ACM.
11. Artjoms Šinkarovs, Hans Viessmann, and Sven-Bodo Scholz. Array languages make neural networks fast. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2021, New York, NY, USA, 2021. ACM.
12. Guy E Blelloch. Prefix sums and their applications. 1990.
13. Artjoms Šinkarovs and Sven-Bodo Scholz. Parallel scan as a multidimensional array problem. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2022, page 1–11, New York, NY, USA, 2022. Association for Computing Machinery.
14. Artjoms Šinkarovs, Thomas Koopman, and Sven-Bodo Scholz. Rank-polymorphism for shape-guided blocking. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, FHPNC 2023, page 1–14, New York, NY, USA, 2023. Association for Computing Machinery.

LLMs in der Lehre oder "Lucy in the Sky with Diamonds"

Dietmar Schreiner[1], Jens Knoop[1], and ChatGPT[2]

[1] Institut für Information Systems Engineering, Arbeitsgruppe Compilers and Languages, Technische Universität Wien
{dietmar.schreiner, jens.knoop}@tuwien.ac.at
[2] OpenAI. (2023)
https://chat.openai.com

Abstract. Die nunmehr für jedermann nahezu freie Verfügbarkeit umfangreich trainierter großer Sprachmodelle (engl. Large Language Models (LLMs)) wie ChatGPT oder LLAMA stellt im tertiären Bildungsbereich, sowohl für Lehrende als auch für Studierende, eine enorme Herausforderung dar. Die Beurteilung der studentischen Leistung mit herkömmlichen Leistungsnachweisen, wie dem Vergleichen wissenschaftlicher Texte, dem selbstständigen Verfassen wissenschaftlicher Arbeiten, oder dem Ausarbeiten von Programmierbeispielen, erweist sich nunmehr als untauglich. Bei geschicktem Einsatz generativer künstlicher Intelligenz können ausreichende Lösungen mit minimalem Zeitaufwand und ohne eigenes Fachwissen, generiert und zur Beurteilung vorgelegt werden.
Abseits dieser Problematik ergeben sich auch auf Seite der (ernsthaft) Studierenden potentielle Probleme durch eine falsche Erwartungshaltung. Befeuert durch fragwürdige Behauptungen in unterschiedlichsten Medien, herrscht weitläufig die Meinung vor, dass derartige Systeme verstünden, was sie generieren, bzw. semantisches Wissen um das bearbeitete Thema hätten. So sind uns Fälle bekannt, in denen von ChatGPT erstellte Dokumente als Informationsquelle zu Lerninhalten verwendet wurden, ohne zu hinterfragen aus welcher Quelle die angebotene Information stammt, bzw. ob diese fachlich überhaupt korrekt ist. Dass LLMs kein tieferes semantisches Verständnis der generierten Inhalte haben und häufig auch "Fakten" extrapolieren, also halluzinieren, ist vielfach nicht bewusst.
Nichtsdestotrotz sind zahlreiche große Sprachmodelle inzwischen allgegenwärtig in Verwendung und dies wird wohl auch so bleiben. Offen ist, wie nun damit - auch im tertiären Bildungsbereich - umgegangen werden soll. In unserem Vortrag werden wir unsere bisherigen Erkenntnisse sowie unser geplantes weiteres Vorgehen zu den oben angeführten Problemen erörtern. Dazu untersuchen wir das tatsächliche Leistungsvermögen von ChatGPT aus Sicht eines Studierenden mittels einer Fallstudie bzw. werden, im Umfeld der Hochschullehre vorgeschlagene, mehr oder weniger zielführende Lösungsvorschläge diskutieren.

Type Inference for Java: Unification of Type Constraints Involving Existential Types

Andreas Stadelmeier[1,2] and Martin Plümicke[1,3]

[1] Duale Hochschule Baden-Württemberg, Campus Horb, Department of Computer Science Florianstraße 15, D–72160 Horb, Germany, https://www.dhbw-stuttgart.de/horb/forschung-transfer/forschungsschwerpunkte/typsysteme-fuer-objektorientierte-programmiersprachen

[2] a.stadelmeier@hb.dhbw-stuttgart.de

[3] pl@dhbw.de

Abstract. Java incorporates more and more type inference into the language specification. It is used in lambda expressions and method calls. Java type inference only works for limeted use cases and depends on existing type annotations, the so called target type. The algorithm proposed here extends the existing type inference for Java. Furthermore it extends existing type inference algorithms for Java with a correct handling of Java Wildcards. Our Unify algorithm solves type constraints generated by typeless Java programs. It is able to calculate correct type solutions for Java programs without any type annotations given.

1 Type Inference for Java

Type inference for Java has many use cases and could be used to help programmers by inserting correct types for them, Finding better type solutions for already typed Java programs (for example more generical ones), or allowing to write typeless Java code which is then type infered and thereby type checked by our algorithm. The algorithm proposed in this paper can determine a correct typing for the untyped Java source code example shown in figure 1a. Our algorithm is also capable of finding solutions involving wildcards as shown in figure 1b.

This paper extends a type inference algorithm for Featherweight Java ([4]) by adding wildcards. The last step to create a type inference algorithm compatible to the Java type system. The goal is to proof soundness in respect to the type rules introduced by [2] and [1].

1.1 Constraints

Constraints consist of normal types and type variables.

`List<String>` $\leqslant a$, `List<Integer>` $\leqslant a$ implies that we have to find a type replacement for type variable a, which is a supertype of `List<String>` and `List<Integer>`. In the Java language wildcards are added by replacing a parameter in a generic type by `?`. Additionally they can hold a upper or lower

```
genList() {
    if( ... ) {
        return new
            List<String>();
    } else {
        return new
            List<Integer>();
    }
}
```

(a) Java method with missing return type

```
List<?> genList() {
    if( ... ) {
        return new
            List<String>();
    } else {
        return new
            List<Integer>();
    }
}
```

(b) Correct type

bound restriction like `List<? super String>`. Our representation of this type is: `X` : [`Object`..`String`].`List<X>` Every wildcard has a name (`X` in this case) and an upper and lower bound (respectively `Object` and `String`).

c	::= `T` $\lessdot$ `T`	Constraint
`T`, `U`, `L`	::= a | `X` | `N`	Type variable or Type
`N`, `S`	::= $\overline{\texttt{X} : [\texttt{L..U}]}$.`C<`$\overline{\texttt{T}}$`>`	Class Type

This paper describes a **Unify** algorithm to solve these constraints. **Unify** computes a set of possible substitutions σ for all type variables in the input constraints.

1.2 Challenges

The introduction of wildcards adds additional challenges. Type variables can also be used as type parameters, for example `List<String>` $\lessdot$ `List<`a`>`. A problem arises when replacing type variables with wildcards.

Lets have a look at two examples:

– *Example 1.* The first one is a valid Java program. The type `List<? super String>` is *capture converted* to a fresh type variable `X` which is used as the generic method parameter `A`.
Java uses capture conversion to replace the generic `A` by a capture converted version of the `? super String` wildcard. Knowing that the type `String` is a subtype of any type the wildcard `? super String` can inherit it is safe to pass `"String"` for the first parameter of the function.

```
<A> List<A> add(A a, List<A> la) {}

    List<? super String> list = ...;
    add("String", list);
```

The constraints representing this code snippet are:

$$\texttt{String} \lessdot a,\ \texttt{X} : [\texttt{String..Object}].\texttt{List<X>} \lessdot \texttt{List<}a\texttt{>}$$

Here $\sigma(a) = \texttt{X}$ is a valid solution.
The correct solution here is to replace a with a capture converted version of `X`.

– *Example 2.* This example displays an incorrect Java program. The method call to `concat` with two wildcard lists is unsound. Each list could be of a different kind and therefore the `concat` cannot succeed.

```
<A> List<A> concat(List<A> l1, List<A> l2) {}

    List<?> list = ... ;
    concat(list, list);
```

The constraints for this example are:
$\texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>} \lessdot \texttt{List<}a\texttt{>}$,
$\texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>} \lessdot \texttt{List<}a\texttt{>}$
Remember that the given constraints cannot have a valid solution. In this example the **Unify** algorithm should not replace a with the captured wildcard `X`.

The **Unify** algorithm only sees the constraints with no information about the program they originated from. The main challenge was to find an algorithm which computes $\sigma(a) = \texttt{X}$ for example 1 but not for example 2.

2 Unify

The **Unify** algorithm computes the type solution.

Input: Constraint set $C = \{\texttt{T} \lessdot \texttt{T}, \texttt{T} \doteq \texttt{T} \ldots\}$ The input constraints must be of the following format:

c	$::=$	$\texttt{T} \lessdot \texttt{T}$	Constraint
$\texttt{T}, \texttt{U}, \texttt{L}$	$::=$	$a \mid \texttt{X} \mid \texttt{N}$	Type variable or Type
$\texttt{N}, \texttt{S}$	$::=$	$\overline{\texttt{X} : [\texttt{L}..\texttt{U}]}.\texttt{C<}\overline{\texttt{T}}\texttt{>}$	Class Type

Additional requirements:
- The input only consists of $\lessdot$ constraints
- No free variables in type parameters. $a \lessdot \texttt{X.List<X>}$ is valid.
- the input is a list of constraints. It cannot be a set. A constraint set containing the constraint $a \lessdot \texttt{T}$ twice is a different to one that contains it only once.

Output: Set of unifiers $Uni = \{\sigma_1, \ldots, \sigma_n\}$ and an environment $\mathbb{W}$

The **Unify** algorithm internally uses the following data types:

C	$::=$	$\overline{c}$	Constraint set
c	$::=$	$\texttt{T} \lessdot \texttt{T} \mid \texttt{T} \doteq \texttt{T}$	Constraint
$\texttt{T}, \texttt{U}, \texttt{L}$	$::=$	$a \mid \texttt{G}$	Type variable or Type
$\texttt{G}$	$::=$	$\texttt{X} \mid \texttt{N}$	Wildcard, or Class Type
$\texttt{N}, \texttt{S}$	$::=$	$\triangle.\texttt{C<}\overline{\texttt{T}}\texttt{>}$	Class Type
$\triangle$	$::=$	$\overline{W}$	Wildcard Environment
W	$::=$	$\texttt{X} : [\texttt{L}..\texttt{U}]$	Wildcard

With `C` being class names and `A` being wildcard names. The wildcard type $\texttt{X} : [\texttt{L}..\texttt{U}]$ consist out of an upper bound `U`, a lower bound `L` and a name `X`.

$$\text{(Upper)}\ \frac{\mathbb{W} \cup \{\mathtt{A} : [\mathtt{L}..\mathtt{U}]\} \vdash C \cup \{\mathtt{A} \lessdot \mathtt{T}\}}{\mathbb{W} \cup \{\mathtt{A} : [\mathtt{L}..\mathtt{U}]\} \vdash C \cup \{\mathtt{U} \lessdot \mathtt{T}\}}$$

$$\text{(Lower)}\ \frac{\mathbb{W} \cup \{\mathtt{A} : [\mathtt{L}..\mathtt{U}]\} \vdash C \cup \{\mathtt{T} \lessdot \mathtt{A}\}}{\mathbb{W} \cup \{\mathtt{A} : [\mathtt{L}..\mathtt{U}]\} \vdash C \cup \{\mathtt{T} \lessdot \mathtt{L}\}}$$

$$\text{(Bot)}\ \frac{\mathbb{W} \vdash C \cup \{\bot \lessdot \mathtt{T}\}}{\mathbb{W} \vdash C}$$

$$\text{(Pit)}\ \frac{\mathbb{W} \vdash C \cup \{a \lessdot \bot\}}{\mathbb{W} \vdash C \cup \{a \doteq \bot\}}$$

Fig. 2: Wildcard reduce rules

$$\text{(Equals)}\ \frac{\mathbb{W} \vdash C \cup \{\mathtt{G} \doteq \mathtt{G}'\}}{\mathbb{W} \vdash C \cup \{\mathtt{G} \lessdot \mathtt{G}', \mathtt{G}' \lessdot \mathtt{G}\}}$$

$$\text{(Erase)}\ \frac{\mathbb{W} \vdash C \cup \{\mathtt{T} \doteq \mathtt{T}\}}{\mathbb{W} \vdash C}$$

$$\text{(Erase)}\ \frac{\mathbb{W} \vdash C \cup \{\mathtt{T} \lessdot \mathtt{T}\}}{\mathbb{W} \vdash C}$$

$$\text{(Swap)}\ \frac{\mathbb{W} \vdash C \cup \{\mathtt{T} \doteq a\}}{\mathbb{W} \vdash C \cup \{a \doteq \mathtt{T}\}}$$

$$\text{(Circle)}\ \frac{\mathbb{W} \vdash C \cup \{a_1 \lessdot a_2, a_2 \lessdot a_3, \ldots, a_n \lessdot a_1\}}{\mathbb{W} \vdash C \cup \{a_1 \doteq a_2, a_2 \doteq a_3, \ldots, a_n \doteq a_1\}} \quad n > 0$$

Fig. 3: Constraint normalize rules

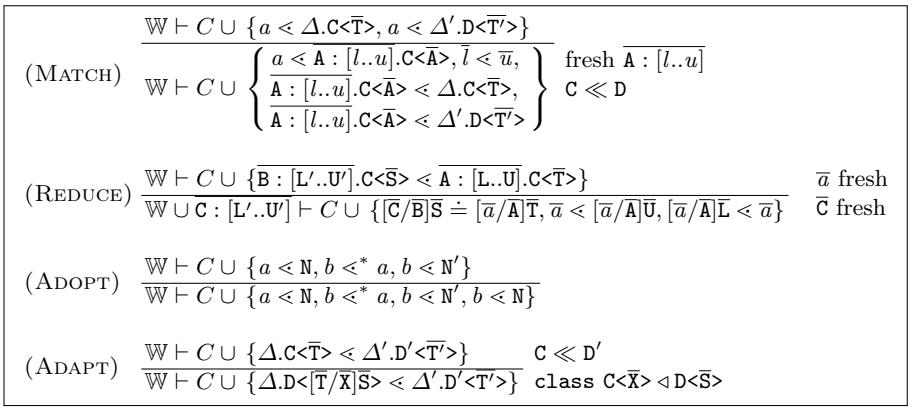


Fig. 4: Constraint reduce rules

The (MATCH) rule generates fresh wildcards $\overline{\mathtt{A} : [l..u]}$. Their upper and lower bounds are fresh type variables.

Wildcards consist out of three parts. A name, a scope and a upper and lower bound.

Wildcards are not reflexive. A box of type `X.Box<X>` is able to hold a value of any type. It could be a `Box<String>` or a `Box<Integer>` etc. Also two of those boxes do not necessarily contain the same type. But there are situations where it is possible to assume that. For example the type `X.Pair<Box<X>, Box<X>>` is a pair of two boxes, which always contain the same type. Inside the scope of the `Pair` type, the wildcard `X` stays the same.

The algorithm starts with an empty wildcard environment $\mathbb{W}$. Only the reduce rule adds wildcards to that environment. And every added wildcard gets a fresh unique name. This ensures the wildcard environment $\mathbb{W}$ never gets the same wildcard twice.

When a new type is generated by the **Unify** algorithm it always has the form $\overline{\mathtt{A}}.\mathtt{C}\langle\overline{\mathtt{A}}\rangle$.

Step 1: Apply the rules depicted in the figures 3, 4 and 5 exhaustively. Starting with the (CIRCLE) rule. Afterwards the other rules in figure 3.

The first step of the algorithm is able to remove wildcards. Removing a wildcard works by setting its lower and upper bound to be equal. (Def: $\mathtt{Object} = \mathtt{A} : [\mathtt{Object}..\mathtt{Object}]$). The (EQUALS) rule is responsible for this.

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{W} \cup \{\mathtt{X} : [l..u]\} \vdash C \cup \{\mathtt{Object} \doteq \mathtt{X}, l \lessdot u\}}}{\mathbb{W} \cup \{\mathtt{X} : [l..u]\} \vdash C \cup \{\mathtt{Object} \lessdot \mathtt{X}, \mathtt{X} \lessdot \mathtt{Object}, l \lessdot u\}}}{\mathbb{W} \cup \{\mathtt{X} : [l..u]\} \vdash C \cup \{\mathtt{Object} \lessdot l, u \lessdot \mathtt{Object}, l \lessdot u\}}}{\dfrac{\ldots}{\mathbb{W} \cup \{\mathtt{X} : [\mathtt{Object}..\mathtt{Object}]\} \vdash C}}$$

$\ll$ relation: The $\ll$ relation is the reflexive and transitive closure of the `extends` relations:

$$\frac{\texttt{C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{D<}\overline{\texttt{N}}\texttt{>}}{\texttt{C} \ll \texttt{D}} \qquad \frac{}{\texttt{C} \ll \texttt{C}} \qquad \frac{\texttt{C} \ll \texttt{D}, \texttt{D} \ll \texttt{E}}{\texttt{C} \ll \texttt{E}}$$

The algorithm uses it to determine if two types are possible subtypes of one another. This is needed in the (ADAPT) and (MATCH) rules.

Wildcard renaming: The (REDUCE) rule separates wildcards from their environment. At this point each wildcard gets a new and unique name. To only rename the respective wildcards the reduce rule renames wildcards up to alpha conversion: ($[\overline{\texttt{C}}/\overline{\texttt{B}}]$ in the (REDUCE) rule)

$$\begin{aligned}
[\texttt{A}/\texttt{B}]\texttt{B} &= \texttt{A} \\
[\texttt{A}/\texttt{B}]\texttt{C} &= \texttt{C} && \text{if } \texttt{B} \neq \texttt{C} \\
[\texttt{A}/\texttt{B}]\Delta.\texttt{N} &= \Delta.[\texttt{A}/\texttt{B}]\texttt{N} && \text{if } \texttt{B} \notin \Delta \\
[\texttt{A}/\texttt{B}]\Delta.\texttt{N} &= \Delta.\texttt{N} && \text{if } \texttt{B} \in \Delta
\end{aligned}$$

Fresh Wildcards: fresh $\overline{\texttt{A} : [l..u]}$ generates fresh wildcards. The new names $\overline{\texttt{A}}$ are fresh, aswell as the type variables $\overline{u}$ and $\overline{l}$, which are used for the upper and lower bounds.

Step 2: We apply the rules (subst) and (ground) exhaustively to C'':

$$\text{(SUBST)} \quad \frac{\mathbb{W} \vdash C \cup \{a \doteq \texttt{T}\}}{[\texttt{T}/a]\mathbb{W} \vdash [\texttt{T}/a]C \cup \{a \doteq \texttt{T}\}}\ a \notin \texttt{T}$$

$$\text{(GROUND)} \quad \frac{\mathbb{W} \vdash C \cup \{a \lessdot \texttt{T}, a \lessdot \texttt{S}\}}{\mathbb{W} \vdash C \cup \{a \doteq \bot\}}$$

We fail if we find any $a \doteq \texttt{T}$ such that a occurs in $\texttt{T}$.

If atleast one substitution is possible the algorithm returns to **step 1**.

Step 3:

If there are no ($\texttt{T} \lessdot a$) constraints remaining in the constraint set C and C is in solved form then C is a valid solution.

Otherwise for every $\texttt{T} \lessdot a$ constraint, the unify algorithm has to consider every possible supertype of $\texttt{T}$. There are two different ways of handling a $\texttt{T} \lessdot a$ constraint:

$$\textbf{(SAME)} \quad \frac{\mathbb{W} \vdash C \cup \Delta.\texttt{C<}\overline{\texttt{T}}\texttt{>} \lessdot a}{\mathbb{W} \vdash C \cup \{\Delta.\texttt{C<}\overline{\texttt{T}}\texttt{>} \lessdot a, a \doteq \overline{\texttt{X} : [l..u]}.\texttt{C<}\overline{\texttt{X}}\texttt{>}, \}} \quad \begin{array}{l}\texttt{class C<}\overline{\texttt{X}}\texttt{>} \triangleleft \texttt{D<}\overline{\texttt{N}}\texttt{>} \\ \text{fresh } \overline{\texttt{X} : [l..u]}\end{array}$$

$$\textbf{(SUPER)} \quad \frac{\mathbb{W} \vdash C \cup \Delta.\texttt{C<}\overline{\texttt{T}}\texttt{>} \lessdot a}{\mathbb{W} \vdash C \cup \{\Delta.\texttt{D<}[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}\texttt{>} \lessdot a\}} \quad \texttt{class C<}\overline{\texttt{X}}\texttt{>} \triangleleft \texttt{D<}\overline{\texttt{N}}\texttt{>}$$

The first one attempts to give a a more general type by replacing only the type parameters with fresh wildcards. The second variation sets a to the direct

supertype of type C. For the constraint $\texttt{Object} \lessdot a$ the algorithm can only apply $a \doteq \texttt{Object}$, because Object has no other supertype than itself.

Constraints of the form $\{a \lessdot \texttt{N}, a \lessdot^* b\}$ need to be handled in a similiar fashion. The type variable b could either be a sub or a supertype of the type N in this scenario. We have to consider both possibilities:

1. $\dfrac{\mathbb{W} \vdash C \cup \{a \lessdot \texttt{N}, a \lessdot^* b\}}{\mathbb{W} \vdash C \cup \{a \lessdot^* b, b \lessdot \texttt{N}\}}$ (Lower)
2. $\dfrac{\mathbb{W} \vdash C \cup \{a \lessdot \texttt{N}, a \lessdot^* b\}}{\mathbb{W} \vdash C \cup \{a \lessdot \texttt{N}, a \lessdot^* b, \texttt{N} \lessdot b\}}$ (Raise)

Constraints of the form $\{a \lessdot \texttt{T}\}$ with $\text{fv}(\texttt{T}) \neq \emptyset$ leave two options. The wildcards in question must be negated by giving them the same upper and lower bound or the type variable a becomes the bottom type $\bot$, which is a subtype of every type. Another possible subtype would be the type T itself, but this would mean to substitute a type with free wildcard variables ($a \doteq \texttt{T}$).

1. $\dfrac{\mathbb{W} \cup \{\texttt{X} : [\texttt{L..U}]\} \vdash C \cup \{a \lessdot \Delta.\texttt{N}\}}{\mathbb{W} \cup \{\texttt{X} : [\texttt{L..U}]\} \vdash C \cup \{a \lessdot \Delta'.\texttt{N}, \texttt{U} \lessdot \texttt{L}, \texttt{L} \lessdot \texttt{U}\}}$ $\texttt{X} \in \text{fv}(\texttt{N})$ $\Delta' = \Delta \cup \{\texttt{X} : [\texttt{L..U}]\}$ (Force)
2. $\dfrac{\mathbb{W} \cup \{\texttt{X} : [\texttt{L..U}]\} \vdash C \cup \{a \lessdot \texttt{T}\}}{\mathbb{W} \vdash C \cup \{a \doteq \bot\}}$ $\texttt{X} \in \text{fv}(\texttt{T})$

The specification of the **Unify** algorithm has only two rules generating $\doteq$-Constraints , (Reduce) and (Ground). $\doteq$-Constraints and the accompaning substitutions are dangerous respective to the soundness of the algorithm. For the soundness proof of the **Unify** algorithm we have to show every generation of equals constraints and the subsequent application of the (subst) rule is correct. We try to use them as sparsely as possible to simplify the soundness proof. You can notice this at (Equals) or (Force): Instead of setting $\texttt{U} \doteq \texttt{L}$, we say $\texttt{U} \lessdot \texttt{L}, \texttt{L} \lessdot \texttt{U}$.

Step 4: Eliminate subtyping constraints between variables by exhaustive application of rule (sub-elim) and (erase). Applying this rule does not affect the solve form property.

$$\text{(sub-elim)}\ \frac{C \cup \{a \lessdot b\}}{[a/b]C \cup \{b \doteq a\}}$$

Afterwards create the output (γ, σ) with:

$$\begin{aligned}\sigma' &= \{b \mapsto \texttt{B} \mid (b \lessdot \texttt{T}) \in C\}\\ \sigma &= \{a \mapsto \sigma'(T) \mid (a \doteq T) \in C\}\\ \gamma &= \mathbb{W} \cup \{\texttt{B} : [\bot..\sigma(\texttt{T})] \mid (b \lessdot \texttt{T}) \in C\}\end{aligned}$$

Solved form A set C of constraints is in solved form if it only contains constraints of the following form:

1. $a \lessdot b$
2. $a \doteq b$

3. $a \doteq \texttt{A}$
4. $a \lessdot \overline{W}.\texttt{C<}\overline{\texttt{T}}\texttt{>}$
5. $a \doteq \overline{W}.\texttt{C<}\overline{\texttt{T}}\texttt{>}$, with $a \notin \overline{\texttt{T}}$

In case 4 and 5 the type variable a does not appear on the left of another constraint of the form 4 or 5.

3 High-Level rules

The **Unify** specification tries to be as simple as possible with each rule doing only one simple transformation. We define additional transformation rules, which deviate directly from the given algorithm. They come to use in the examples section.

$$\text{(Encase)} \quad \frac{\mathbb{W} \vdash C \cup \{\texttt{C<T>} \lessdot \texttt{X} : [\texttt{L..U}].\texttt{C<X>}\}}{\mathbb{W} \vdash [\texttt{T}/x]\, C \cup \{\texttt{T} \lessdot \texttt{U}, \texttt{L} \lessdot \texttt{T}\}}$$

$$\text{(Flatten)} \quad \frac{\mathbb{W} \vdash C \cup \{\texttt{T} \lessdot a, a \lessdot \texttt{T}\}}{\mathbb{W} \vdash [\texttt{T}/a]\, C \cup \{a \doteq \texttt{T}\}}$$

$$\text{(Assimilate)} \quad \frac{\mathbb{W} \vdash C \cup \{\texttt{X} : [l..u].\texttt{C<X>} \lessdot \texttt{C<T>}, l \lessdot u\}}{\mathbb{W} \vdash C \cup \{u \doteq \texttt{T}, l \doteq \texttt{T}\}}$$

$$\text{(Narrow)} \quad \frac{\mathbb{W} \vdash C \cup \{\texttt{X} : [\texttt{L..U}].\texttt{C<X>} \lessdot \texttt{X} : [\texttt{L}'..\texttt{U}'].\texttt{C<X>}\}}{\mathbb{W} \vdash C \cup \{\texttt{L}' \lessdot \texttt{L}, \texttt{U} \lessdot \texttt{U}'\}}$$

$$\text{(Redeem)} \quad \frac{\mathbb{W} \vdash C \cup \{\Delta.\texttt{C<X>} \lessdot \texttt{X} : [\bot..\texttt{Object}].\texttt{C<X>}\}}{\mathbb{W} \vdash C}$$

$$\text{(Standoff)} \quad \frac{\mathbb{W} \cup \{\texttt{X} : [\texttt{L..U}], \texttt{Y} : [\texttt{L}'..\texttt{U}']\} \vdash \texttt{X} \doteq \texttt{Y}}{\mathbb{W} \cup \{\texttt{X} : [\texttt{L..U}], \texttt{Y} : [\texttt{L}'..\texttt{U}']\} \vdash \texttt{U} \lessdot \texttt{L}', \texttt{U}' \lessdot \texttt{L}}$$

Fig. 5: Common transformations

4 Related Work

In this section we consider the older Java type unification approach [3, 5] where we do not consider the so-called capture conversion. This approach is integrated in Java–TX at the moment.
The approach is to continue the subtyping ordering on wildcard-types. Therefore we considered the semantics of wildcard-types:

`? extends` θ**:** There is a subtype of θ.

`? super` θ'**:** There is a supertype of θ'.

From this follows the definition: For $\theta <: \theta'$ holds

$$\theta \ <: \ \texttt{? super}\,\theta',$$

$$\texttt{? extends}\,\theta \ <: \ \theta', \text{ and}$$

$$\texttt{? extends}\,\theta \ <: \ \texttt{? super}\,\theta'.$$

This definition guarantees that $<:$ is not reflexiv on wildcard-types.
Let us consider Example 5 and Example 5 again. The add method is no problem as `String` $<:$ `? super String`.
But the `concat` method would not work. The type inference algorithm would infer the following signature:

```
<A,B extends A> void concat(List<A> l1, List<B> l2) {
    for (B x:l2) l1.add(x);
}
```

A method call

rec`.<? extends Object, ? extends Object>conat(list, list)`

would fail as `? extends Object` $\not<:$ `? extends Object`.
This means Java–TX has two drawbacks. On the one hand a standard Java method

```
<A> void concat(List<A> l1, List<A> l2) { ... }
```

can not be called with wildcards in general as capture conversion is necessary to guarantee soundness. On the other hand a call with the same parameter is not possible even if the method is type inferred by Java–TX.

But there are some pros in the Java–TX approach.

Let us consider the following `print` method:

```
<T> void print(Vector<T> v1, Vector<T> v2) {
  System.out.println(v1);
  System.out.println(v2);
}

Vector<?> v1 = ...
Vector<?> v2 = ...
print(v1, v2);
```

The method call would fail as the capture conversion produces two different fresh type variables for the wildcards, respectively. This program can be made correct using two different type variables:

```
<T,S> void print(Vector<T> v1, Vector<S> v2).
```

But in the following example this is not possible.

```
<X> void assign(Vector<X> v1, Vector<X> v2) {
   v1 = v2;
}

void main() {
  Vector<?> v1 = ...;
  Vector<?> v2 = ...;
  v1 = v2; //OK
  assign(v1, v2); //not type correct but sound
}
```

As `Vector<?>` is a subtype of itself the assigment `v1 = v2;` as well as the method call `assign(v1, v2);` are type correct. Therefore in Java–TX this would be correct.

5 Examples

Example 1

```
<A> List<A> add(A a, List<A> la) {}
```

```
List<? super String> l;
add("String", l);
```

Constraints:
$\texttt{String} \lessdot a$,
$\texttt{X} : [\texttt{String}..].\texttt{List<X>} \lessdot \texttt{List}\langle a \rangle$

$$\frac{\frac{\frac{\frac{\texttt{String} \lessdot a, \texttt{X} : [\texttt{String}..\texttt{Object}].\texttt{List<X>} \lessdot \texttt{List}\langle a \rangle}{\texttt{X} : [\texttt{String}..\texttt{Object}] \vdash \texttt{String} \lessdot a, \texttt{X} \doteq a}\,(\textsc{Reduce})}{\texttt{X} : [\texttt{String}..\texttt{Object}] \vdash \texttt{String} \lessdot a, a \doteq \texttt{X}}\,(\textsc{Swap})}{\texttt{X} : [\texttt{String}..\texttt{Object}] \vdash \texttt{String} \lessdot \texttt{X}, a \doteq \texttt{X}}\,(\textsc{Subst})}{\texttt{X} : [\texttt{String}..\texttt{Object}] \vdash \texttt{String} \lessdot \texttt{String}, a \doteq \texttt{X}}\,(\textsc{Lower})}{\texttt{X} : [\texttt{String}..\texttt{Object}] \vdash \cancel{\texttt{String} \lessdot \texttt{X}}, a \doteq \texttt{X}}\,(\textsc{Erase})$$

The constraint set $C = \{a \doteq \texttt{X}\}$ is in solved form and **Unify** terminates.

Example 2

```
<A> void concat(List<A> l1, List<A> l2) {...}
```

```
List<?> l = ...;
concat(l, l)
```

Constraints:
$\texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>} \lessdot \texttt{List}\langle Z \rangle$,
$\texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>} \lessdot \texttt{List}\langle Z \rangle$

This example shows an ill typed Java code snippet. `concat` requires two lists of the same type. The goal is to replace `Z` with a type, which suffices the constraints $\texttt{X.List<X>} \lessdot \texttt{List}\langle Z \rangle, \texttt{X.List<X>} \lessdot \texttt{List}\langle Z \rangle$. This is not possible. Replacing Z with `X` is not correct. A wildcard is only valid inside its scope.

This seems unintuitive here, because `concat` is given the same exact list twice. Nonetheless the **Unify** algorithm has to spot the error here. The reduce rule performs a capture conversion every time a wildcard type is unpacked.

As you can see the equality relation (=) is not reflexive on the types used here. Despite the subtype relation (<:) being reflexive. `T` <: `T` holds, but `T` $\neq$ `T` does not. This is the reason the **Unify** algorithm cannot use sets and has to store the constraints in a set. The constraints $a \lessdot \texttt{T}, a \lessdot \texttt{T}$ have a different meaning than the constraint $a \lessdot \texttt{T}$ alone.

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{X.List<X>} \lessdot \texttt{List<}z\texttt{>}, \texttt{X.List<X>} \lessdot \texttt{List<}z\texttt{>}}{\texttt{A} : [\bot..\texttt{Object}], \texttt{B} : [\bot..\texttt{Object}] \vdash \texttt{A} \doteq z, \texttt{B} \doteq z}\,(\textsc{Reduce})}{\texttt{A} : [\bot..\texttt{Object}], \texttt{B} : [\bot..\texttt{Object}] \vdash z \doteq \texttt{A}, \texttt{B} \doteq z}\,(\textsc{Swap})}{\texttt{A} : [\bot..\texttt{Object}], \texttt{B} : [\bot..\texttt{Object}] \vdash z \doteq \texttt{A}, \texttt{B} \doteq \texttt{A}}\,(\textsc{Subst})}{\texttt{A} : [\bot..\texttt{Object}], \texttt{B} : [\bot..\texttt{Object}] \vdash z \doteq \texttt{A}, \bot \lessdot \texttt{Object}, \bot \lessdot \texttt{Object}}\,(\textsc{Standoff})$$

Error: $\texttt{Object} \lessdot \bot$

5.1 Example Pair

```
<X, Y> Pair<X,Y> make(List<X> l1, List<Y> l2) {...}
<Z> Y compare(Pair<Z,Z> p) {...}
```

```
void m(List<?> l) {
  compare(make(l, l));
}
```

Constraints:
$\texttt{A} : [\bot..\texttt{Object}].\texttt{List<A>} \lessdot \texttt{List<}x\texttt{>}$,
$\texttt{A} : [\bot..\texttt{Object}].\texttt{List<A>} \lessdot \texttt{List<}y\texttt{>}$,
$\texttt{Pair<}x, y\texttt{>} \lessdot \texttt{Pair<}z, z\texttt{>}$

Unify annotates fresh wildcard names during the reduce step. As soon as a wildcard leaves its scope it has to get an unique name. This is the reason this example has no solution. The input program is incorrect.

$$\dfrac{\dfrac{\dfrac{\texttt{A} : [\bot..\texttt{Object}].\texttt{List<A>} \lessdot \texttt{List<}x\texttt{>},\ \texttt{A} : [\bot..\texttt{Object}].\texttt{List<A>} \lessdot \texttt{List<}y\texttt{>},\ \texttt{Pair<}x, y\texttt{>} \lessdot \texttt{Pair<}y, y\texttt{>}}{\texttt{B} : [\bot..\texttt{Object}],\ \texttt{B} \doteq x,\ \texttt{C} : [\bot..\texttt{Object}] \vdash \texttt{C} \doteq y,\ x \doteq z, y \doteq z}\,(\textsc{Reduce})}{\texttt{B} : [\bot..\texttt{Object}], \texttt{C} : [\bot..\texttt{Object}] \vdash \texttt{B} \doteq \texttt{C}, \ldots}\,(\ldots)}{\texttt{Object} \lessdot \bot, \texttt{Object} \lessdot \bot, \ldots}\,(\textsc{Standoff})$$

Error: $\texttt{Object} \lessdot \bot$

5.2 Capture Conversion Example

```
<X> Pair<X,X> make(List<X> l) {...}
<Y> Y compare(Pair<Y,Y> p) {...}
```

```
m(List<?> l) {
  return compare(make(l));
}
```

Constraints:
$\texttt{A} : [\bot..\texttt{Object}].\texttt{List<A>} \lessdot \texttt{List}\langle x\rangle$,
$\texttt{Pair}\langle x, x\rangle \lessdot \texttt{Pair}\langle y, y\rangle$,
$y \lessdot m$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{A} : [\bot..\texttt{Object}].\texttt{List<A>} \lessdot \texttt{List}\langle x\rangle,\ \texttt{Pair}\langle x, x\rangle \lessdot \texttt{Pair}\langle y, y\rangle, y \lessdot m}{\texttt{B} : [\bot..\texttt{Object}] \vdash \texttt{B} \doteq x, x \doteq y, y \lessdot m}\ (\textsc{Reduce})}{\texttt{B} : [\bot..\texttt{Object}] \vdash x \doteq \texttt{B}, x \doteq y, y \lessdot m}\ (\textsc{Swap})}{\texttt{B} : [\bot..\texttt{Object}] \vdash x \doteq \texttt{B}, \texttt{B} \doteq y, y \lessdot m}\ (\textsc{Subst})}{\texttt{B} : [\bot..\texttt{Object}] \vdash x \doteq \texttt{B}, y \doteq \texttt{B}, y \lessdot m}\ (\textsc{Swap})}{\texttt{B} : [\bot..\texttt{Object}] \vdash x \doteq \texttt{B}, y \doteq \texttt{B}, \texttt{B} \lessdot m}\ (\textsc{Subst})}{\texttt{B} : [\bot..\texttt{Object}] \vdash x \doteq \texttt{B}, y \doteq \texttt{B}, \texttt{Object} \lessdot m}\ (\textsc{Upper})$$

Solution:

```
Object m(List<?> l) {
  return compare(make(l));
}
```

5.3 Method Parameter Example

Full example of wildcards being used in method parameters.

```
m(l, la, lb){
    la.add(1);
    lb.add("abc");
    ll.add(la);
    ll.add(lb);
}
```

Constraints:
$la \lessdot \texttt{List}\langle a\rangle, \texttt{Integer} \lessdot a, a \lessdot \texttt{Object}$
$lb \lessdot \texttt{List}\langle b\rangle, \texttt{String} \lessdot b, b \lessdot \texttt{Object}$
$ll \lessdot \texttt{List}\langle x\rangle, la \lessdot x$
$ll \lessdot \texttt{List}\langle y\rangle, lb \lessdot y$

We outline the steps needed to create a correct solution for this constraint set. Hereby we take the direct route by always applying the correct transformations. The **Unify** algorithm usually can't do that and has to try multiple possibilities (by backtracking for example).

First we show the transformation of the constraints $la \lessdot \texttt{List}\langle a\rangle, \texttt{Integer} \lessdot a, a \lessdot \texttt{Object}$.

$$\frac{la \lessdot \texttt{List<}a\texttt{>},\ \texttt{Integer} \lessdot a,\ a \lessdot \texttt{Object}}{la \lessdot \texttt{List<}a\texttt{>},\ \texttt{Integer} \lessdot a,\ a \doteq \texttt{Integer},\ a \lessdot \texttt{Object}}\ (\textsc{Same})$$

$$\frac{la \lessdot \texttt{List<}a\texttt{>},\ \texttt{Integer} \lessdot a,\ a \doteq \texttt{Integer},\ a \lessdot \texttt{Object}}{la \lessdot \texttt{List<Integer>},\ \texttt{Integer} \lessdot \texttt{Integer},\ a \doteq \texttt{Integer},\ \texttt{Integer} \lessdot \texttt{Object}}\ (\textsc{Subst})$$

$$\frac{la \lessdot \texttt{List<Integer>},\ \texttt{Integer} \lessdot \texttt{Integer},\ a \doteq \texttt{Integer},\ \texttt{Integer} \lessdot \texttt{Object}}{la \lessdot \texttt{List<Integer>},\ a \doteq \texttt{Integer},\ \texttt{Integer} \lessdot \texttt{Object}}\ (\textsc{Erase})$$

$$\frac{la \lessdot \texttt{List<Integer>},\ a \doteq \texttt{Integer},\ \texttt{Integer} \lessdot \texttt{Object}}{la \lessdot \texttt{List<Integer>},\ a \doteq \texttt{Integer},\ \texttt{Object} \lessdot \texttt{Object}}\ (\textsc{Super})$$

$$\frac{la \lessdot \texttt{List<Integer>},\ a \doteq \texttt{Integer},\ \texttt{Object} \lessdot \texttt{Object}}{la \lessdot \texttt{List<Integer>},\ a \doteq \texttt{Integer}}\ (\textsc{Erase})$$

Result: $la \lessdot \texttt{List<Integer>}, a \doteq \texttt{Integer}$

The same sequence of rules applies for the $lb \lessdot \texttt{List<}b\texttt{>}, \texttt{Integer} \lessdot b, b \lessdot \texttt{Object}$ constraints, leading to constraint set:

$la \lessdot \texttt{List<}a\texttt{>}$,		$la \lessdot \texttt{List<Integer>}$,
$\texttt{Integer} \lessdot a, a \lessdot \texttt{Object}$,		$lb \lessdot \texttt{List<String>}$,
$lb \lessdot \texttt{List<}b\texttt{>}$,		$a \doteq \texttt{Integer}$
$\texttt{String} \lessdot b, b \lessdot \texttt{Object}$,	$\rightarrow$	$b \doteq \texttt{String}$
$ll \lessdot \texttt{List<}x\texttt{>}$,		$ll \lessdot \texttt{List<}x\texttt{>}$,
$la \lessdot x$,		$la \lessdot x$,
$ll \lessdot \texttt{List<}y\texttt{>}$,		$ll \lessdot \texttt{List<}y\texttt{>}$,
$lb \lessdot y$		$lb \lessdot y$

Now we have a look at the constraints $la \lessdot \texttt{List<Integer>}, la \lessdot x, ll \lessdot \texttt{List<}x\texttt{>}$

$$\frac{la \lessdot \texttt{List<Integer>},\ la \lessdot x,\ ll \lessdot \texttt{List<}x\texttt{>}}{la \lessdot \texttt{List<Integer>},\ la \lessdot x,\ \texttt{List<Integer>} \lessdot x,\ ll \lessdot \texttt{List<}x\texttt{>}}\ (\textsc{Raise})$$

$$\frac{la \lessdot \texttt{List<Integer>},\ la \lessdot x,\ \texttt{List<Integer>} \lessdot x,\ ll \lessdot \texttt{List<}x\texttt{>}}{la \lessdot \texttt{List<Integer>},\ la \lessdot x,\ \texttt{List<Integer>} \lessdot x,\ x \doteq \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<}x\texttt{>}}\ (\textsc{Same})$$

$$\frac{la \lessdot \texttt{List<Integer>},\ la \lessdot x,\ \texttt{List<Integer>} \lessdot x,\ x \doteq \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<}x\texttt{>}}{la \lessdot \texttt{List<Integer>},\ la \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<}\texttt{X} : [l..u].\texttt{List<X>}\texttt{>},\ \texttt{List<Integer>} \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ x \doteq \texttt{X} : [l..u].\texttt{List<X>}}\ (\textsc{Subst})$$

$$\frac{la \lessdot \texttt{List<Integer>},\ la \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<}\texttt{X} : [l..u].\texttt{List<X>}\texttt{>},\ \texttt{List<Integer>} \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ x \doteq \texttt{X} : [l..u].\texttt{List<X>}}{la \lessdot \texttt{List<Integer>},\ la \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<X} : [l..u].\texttt{List<X>>},\ \texttt{Integer} \doteq x', x' \lessdot u, l \lessdot x',\ x \doteq \texttt{X} : [l..u].\texttt{List<X>}}\ (\textsc{Reduce})$$

$$\frac{la \lessdot \texttt{List<Integer>},\ la \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<X} : [l..u].\texttt{List<X>>},\ \texttt{Integer} \doteq x', x' \lessdot u, l \lessdot x',\ x \doteq \texttt{X} : [l..u].\texttt{List<X>}}{la \lessdot \texttt{List<Integer>},\ la \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<X} : [l..u].\texttt{List<X>>},\ x' \doteq \texttt{Integer},\ x' \lessdot u, l \lessdot x',\ x \doteq \texttt{X} : [l..u].\texttt{List<X>}}\ (\textsc{Swap})$$

$$\frac{la \lessdot \texttt{List<Integer>},\ la \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<X} : [l..u].\texttt{List<X>>},\ x' \doteq \texttt{Integer},\ x' \lessdot u, l \lessdot x',\ x \doteq \texttt{X} : [l..u].\texttt{List<X>}}{la \lessdot \texttt{List<Integer>},\ la \lessdot \texttt{X} : [l..u].\texttt{List<X>},\ ll \lessdot \texttt{List<X} : [l..u].\texttt{List<X>>},\ x' \doteq \texttt{Integer},\ \texttt{Integer} \lessdot u, l \lessdot \texttt{Integer},\ x \doteq \texttt{X} : [l..u].\texttt{List<X>}}\ (\textsc{Subst})$$

Note: At this point we could set $u \doteq \texttt{Integer}$, to satisfy the $\texttt{Integer} \lessdot u$ constraint. But this leads to an unsolvable constraint set. The right way is to set the upper bound u to `Object`:

Step 2

$la \lessdot \texttt{List<Integer>}$, $lb \lessdot \texttt{List<String>}$, $a \doteq \texttt{Integer}$ $b \doteq \texttt{String}$ $ll \lessdot \texttt{List<}x\texttt{>}$, $la \lessdot x$, $ll \lessdot \texttt{List<}y\texttt{>}$, $lb \lessdot y$	$la \lessdot \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}$, $x' \doteq \texttt{Integer}$, $x \doteq \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}$, $la \lessdot \texttt{List<Integer>}$, $lb \lessdot \texttt{List<String>}$, $a \doteq \texttt{Integer}$ $b \doteq \texttt{String}$ $ll \lessdot \texttt{List<X} : [\bot..\texttt{Object}].\texttt{List<X>>}$, $ll \lessdot \texttt{List<}y\texttt{>}$, $lb \lessdot y$, $u \doteq \texttt{Object}$, $l \doteq \bot$

Now we apply the match rule to
$la \lessdot \texttt{List<Integer>}$,
$la \lessdot \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}$

$$\frac{la \lessdot \texttt{List<Integer>}, la \lessdot \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}}{\begin{array}{l} la \lessdot \texttt{Z} : [l''..u''].\texttt{List<Z>}, l'' \lessdot u'', \\ \texttt{Z} : [l''..u''].\texttt{List<Z>} \lessdot \texttt{List<Integer>}, \\ \texttt{Z} : [l''..u''].\texttt{List<Z>} \lessdot \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>} \end{array}} \text{(Match)}$$

$$\frac{\begin{array}{l} la \lessdot \texttt{Z} : [l''..u''].\texttt{List<Z>}, l'' \lessdot u'', \\ \texttt{Z} : [l''..u''].\texttt{List<Z>} \lessdot \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>} \end{array}}{\begin{array}{l} la \lessdot \texttt{Z} : [l''..u''].\texttt{List<Z>}, l'' \lessdot u'', \\ \texttt{Z} : [l''..u''].\texttt{List<Z>} \lessdot \texttt{List<Integer>} \end{array}} \text{(Redeem)}$$

$$\frac{\begin{array}{l} la \lessdot \texttt{Z} : [l''..u''].\texttt{List<Z>}, l'' \lessdot u'', \\ \texttt{Z} : [l''..u''].\texttt{List<Z>} \lessdot \texttt{List<Integer>} \end{array}}{\begin{array}{l} la \lessdot \texttt{Z} : [l''..u''].\texttt{List<Z>}, l'' \lessdot u'', \\ u'' \doteq \texttt{Integer}, l'' \doteq \texttt{Integer} \end{array}} \text{(Assimilate)}$$

$$\frac{\begin{array}{l} la \lessdot \texttt{Z} : [l''..u''].\texttt{List<Z>}, l'' \lessdot u'', \\ u'' \doteq \texttt{Integer}, l'' \doteq \texttt{Integer} \end{array}}{\begin{array}{l} la \lessdot \texttt{Z} : [\texttt{Integer}..\texttt{Integer}].\texttt{List<Z>}, \\ \texttt{Integer} \lessdot \texttt{Integer}, \\ u'' \doteq \texttt{Integer}, l'' \doteq \texttt{Integer} \end{array}} \text{(Subst)}$$

$$\frac{\begin{array}{l} la \lessdot \texttt{Z} : [\texttt{Integer}..\texttt{Integer}].\texttt{List<Z>}, \\ \texttt{Integer} \lessdot \texttt{Integer}, \\ u'' \doteq \texttt{Integer}, l'' \doteq \texttt{Integer} \end{array}}{\begin{array}{l} la \lessdot \texttt{Z} : [\texttt{Integer}..\texttt{Integer}].\texttt{List<Z>}, \\ u'' \doteq \texttt{Integer}, l'' \doteq \texttt{Integer} \end{array}} \text{(Erase)}$$

To recapitulate the complete transformation of the constraints

$la \lessdot \texttt{List<Integer>}, la \lessdot x, ll \lessdot \texttt{List<}x\texttt{>}$:

$$
\begin{array}{l}
\\
la \lessdot \texttt{List<Integer>}, \\
lb \lessdot \texttt{List<String>}, \\
a \doteq \texttt{Integer} \\
b \doteq \texttt{String} \\
ll \lessdot \texttt{List<}x\texttt{>}, \\
la \lessdot x, \\
ll \lessdot \texttt{List<}y\texttt{>}, \\
lb \lessdot y
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
x \doteq \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}, \\
la \lessdot \texttt{Z} : [\texttt{Integer..Integer}].\texttt{List<Z>}, \\
lb \lessdot \texttt{List<String>}, \\
a \doteq \texttt{Integer} \\
b \doteq \texttt{String} \\
ll \lessdot \texttt{List<X} : [\bot..\texttt{Object}].\texttt{List<X>>}, \\
ll \lessdot \texttt{List<}y\texttt{>}, \\
lb \lessdot y \\
u \doteq \texttt{Object}, \\
l \doteq \bot
\end{array}
$$

Applying the same rules analogously to $lb \lessdot \texttt{List<String>}$, $lb \lessdot y$, $ll \lessdot \texttt{List<}y\texttt{>}$:

$$
\begin{array}{l}
\\
la \lessdot \texttt{List<Integer>}, \\
lb \lessdot \texttt{List<String>}, \\
a \doteq \texttt{Integer} \\
b \doteq \texttt{String} \\
ll \lessdot \texttt{List<}x\texttt{>}, \\
la \lessdot x, \\
ll \lessdot \texttt{List<}y\texttt{>}, \\
lb \lessdot y
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
x \doteq \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}, \\
la \lessdot \texttt{Z} : [\texttt{Integer..Integer}].\texttt{List<Z>}, \\
y \doteq \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}, \\
lb \lessdot \texttt{Z} : [\texttt{Integer..Integer}].\texttt{List<Z>}, \\
a \doteq \texttt{Integer} \\
b \doteq \texttt{String} \\
ll \lessdot \texttt{List<X} : [\bot..\texttt{Object}].\texttt{List<X>>}, \\
ll \lessdot \texttt{List<X} : [\bot..\texttt{Object}].\texttt{List<X>>}, \\
u \doteq \texttt{Object}, \\
l \doteq \bot
\end{array}
$$

The last step is to apply the match rule to the constraints
$ll \lessdot \texttt{List<X} : [\bot..\texttt{Object}].\texttt{List<X>>}$ $ll \lessdot \texttt{List<X} : [\bot..\texttt{Object}].\texttt{List<X>>}$

This leaves us with a constraint set, which is in solved form and therefore a valid solution.

$$
\begin{array}{l}
x \doteq \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}, \\
la \lessdot \texttt{Z} : [\texttt{Integer..Integer}].\texttt{List<Z>}, \\
y \doteq \texttt{X} : [\bot..\texttt{Object}].\texttt{List<X>}, \\
lb \lessdot \texttt{Z} : [\texttt{Integer..Integer}].\texttt{List<Z>}, \\
a \doteq \texttt{Integer} \\
b \doteq \texttt{String} \\
ll \lessdot \texttt{List<X} : [\bot..\texttt{Object}].\texttt{List<X>>}, \\
u \doteq \texttt{Object}, \\
l \doteq \bot
\end{array}
$$

This result is equivalent to the following typisation of the input method:

```
void m(List<List<? extends Object>> l,
       List<Integer> la, List<String> lb){
  la.add(1);
  lb.add("String");
  l.add(la);
  l.add(lb);
}
```

6 Conclusion and Further Work

The problems introduced in the openening 1.2 can be solved via our **Unify** algorithm (see examples 5 and 5). As you can see by the given examples our type inference algorithm can calculate type solutions for programs involving wildcards.

This is currently a draft paper. We plan to extend it with a constraint generation algorithm aswell as a well defined way to insert type solutions into the typeless input program. Hereby we plan to use Featherweight Java as a starting point and to proof soundness and completeness.

References

1. Bierhoff, K.: Wildcards need witness protection. Proc. ACM Program. Lang. **6**(OOPSLA2) (oct 2022). https://doi.org/10.1145/3563301, https://doi.org/10.1145/3563301
2. Cameron, N., Drossopoulou, S., Ernst, E.: A model for java with wildcards. In: Vitek, J. (ed.) ECOOP 2008 – Object-Oriented Programming. pp. 2–26. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
3. Plümicke, M.: Java type unification with wildcards. In: Seipel, D., Hanus, M., Wolf, A. (eds.) 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers. Lecture Notes in Artificial Intelligence, vol. 5437, pp. 223–240. Springer-Verlag Heidelberg (2009)
4. Stadelmeier, A., Plümicke, M., Thiemann, P.: Global Type Inference for Featherweight Generic Java. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming (ECOOP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 222, pp. 28:1–28:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.28, https://drops.dagstuhl.de/opus/volltexte/2022/16256
5. Steurer, F., Plümicke, M.: Erweiterung und Neuimplementierung der Java Typunifikation. In: Knoop, J., Steffen, M., y Widemann, B.T. (eds.) Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts. pp. 134–149. No. 482 in Research Report, Faculty of Mathematics and Natural Sciences, UNIVERSITY OF OSLO (2018), iSBN 978-82-7368-447-9, (in german)

LLJava(-Live): Brew As You Go

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

[1] Nordakademie, Elmshorn, DE
[2] semantics gGmbH, Berlin, DE
baltasar@trancon.de
http://bandm.eu/

Abstract. LLJava is a low-level programming language that runs on the Java Virtual Machine. It represents the capabilities of the machine faithfully, but abstracts from tedious technical details of its bytecode. LLJava comes in two forms: as a stand-alone language with textual representation and a full-fledged compiler [1] (implemented in Java), and as LLJava-Live, a Java builder API for the intermediate representation [3]. LLJava-Live supports bytecode generation, class loading, and object linking at run-time. It works well above the abstraction level of popular JVM libraries such as ASM, and supports sophisticated dynamic meta-programming. We review the key design features of both LLJava and LLJava-Live, and discuss some complete [2] and work-in-progress [4] applications.

References

[1] B. Trancón y Widemann and M. Lepper. "LLJava: Minimalist Structured Programming on the Java Virtual Machine". In: *Proc. Principles and Practices of Programming on the Java Platform (PPPJ 2016)*. ACM, 2016. ISBN: 978-1-4503-4135-6. DOI: 10.1145/2972206.2972218.

[2] B. Trancón y Widemann and M. Lepper. "Improving the Performance of the Paisley Pattern-Matching EDSL by Staged Combinatorial Compilation". In: *Declarative Programming and Knowledge Management*. Vol. 12057. LNAI. Cham, CH: Springer, 2019. ISBN: 978-3-030-46713-5. URL: https://doi.org/10.1007/978-3-030-46714-2.

[3] B. Trancón y Widemann and M. Lepper. "LLJava Live at the Loop – A Case for Heteroiconic Staged Meta-Programming". In: *MPLR — Managed Prorgramming Languages 2021*. 2021, pp. 113–126. DOI: 10.1145/3475738.3480942.

[4] B. Trancón y Widemann and M. Lepper. "Sig-adLib: A Compilable Embedded Language for Synchronous Data-Flow Programming on the Java Virtual Machine". In: *Draft Proc. TFP 2022*. Ed. by N. Wu and W. Swierstra. 2022, pp. 84–105. URL: https://trendsfp.github.io/2022/tfp-abstracts.pdf.

(Almost) Agile Development of Verified Compilers

Wolf Zimmermann[1], Thomas Kühn[1], and Mandy Weißbach[1]

Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg, 06120 Halle(Saale), Germany wolf.zimmermann@informatik.uni-halle.de

1 Introduction

In practice, compilers are developed iterativly [1]. First compilers for sublanguages starting from an initial version are developed and successively extended until the whole programming language is implemented. Then optimizations are been added. This paper introduces the idea to make this approach agile, i.e., starting from a compiler the most simple program of a programming language and extending it step by step for each language concept:

$$\text{Initial Version} + \Delta_{0.1} + \cdots + \Delta_{0.n}$$

where $\Delta_{0.i}$ is a language concept (including built-in types, built-in procedures, and standard libraries). Furthermore, we want to examine whether this construction principle can also be applied (using the same iterations as in the construction) for the definition of the semantics and the verification of the compiler.

Our hypothesis is that this iterative construction possible without changing previous implementations, semantics definitions, and verification proofs provided that an adequate iteration plan is developed prior to starting the agile development process.

We plan to demonstrate this approach by the example of a compiler from Sather-K to the MIPS assembly language. Sather-K [6] is chosen because it is an object-oriented language that contains a sufficiently number of different language concepts (including built-in classes and methods) such as (among others):

- Parameterized Classes with static call-by-name semantics
- Notion of used classes
- Streams/Iterators
- Polymorphic and monomorphic classes
- Builtin Functions and Builtin classes
- Value Classes and Reference Classes
- Distinction subtyping and inheritance
- Methods and Streams as (assignable) values
- Partial Evaluation

and it is a language that is far less complex (the language report [5] has only 35 pages) than other languages, e.g. Java, C++, or C#. Although Sather-K is not a toy language, it seems to be manageable and feasable to build a verified compiler. The MIPS assembly target language is chosen since it is well-documented and there are emulators such as Spim [17] and the related tool QtSpim which [18] supports debugging of MIPS assembly code.

We plan to examine and answer the following research questions:

- Is a small step development of a compiler possible that avoids refactorings?
- What are the requirements and tools that support this development?
- Is it possible to define static and dynamic semantics along the same iterations as the compiler develops without refactoring of previous versions?
- Is it possible to verify the compiler along the same iterations as it develops without revising previous proofs?
- What are the requirements for tool supporting the iterative verification process?

The paper is organized as follows: Section 2 discusses iterative compiler development and Section 3 compiler verification approaches. Each of these sections starts with an overview of the state-of-the-art and proposes then the approaches to be used for agile development. Section 4 concludes the paper. In particular, it discusses first experiences on building the initial version in the iterative process.

2 Iterative Compiler Development

The process of iterative compiler development based on a hierarchy of sub-languages has been prososed by Basili and Turner [1]. We were not able to find other works on iterative compiler development - in particular no work on agile compiler development.

We practice the iterative development approach in our compiler lab where students implement the first 30 steps in developing a compiler for the language LAX (Appendix A in [25]). The initial version is pretty heavy, but the subsequent versions are developed in general rapidly. Some iteration cycles take only a few hours. Therefore the approach is almost agile. The commonalities to agile approaches is that each sprint is adding a new language construct (there is only one version that solely refactors the attribute grammar). The compiler is kept running - even within sprints. Hence, continous integration is practiced with large test sets (in higher versions several millions generated test cases).

Remark 1. In the Sather-K project, the large test set might be replaced by a smaller test set and extended by the verification of each version.

However, there are also differences to agile approaches. The first and probably most important one is that the initial version requires a lot of work because this is important for the subsequent versions. Basili and Turner mentioned already the importance of the initial version [1] because great attention is required for

major decisions in the compiler implementation. A wrong decision may lead in a subsequent version to a complete refactoring. We made the same experience on the effort of the initial version in the compiler lab as well as in the construction of the initial version of the Sather K project. However each decision in any version must be carefully examined to avoid points-of-no-returns that may lead to refactorings in subsequent versions.

This is the second major difference to agile approaches: if a sprint in an agile development is finished then the next sprint is being planned. This plan can be to refactor the code in order to eleminate defects and bugs. In compiler development this might become painful and lead to partial re-design of the current compiler version. Thus, for agile compiler development, it is essential to respect the complete source language (and possibly target language) in order to avoid refactorings.

Remark 2. We believe that comparable practices might be useful in agile approaches.

Since the complexitiy of the added language concepts are different, the sprint length may vary. For example, adding an integer addition is much less effort than adding for if-then-else expressions and statements since the latter requires the implementation of basic block graphs (provided that this is the first version with branching control).

This almost agile approach requires tool support. Compiler tools have a sophisticated foundational background. We would like to demonstrate that the following concepts used by compiler generators support the almost agile compiler development approach

- LALR(1)-grammars [16]. There is evidence as LALR(1)-grammars support modularity in language design and implementation [3] although there are other papers that state the opposite [14]
- Ordered attribute grammars [12]: for a new language concept, attributes and attribution rules need to be added. This might require to add also attribution rules to attribute grammar production of previous sprints. [13] show that ordered attribute grammars support modularity.
- Configurable Definition Tables: specification of semantic properties are just being added to a definition table specification [23]
- Tree transformations for generating intermediate representation [9]: new transformation rules for new language concepts need to be added.
- Bottom-Up Rewrite Systems[22]: For a new instruction in the intermediate code, a tree grammar rule needs to be added.
- Abstract interpretations for register assignments [26]: In general these are local abstract interpretations, i.e., on the basic block level. If a version introduces a new target tree instruction, then only a new abstract state transition needs to be introduced. Register assignment can be implemented only on the base of the current abstract state.
- Fixed point computations [15]. If a new intermediate language instruction is being introduced, it is only necessary to add the transfer function for this instruction.

Remark 3. There is no formal justification for the extensibiliy of LALR(1)-grammars and ordered attribute grammars. However, in the compiler lab (where students develop about 25-30 versions), we never observed shift-reduce or reduce-reduce conflicts over a period of 10 years.

For ordered attribute grammars, it happened that extensions lead to a non-ordered attribute grammar. However, in most cases the attribute grammar of the extended version was absolutely acyclic. This means that adding an artificial dependency lead to an ordered attribute grammar. Furthermore, if the extension lead to cyclic induced dependencies, this was a consequence of some mis-interpretations in previous versions.

We use the Eli-Compiler Toolset [8] since it supports all of the above requirements except the tree transformation (this is implemented by C-functions that are weaved into the compiler).

3 Verified Compilers

The field of compiler verification was already successful, but we are not aware of any work that systematically investigates the construction process and adopts the iterative method for compiler construction. Each of the following works uses operational semantics and base their notion of correctness on preservation of observable behaviour (with some variants).

The ProCos approach [11] uses proven algebraic identies that are applied in the compilation. They apply their approach to a small language based on Dijkstra's guarded commands. Their approach is based on the assumption that language concepts are orthogonal. However, language concepts are only orthogonal to some extent. Concepts such as e.g. exception handling, break- and continue statements, explicit return statements influence the semantics of loops, procedure calls.

In *Verifix* [7] approaches towards compilers for realistic compilers are considered. The focus was on the construction approaches towards compilers. Some proofs are checked using PVS [4]. *Verifix* allows that the target code may abort due to lack of resources (such as e.g. exceeding available memory). A complete back-end producing binary code for the DEC-Alpha processor family has been proven[27]. Approaches for correct parsing has been developed [10]. Otherwise, the correctness is based on an abstract syntax including correct static semantic properties. A fully verified compiler for a realistic language has not been developed.

Schmid, Stärk, and Börger proved the correctness of a compiler from Java to the Java Virtual Machine [24]. Their approach defines a stack of five sub-languages. In a certain sense, their approach is iterative with coarse-grained iteration cycles. The approach doesn't include machine checked proofs. The semantics is based on abstract syntax including correct static semantic properties based on the Java language definition. The target code is Java Byte code, but not the binary. In particular, compilation to machine code of a target processor is not considered.

CompCert [19] is a fully verified compiler for a realistic sub-language of C that produces PowerPC assembly code. This sub-language has been extended [2]. Furthermore, the compiler includes now optimizations [21, 20]. The correctness is proven completely with Coq, i.e., the correctness proof is fully checked by machine. It is use a similar notion of correctness as *Verifix* since it requires that safe programs are being compiled. In particular the notion of safe programs excludes undefined behaviour and also violation of resource constraints. The target code is the symbolic assembly code of the PowerPC. The proofs start with abstract syntax and static semantic properties. The correctness of parsing towards attributed abstract syntax trees is not considered.

As mentioned above, none of these works consider the process of construction of verified compilers using *lightweight iteration cycles*.

Furthermore, for verified compilers the computation of *static semantic properties* must be verified. For example, the instantiation of parameters of parameterized classes is a static property in Sather-K and in many other languages having generics (e.g. C++, Eiffel, C#, Ada, Haskell, ML). The correct instantiation of generic parameters is essential for compiler correctness. For this purpose, we consider to apply the ProCos approach. However, there are many more static semantic properties that need to be correctly computed in a compiler. Another issue is that all of the above approaches may allow that a compiler produces code from source text that doesn't belong to the source language. This is certainly unexpected. Instead, a verified compiler should reject texts that doesn't satisfy static semantic properties defined by the source language definition.

None of the above approaches consider *built-in* procedures, built-in classes, or built-in types. These built-in features interact with the dynamic semantics but are usually not part of it. There are several possibilities to deal with built-in properties. For types and classes, the value must be representend in the machine and the correctness of the functions/procedures provided by the type must be proven. For the latter, many functions are built-in. We currently see the following alternatives: First, the built-in function is implemented using assembly code of the target machine, the correctness of the implementation is proven, and the correctnes of the function call must be proven w.r.t. the assembly code implementation. Second, built-in functions are implemented using intermediate code and each call inlines the intermediate code. Both, the correctness of the implementation and inlining must be proven.

For the agile verified compiler development, a semantics definition formalism that suppors fine-grained iteration as well would be helpful. The approaches by Stärk, Schmid and Börger [24], Verifix [7], and CompCert [19] all use operational semantics. The Abstract State Machine approach used in [24, 7] demonstrate that this base of semantic definition supports iterative construction of operational semantics that doesn't require changes in preceeding versions.

A tool support for checking the correctness proof is helpful for the agile development of verified compilers. In particular, these can be used for a *regression verification*, analogously regression testing, that ensures that proofs of the previous version really remain valid.

Thus, the following challenges must be solved for an almost agile development of verified compilers:

- Verification of static semantic properties
- Verification of built-in classes, built-in types, and built-in procedures/functions
- fine-grained successive language semantics definition extensions without refactoring the language semantics
- Verification of the extensions made in the sprint
- Automatic regression verification.

4 Conclusions

Currently, we have developed the kernel version and planned over 200 iteration cycles. Each iteration corresponds to a language concept, a builtin function, a builtin class, a class of the standard library, or a method of the standard library. It must be carefully examined which iteration influences other iterations in order to avoid refactorings.

```
class MAIN is
  main is
  end -- main
end -- MAIN
```

Fig. 1. Sather-K Program for the Initial Version

The initial version is a main class only containing a main method with an empty body, cf. Fig. 1. The main class must be provided when calling the compiler, otherwise the file name converted to upper case letters is chosen as the main class. The semantic is a static method call `MAIN::main`. The implementation of this initial version requires already:

- the implementation of runtime stack into the memory of the target machine,
- the implementation of (parameterless) procedure calls,
- successive static transformations according to the language definition,
- building basic block graphs for procedures where instructions in a basic block are intermediate language trees (note that administrative information for procedure frame such as dynamic predecessors, return address etc. need to be managed),
- code selection by bottom-up rewrite systems and
- local abstract interpretation for register assignment.

Furthermore, the compiler for the initial version has already the same phases as the final compiler (although these are not required for the initial version). According to the language definition there are the following phases:

- Full abstract syntax tree of the program

- Establish an abstract syntax tree that only contains classes being used starting from the main class (according to the algorithm in the language definition)
- Establish an abstract syntax tree that expands all instantiations of parameterized classes
- Establish an abstract syntax tree that resolves all inheritances
- Generate a basic block graph for the target tree
- Generate MIPS code

Each step analyzes the static semantics that is required for the next step.

Hence, developing the initial kernel version is not lightweight. Important decisions for the subsequent versions are already done and cannot be altered. These decisions must be made on the complete language definition such that no points-of-no-returns are introduced.

The next step in completing the initial version is to define a dynamic semantics and to verify the compiler. The dynamic semantics (and therefore the verification of the compiler) is based on the abstract syntax in the fourth phase. After this step, all other 200 planned iteration cycles must be developed according the proposed phases.

References

1. Victor R Basil and Albert J Turner. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, SE-1(4):390–396, 1975.
2. Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54:135–163, 2015.
3. Walter Cazzola, Edoardo Vacchi, et al. Dexter and neverlang: a union towards dynamicity. *Proceedings of ICOOOLPS*, 12, 2012.
4. Axel Dold and Vincent Vialard. Formal verification of a compiler back-end generic checker program. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 470–480. Springer, 1999.
5. Gerhard Goos. Sather-k. Technical report, Universität Karlsruhe, Fakultät für Informatik, 1996. Revised Report with Heinz Schmidt.
6. Gerhard Goos. Sather-k – the language. *Software-Concepts and Tools*, 18(3):91–109, 1997.
7. Gerhard Goos and Wolf Zimmermann. Verification of compilers. *Correct System Design: Recent Insights and Advances*, page 201, 1999.
8. Robert W Gray, Steven P Levi, Vincent P Heuring, Anthony M Sloane, and William M Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.
9. Josef Grosch. Transformation of attributed trees using pattern matching. In *Compiler Construction: 4th International Conference, CC'92 Paderborn, FRG, October 5–7, 1992 Proceedings 4*, pages 1–15. Springer, 1992.
10. Andreas Heberle, Thilo Gaul, Wolfgang Goerigk, Gerhard Goos, and Wolf Zimmermann. Construction of verified compiler front-ends with program-checking. In *Perspectives of System Informatics: Third International Andrei Ershov Memorial Conference, PSI 99*, pages 481–492. Springer, 2000.

11. Charles Antony Richard Hoare, He Jifeng, and Augusto Sampaio. Normal form approach to compiler design. *Acta informatica*, 30:701–739, 1993.
12. Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.
13. Uwe Kastens and William M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
14. Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 918–932, 2010.
15. Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.
16. Wilf R LaLonde, ES Lee, and James J Horning. An lalr (k) parser generator. In *IFIP Congress (1)*, pages 513–518, 1971.
17. James Larus. *Assemblers, linkers, and the SPIM simulator*, chapter Appendix. Morgan Kaufmann, 2005.
18. James Larus. Qtspim, 2019.
19. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
20. David Monniaux and Cyril Six. Formally verified loop-invariant code motion and assorted optimizations. *ACM Transactions on Embedded Computing Systems*, 22(1):1–27, 2022.
21. Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for compcert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 448–461, 2016.
22. Eduardo Pelegri-Llopart and Susan L Graham. Optimal code generation for expression trees: an application burs theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 294–308, 1988.
23. Anthony M Sloane. Generating dynamic program analysis tools. In *Proceedings of Australian Software Engineering Conference ASWEC 97*, pages 166–173. IEEE, 1997.
24. Robert F Stärk, Joachim Schmid, and Egon Börger. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media, 2012.
25. William M Waite and Gerhard Goos. *Compiler construction*. Springer Science & Business Media, 2012.
26. Thomas R Wilcox. Generating machine code for high-level programming languages. Technical report, Cornell University, 1971.
27. Wolf Zimmermann and Thilo Gaul. On the construction of correct compiler backends: An asm-approach. *J. Univers. Comput. Sci.*, 3(5):504–567, 1997.