

Engineering

INSIGHTS

Schriftenreihe der Fakultät Technik: 2/2024

Tagungsband des Jahrestreffens 2024 der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“

Daniel Holle, Prof. Dr. Jens Knoop, Prof. Dr. habil. Martin Plümicke, Prof. Dr. Peter Thiemann,
Prof. Dr. habil. Baltasar Trancón y Widemann (Hrsg.)



Daniel Holle

Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
Studiengang Informatik
Florianstraße 15
72160 Horb am Neckar
E-Mail: d.holle@hb.dhbw-stuttgart.de



Prof. Dr. Jens Knoop

Technische Universität Wien
Fakultät für Informatik
Argentinierstr. 8
1040 Wien
Österreich
E-Mail: jens.knoop@tuwien.ac.at



Prof. Dr. habil. Martin Plümicke

Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
Studiengang Informatik
Florianstraße 15
72160 Horb am Neckar
E-Mail: m.pluemicke@hb.dhbw-stuttgart.de



Prof. Dr. Peter Thiemann

Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee Geb.079
79110 Freiburg i. Br.
E-Mail: thiemann@informatik.uni-freiburg.de



Prof. Dr. habil. Baltasar Trancón y Widemann

Technische Hochschule Brandenburg
Fachbereich Informatik und Medien
Praktische Informatik
Magdeburger Straße 50
14770 Brandenburg an der Havel
E-Mail: trancon@th-brandenburg.de



Die Teilnehmenden des Workshops

Vorwort

Seit 1984 veranstaltet die GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“ jedes Frühjahr einen Workshop im Physikzentrum Bad Honnef. Das Arbeitstreffen ist eine wichtige Plattform für Informatikerinnen und Informatiker zum Erfahrungsaustausch, der Diskussion, zum Netzwerken und zur Vertiefung von Kontakten. Vorgestellt werden Vorträge und Demonstrationen sowohl zu abgeschlossenen als auch laufenden Arbeiten, darunter Themen wie

- Sprachen, Sprachparadigmen
- Korrektheit von Entwurf und Implementierung
- Werkzeuge
- Software-/Hardware-Architekturen
- Spezifikation, Entwurf
- Validierung, Verifikation
- Implementierung, Integration
- Sicherheit (Safety und Security)
- eingebettete Systeme
- hardware-nahe Programmierung

In diesem Tagungsband wurden Beiträge des 40. Workshops aufgenommen, der vom 28. bis 30. April 2024 stattfand. Die Autorinnen und Autoren reichten Zusammenfassungen ihrer Beiträge genauso wie komplette Artikel ein. Auf dem Programm standen alle Kernthemen des Gebietes, unter anderem Semantik, Compiler, Domänen & Paradigmen, Sprachen & Engineering, Syntax & IDEs und Interpreter. Außerdem wurden Prototypen von Werkzeugen demonstriert.

Wir danken allen Teilnehmenden, die den Workshop mit ihren Vorträgen, Abstracts, Papers und lebendigen Diskussionen aufs Neue zu einem interessanten und aufschlussreichen Ereignis machten. Ein besonderer Dank gilt den Mitarbeitenden des Physikzentrums Bad Honnef, die auch am Sonntag durch ihre umfassende Betreuung für eine angenehme und anregende Atmosphäre gesorgt haben.

Daniel Holle, Jens Knoop, Martin Plümicke,
Peter Thiemann, Baltasar Trancón Widemann
September 2024

Inhaltsverzeichnis

<i>Alina Weber</i> – Compartmentalization through Diversification for Node.js	2
<i>Anton Ertl</i> – Interpreter vs. Compiler Performance at Run-Time	7
<i>Björn Lötters</i> – Generalizing Context-Free Subphrase Grammars	13
<i>Janis Voigtländer</i> – Parametrisierung von Haskell-Programmieraufgaben	15
<i>Jens Knoop</i> – Artificiosa intelligentia ad portas!	17
<i>Kai-Oliver Prott</i> – Compileroptimierung von Curry-Code mit Haskell als Zielsprache	19
<i>Marcellus Siegburg</i> – A Report on Automatic Generation of Petri Net Exercise and Exam Task Instances	21
<i>Baltasar Trancón y Widemann, Markus Lepper</i> – Morton Feldman's "Projections One to Five" — Exploring a Classical Avant-Garde Notation by Mathematical Remodelling	23
<i>Markus Raab, Yvonne Markl</i> – Precise to Universal Configurations: Elektra to PermaplanT	25
<i>Michael Hanus</i> – Inference and Verification of Arithmetic Non-Fail Conditions in Declarative Programs	29
<i>Niels Bunkenburg</i> – Anwendung einer adapterbasierten Effektmodellierung zur Implementierung eines Interpreters	31
<i>Julian Schmidt, Martin Plümicke</i> – Java-TX Compiler in Java-TX	33
<i>Andreas Stadelmeier, Martin Plümicke</i> – Global Type Inference for Featherweight Java with Wild-cards	53
<i>Peter Thiemann, Marius Weidner</i> – Towards Tagless Interpretation of Stratified System F	81
<i>Daniel Holle</i> – Pattern Matching in Java-TX	87
<i>Martin Plümicke</i> – Featherweight-Java-TX: A Minimal Core Calculus for Java-TX	93

Compartmentalization through Diversification for Node.js

Alina Weber-Hohengrund, Stefan Brunthaler
μCSRL - Munich Computer Systems Research Lab
Research Institute CODE
University of Bundeswehr Munich
Neubiberg, Germany
alina.weber-hohengrund@unibw.de

Abstract

As evidenced by multiple attacks, such as typo-squatting, protestware, and cryptojacking, JavaScript is both an easy and attractive target. Although each attacks address individual shortcomings, a common denominator among all attacks is a lack of *compartmentalization*. Further, the Node package manager (NPM) enables the reuse of huge amounts of code and projects, but without security guarantees of the loaded modules, leaving backend applications vulnerable to various supply-chain attacks. Furthermore, the missing compartmentalization leads to nearly immediate compromise after initial access. This weakness led to multiple approaches to sandbox JavaScript browser applications in order to protect the host system. The same observations hold equally well for JavaScript on servers, as evidenced by the same attacks being reported for the node.js ecosystem. In this paper we will describe a simple and lightweight compartmentalization to JavaScript for Node.js modules. It uses diversification to break the API between modules and with this the monoculture of the namespace for only non-legitimate inter-module calls. The approach is designed without using annotations by the developer while targeting minimum to zero runtime overhead for the user.

1 Introduction

JavaScript was developed in ten days [8]. At that time access modifiers such as `private` and `public` were omitted in the language design, which results in all properties of any object or function being public by nature. The use of closures enables support for private properties, also known as the Crockford Privacy Pattern [8]. Modules foster code reuse. They are loaded dynamically and again do not support any form of access modifiers, therefore at all points of the code any module code can be loaded and executed. Through the missing *compartmentalization* in JavaScript it is quite easy for attackers to expand to further code, making it a weakness of the language. Additionally, JavaScript is a reflective language [10] where

modules can be dynamically loaded and executed code can be changed at runtime.

In the beginning, JavaScript was mainly used for browser applications but because of its popularity it also gained backend support which is known as Node.js [7]. Because of the commonly known weaknesses of JavaScript, browsers such as Chromium by default sandbox renderers where JavaScript code runs [9]. Hence, the executed code can not interfere with the host system. There have been attempts to sandbox JavaScript processes for Node.js as well, but they always come with a performance impact, which is normally undesirable for backend applications [1, 6, 11]. At the beginning, Node.js only supported CommonJS modules, which differ in their specification from the JavaScript standard, ECMAScript

[7]. By now both, ECMAScript modules and CommonJS modules, are supported and both can be managed via the *Node Package Manager* (NPM). Through this package manager arbitrary packages can be downloaded and installed. Once installed, they can be loaded by any Node.js application. The packages are not checked by any trusted entity before made public, therefore execution of modules from NPM are basically executions of third-party code that was neither authenticated nor checked for trustworthiness. This opens possibilities for supply-chain attacks such as Typosquatting, Protestware, and Cryptojacking [4, 5, 12]. In listing 1, a simple code snippet is shown, which serves some ASCII art but on a specific request (line 5) different code is executed:

```

1 exports.asciiArt = function (figure) {
2   var fs = require("./fs-wrapper.js");
3   var penguin = fs.readFileSync(
4     "./penguinAsciiArt.txt", "utf8");
5   if (figure == 'attack'){
6     ...
7   }
8   else if (figure != 'penguin') {
9     return "Sorry we are currently
10      only serving penguins\n" +
11      penguin;
12 }

```

Listing 1: This code snippet serves some ASCII art depending on the requested figure. If figure equals 'attack' some unknown code is executed.

Such triggers are called *backdoors* and inserted by attackers to access a system at an arbitrary point in the future. As many packages are widely used and often refer to other packages, the amount of imported third-party code grows exponentially, thereby making it infeasible to check all this code on potentially malicious code. CVE-2022-23812 is a good example for the impact of this problem [3]. The developers of the commonly used package `node-ipc` added protestware to the module. If the host machine is identified to have an IP originating from Russia or Belarus by the additional code, the whole filesystem of the host is overwritten.

In JavaScript all functions and objects of modules are accessed using identifiers on the closure, which the module is wrapped in. These accesses are implemented through dynamic lookups on the available identifiers of the referenced object without any static verification. As a result, the

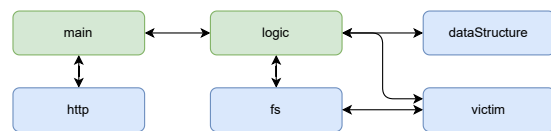


Figure 1: Show legitimate communications (black) between own modules (green) and third-party modules (blue).

accesses must stick to the available identifiers or else are returned `undefined`. In a sense, inter-module calls can be seen as API calls between different modules. In this work, we show a language-based approach to support *compartmentalization* in JavaScript. The approach purposefully breaks the API for inter-module calls in a controlled manner and uses two different modes: tracking and protecting. The design is meant to assist the developers while having minimal runtime overhead for the user. The language-based characteristic is chosen to treat the problem at the core through compartmentalization instead of building an additional environment around it. This also opens the possibility to run the final code in a sandbox to further enhance security.

2 Design

As already introduced, we intend to enable compartmentalization for JavaScript, without targeting the general language support for such a feature. Instead, our idea hardens the borders between different modules to restrict non-legitimate interactions. This raises the question of which module calls are legitimate and which are non-legitimate. As we do not require developers to annotate their code in any way but want the approach to work out-of-the-box, we have to decide this via different means. This is decided through the *Tracking Mode* (sec. 2.1). Later we use the information gained during tracking to recompile the source code to rename identifiers (sec. 2.2). Last but not least, the application is run in the *Protecting Mode* (sec. 2.3) where the final source code is loaded into the application with potential reactive code to handle irregular behavior.

2.1 Tracking Mode

In order to differentiate legitimate from non-legitimate accesses a ground truth is required. This is the task of the tracking mode.

Figure 1 shows connections (black) between different modules in an abstract application. Green modules are written by the same author as the main application and blue modules are included third-party code. In the tracking mode the application is run normally on a test suite but during the execution all or most interactions between modules are tracked and logged. This log then is processed to extract a map including all used modules and showing what other modules are accessed by them on which identifiers. All logged calls are considered legitimate and form the ground truth. This way of constructing the ground truth is important, because it means that initial compromise is not part of the threat model. The extracted information can also be visualized and then inspected by developers. This data can give interesting insights into dependencies and can also uncover unexpected and malicious accesses. Furthermore, the extracted map is also comparable between different versions of the application or just its dependencies to again uncover anomalies. In this sense, the extracted mapping can be used in a versatile manner to aid developers in securing their application against supply chain attacks. Call graphs are a common tool to find vulnerabilities [2].

2.2 Renaming

Next follows the renaming of relevant identifiers. In both CommonJS and ECMAScript, modules are normal JavaScript code and the exported values (e.g. objects, functions, constants) are assigned to an `export` object. The modules are then wrapped in a closure, with all identifiers assigned to `export` being accessible as property of the functional object of the closure. Closures are special functions that can access the local variables of the environment where they were created. Functions are also objects in JavaScript and therefore can be assigned properties analogous to normal objects.

As a result, all accesses from one module to another module need to access the closure or object via the defined names for the identifiers. Our approach hides accessed objects for non-legitimate calls. To this end, we rename all valid identifiers to random versions. The original identifiers are kept and now point to reactive code. All legitimate inter-module accesses are also adjusted to use the new identifiers by using the extracted mapping from the tracking mode. The consistent renaming will be implemented in the Google-Closure com-

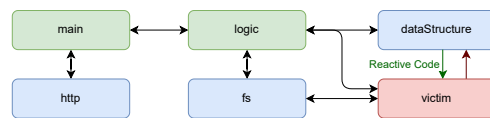


Figure 2: Same as before but with an non-legitimate connection (red) between the corrupted victim and `dataStructure`.

piler. The consistency ensures that all previously seen interactions are executed without any performance impact in the following protecting mode.

2.3 Protecting Mode

Figure 2 shows the same application as figure 1, but this time the victim was corrupted. The now malicious module (red) accesses the `dataStructure`. Because the original identifiers of this module were renamed and the original identifiers now point to reactive code, the irregular access can be handled by some user defined policy. This modularity is crucial to be able to deal with false positives. The policy can for example support some whitelisting and logging for different module interactions or also terminate the program in case of highly suspicious accesses. As test suits are rarely complete, the defense needs to be able to dynamically handle and differentiate malicious and legitimate accesses. Dynamic decisions during runtime normally result in a high performance impact, because of the interception of the executed code. The consistent renaming ensures that this reactive code is only triggered for suspicious cases, thereby keeping the performance impact of the protecting mode for a reasonably tested and tracked application low.

3 Discussion

An interesting question for the effectiveness of a defense is of course how well it works in case the attacker knows about the defense. As a positive side effect in our approach, all accesses will become visible either in the tracking mode or later in the protecting mode. In the first case they are visible in the mapping, hence renamed consistently. In the second case, they are not renamed consistently and therefore trigger the reactive code later on. The main issue will be to harden the approach against side-channel attacks for retrieving the new identifier names and bypassing the defense. In general, the core idea of the approach is to deny

access or trap malicious code right at the root by hindering it from interacting with anything else. The problem is that JavaScript is a reflective programming language where nearly everything can be changed and inspected at runtime. To name some examples: functions can be reassigned new code and a list of available identifiers on an object can be retrieved anytime during runtime. Assuming arbitrary code execution, this enables the attacker to simply traverse all possible identifiers until the identifier pointing to the expected source code is found. To close this attack vector it will be necessary to restrict this reflective nature of JavaScript which could reveal too much information about the internal state of the program. At the same time, the defense is designed to have nearly no performance impact in protecting mode which is a crucial factor for defenses to be actually used in production for real-world applications.

4 Conclusion

In this paper we proposed and sketched a language-based approach for enhancing compartmentalization of modules for Node.js. The idea is designed to assist during development with the tracking mode and secure the application in protecting mode. During protection, the identifiers are renamed to random names and the original names point to reactive code enabling dynamic handling of previously unseen calls at runtime depending on a user defined policy. The consistent renaming keeps the performance impact to a minimum contrary to sandboxing approaches. Currently a proof-of-concept is developed to test the idea against the various challenges of JavaScript itself, but also real-world conditions.

References

- [1] Marco Abbadini et al. "Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses". In: *ASIA CCS '23: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (2023).
- [2] Gábor Antal et al. "Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools". In: *IEEE Access* (2023).
- [3] *CVE-2022-23812*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23812>.
- [4] Alexandre Decan et al. "On the impact of security vulnerabilities in the npm package dependency network". In: *MSR '18: Proceedings of the 15th International Conference on Mining Software Repositories* (2018).
- [5] Chengwei Liu et al. "Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem". In: *ICSE '22: Proceedings of the 44th International Conference on Software Engineering* (2022).
- [6] Taemin Park et al. "NoJITsu: Locking Down JavaScript Engines". In: *Network and Distributed System Security Symposium* (2020).
- [7] Shelley Powers. *Learning Node 2e: Moving to the Server-Side*. 2016.
- [8] Axel Rauschmayer. *Speaking JavaScript: An In-Depth Guide for Programmers*. 2014.
- [9] *Sandbox*. URL: <https://chromium.googlesource.com/chromium/src/+HEAD/docs/design/sandbox.md>.
- [10] David Ungar et al. "Self: The power of simplicity". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications* (1987).
- [11] Nikos Vasilakis et al. "BreakApp: Automated, Flexible Application Compartmentalization." In: *Network and Distributed Systems Security (NDSS) Symposium* (2018).
- [12] Markus Zimmermann et al. "Small World with High Risks: A Study of Security Threats in the npm Ecosystem". In: *SEC'19: Proceedings of the 28th USENIX Conference on Security Symposium* (2019).

Interpreter vs. Compiler Performance at Run-Time

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

Abstract

Both “interpreter” and “compiler” cover a wide range of implementation techniques, and in some cases one can argue where to draw the line. Consequently, both approaches also have a wide performance spread, leading to comparable performance in some cases. The present work looks closely at the SwiftForth JIT compiler and the Gforth interpreter. Even with dynamic superinstructions and IP-update optimizations, Gforth suffers from interpretive overhead, but that is compensated by Gforth’s additional sophistication in caching stack items in registers, an optimization that SwiftForth does not employ.

1 Introduction

The following properties are desired for programming language implementations:

- Execution speed
- Compilation speed
- Cheap development and maintenance (of the programming language implementation)
- Portability/retargetability

Programming language implementation techniques have been traditionally been classified as interpreters or compilers.¹ Both classes encompass a wide range of implementation techniques.

The conventional wisdom is that compilers are not just faster than interpreters, but that the two classes are so far apart that the claim

¹ There is also metacompilation, where an interpreter is partially evaluated [9]; this approach has become practical in recent years thanks to huge efforts by developers of systems like GraalVM/Truffle [12]. However, in the present work we focus on more traditional approaches, and leave metacompilation to further work.

There is a bigger performance gap between a performance oblivious interpreter and a high-performance interpreter than between a high-performance interpreter and a JIT.

in a draft version of a paper led a reviewer to ask for some support for this claim. In a later draft this claim had morphed into its final version [7]:

While [interpreters] cannot compete in execution performance with JIT compilers or ahead-of-time compilers, a fast interpreter is not that far away: e.g., with the IP update optimizations of the present work, Gforth has similar performance to the SwiftForth JIT compiler and to gcc -O0.

which led one reviewer to speculate “It might be that SwiftForth [...] is simply a very weak compiler.” and the statement “None of these can let us conclude that, in general, interpreters are not lagging far behind compilers.” Another reviewer was very impressed by the results (also shown in Section 2).

These reactions show a widespread belief that interpreter performance and compiler run-time performance are islands separated by a vast ocean.

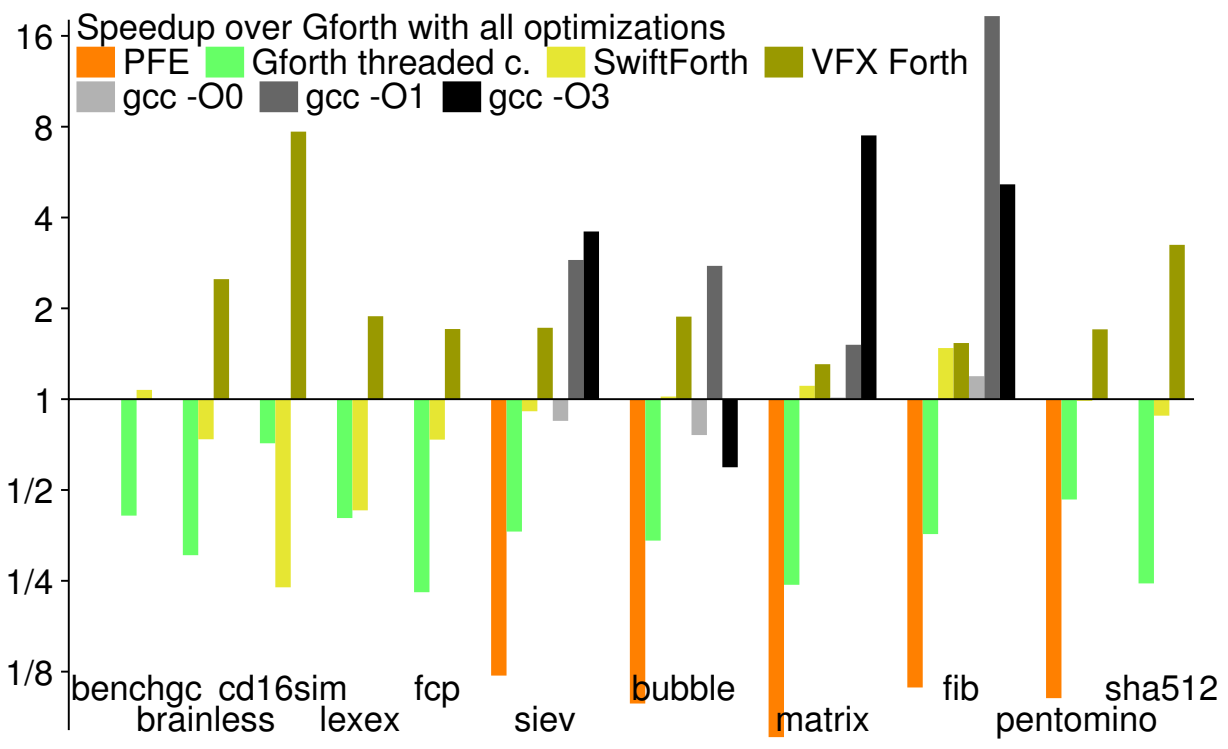


Figure 1: Speedup (above the baseline) or slowdown (below the baseline) of several Forth systems and gcc over the fastest Gforth version, on Tiger Lake (Core i5-1135G7). If a benchmark does not work on a system, no bar is shown for the combination.

I have written the present paper to address this belief.

First I look at some performance results (Section 2), then take a closer look at two of the involved systems (Section 3).

2 Performance

Figure 1 shows the same data as Figure 10 of our upcoming paper [7], but uses a different baseline system (in the speedup numbers the run time of the baseline is divided by the run time of the system specified for the bar): here the baseline is the fastest Gforth version, the one that includes the optimizations from our recent paper [7]. We consider it to be a virtual-machine (VM) interpreter, because it needs to access the representation of the VM code at run-time in order to access immediate operands and to perform control flow. However, as we will see there is a degree of machine-code generation involved.

The systems and compilers compared with the baseline are:

PFE is an interpreted Forth system written in C that uses one C function per VM instruction implementation. PFE is designed to rely on explicit register allocation (a GCC extension) for performance, but unfortunately, for AMD64 no explicit register definitions have been added yet. We use PFE-0.33.71.

Gforth threaded c. This is the baseline Gforth with the option `-no-dynamic`, which means that it falls back to using direct-threaded code [1]; this option also disables stack caching. This is similar in implementation and performance to Gforth around the year 2000.

SwiftForth, VFX Forth Two commercial Forth systems with JIT compilers. We measured SwiftForth x64-Linux 4.0.0-RC87 and VFX Forth 64 5.43.

gcc-12 Various optimization options for GCC 12.2. Manually written C code for four of the benchmarks is available and was used for generating these results. For gcc the results do not include the compile time (unlike for the Forth systems).

Coming back to the claims that prompted this paper, while PFE is not performance-oblivious,

the baseline Gforth has a higher speedup over it for the benchmarks where PFE works than VFX Forth (the fastest JIT compiler we measured) has over the baseline.

VFX Forth performs extensive inlining and allocates data stack items to registers within basic blocks. This leads to a speedup over Gforth by about a factor of 2 in most benchmarks. Idiomatic Forth code consists of short definitions, resulting in many calls, so inlining is particularly effective. *cd16sim* contains a large number of (implicit) calls to an empty definition, making the inlining of VFX particularly effective here.

SwiftForth, another JIT compiler, is typically in the same ballpark as the fastest Gforth; for *bubble* and *pentomino* the performance is so similar that the SwiftForth bar is barely visible. We will discuss SwiftForth in depth in Section 3.

`gcc -O0` produces performance similar to the baseline. `gcc -O1` shows a good speedup. `gcc -O3` is better on some benchmarks, but worse on others. We have looked at the two slowdown cases. For *bubble*, `gcc -O3` auto-vectorizes, and the result is that there is partial overlap between a store and a following load, which results in the hardware taking a slow path rather than performing one of its store-to-load forwarding optimizations. For *fib*, we have not found the reason for the slowdown.

3 A tale of two Forth systems

This section provides a closer look at SwiftForth and Gforth and the tradeoffs in their creation, and how this affects performance.

3.1 SwiftForth

SwiftForth is a commercial Forth system from Forth, Inc. Forth, Inc. (the first commercial Forth vendor) developed a number of interpreters using indirect-threaded code [2] from the 1970s until the 1990s [11]. These systems were written in Forth with a foundation of (Forth) assembly language.

With the introduction of SwiftForth during the 1990s Forth, Inc. switched from threaded-code interpreters to native-code compilation (aka JIT compilation). The intention of this change was improved performance, and that also shows in SwiftForth's name. And it delivers: In most benchmarks, SwiftForth is 2–4 times faster than Gforth

threaded code (which probably has performance similar to the threaded-code polyForth II system that preceded SwiftForth). Fortunately, while SwiftForth's source code is proprietary, it is delivered with SwiftForth, which makes it easier to study the compilation techniques used.

The basic compiler of SwiftForth concatenates the native code of the primitive Forth words in a definition into the native code for that definition. The primitives are still written in assembly language, and are probably very similar to the code for the primitives in polyForth II, with one exception: In polyForth II each primitive ends with a threaded-code dispatch, while in SwiftForth it ends with a native-code `ret` instruction (which is not copied in the concatenation).

In addition, for control-flow primitives the native code is first copied and the target offsets are patched into the copied code; similarly, for literal numbers the number is patched into the copied code for the primitive (`literal`). This is the fundamental difference to Gforth's interpreter-based approach, which does not patch native code, but instead keeps the interpreted code around and accesses it when control flow or literal values are needed.

In addition, SwiftForth optimizes pairs of Forth primitives to better code with optimization rules like

```
OPTIMIZE DUP +          SUBSTITUTE 2*
```

The result of the substitution can be subject to another optimization rule, resulting in the optimization of longer sequences. There are 346 optimization rules used in the version of SwiftForth we measured; in many cases this requires to also define the substitution word and/or a word that performs additional compile-time work when an optimization rule triggers. These optimizations include tail-call optimization. In the version of SwiftForth we measured, files containing a total of 1819 lines contain most of the optimizations.

So while SwiftForth's compiler is not the most sophisticated one in existence (and simplicity is a major goal in SwiftForth development), it is far from being "very weak".

Forth native-code compilers have suffered from running into microarchitectural pitfalls due to legacy techniques for implementing certain features of Forth. In particular, the measured version of SwiftForth implements Forth's `does>` in a way

that puts native code close to written-to data, resulting in ping-ponging between the I-cache and the D-cache due to false sharing. It also pops the return address of a call instead of returning to it, resulting in branch mispredictions in subsequent returns. These pitfalls are responsible for the low performance of SwiftForth on at least *cd16sim*.

These pitfalls have been reported to Forth, Inc., which fixed them (for the common case) in SwiftForth-4.0.0-RC89, but that version appeared too late for producing new results and working them into this paper.

This episode demonstrates that on modern CPUs it's not enough to reduce the number of executed instructions, you also have to be aware of microarchitectural pitfalls and avoid them. The *bubble* result of `gcc -O3` demonstrates that even a project that puts massive manpower into optimization can run afoul of such problems.

One cost of SwiftForth's assembly-based approach is that each of the i386 and the x64 ports has about 7000 lines of architecture-specific files.

3.2 Gforth

Gforth is a non-commercial free software project and was developed mainly on Unix systems starting in 1992. Among the goals of Gforth was that it should be efficient and available on many machines. These goals were initially achieved by implementing Gforth mostly in Forth, with an indirect-threaded code interpreter, i.e., along traditional Forth implementation techniques, but using GNU C instead of assembly language for writing the interpreter foundation.

Given the wide range of general-purpose computer architectures of the 1990s, our portability goals made machine-specific code without a fallback to machine-independent code unattractive, so we did not switch to native-code compilers.

From 2001 to 2005 we implemented a number of optimizations and enabling changes: switching to primitive-centric threaded code [3], static superinstructions [8], dynamic superinstructions and stack caching [4, 6]. This frenzy was followed by a long period of consolidation and focussing on other topics, but eventually additional optimizations were added: generalized constant folding [10] and the IP-update optimizations [7]. These improvements were research-driven, i.e.,

they used Gforth as a research vehicle, but they then became production features.

A static superinstruction combines a sequence of primitives into a better primitive, like simple substitution rules for SwiftForth's optimizer. The version of Gforth that we have measured uses 55 static superinstructions; the reasons why we don't use more are: One of the benefits of static superinstructions is subsumed by stack caching, so we use static superinstructions only in cases where we expect an additional benefit; our implementation of static superinstructions does not work as well with stack caching as we would like; adding more static superinstructions increases the compile time of Gforth.

Dynamic superinstructions optimize a sequence of primitives by concatenating their native code, but without the threaded-code dispatch. The effect on straight-line code is similar to the effect of SwiftForth's basic compiler. One difference is that immediate operands of primitives (e.g., of literals) and the targets of control flow are not patched into the native code, but are still accessed through the virtual-machine instruction pointer (IP), which points to the same place as in the original threaded code. Performing the next primitive in sequence is achieved by falling through to the code of the next primitive, but all other control flow is performed through a threaded-code dispatch. A benefit of this approach is that one can fall back to plain threaded code for a single primitive (e.g., because it is not relocatable) by appending the threaded-code dispatch code to the dynamic superinstruction preceding it.

As implemented until 2023, every primitive still updated the IP. Since 2023, we have implemented a number of IP-update optimizations, which produce big speedups (up to a factor of 3) on loop-dominated benchmarks on modern hardware.

Forth is a stack-based language. Stack caching represents the stack in several different ways, each with a different number of stack items in registers. This allows to implement variants of primitives that perform fewer memory accesses and fewer stack pointer updates than the base variant of the primitive that uses the same representation on entry and on exit. Gforth uses up to three registers for stack items on AMD64. This optimization has a part of the effect of VFX's register allocation for data-stack items in a basic block.

Generalized constant folding allows to perform optimizations such as turning division by a constant

into a multiplication by the reciprocal, but in the interest of brevity I will not discuss it further here.

Performance-wise, compared to SwiftForth, Gforth suffers from having to perform control flow through threaded-code dispatch and having to access immediate operands in memory through IP, but it benefits from stack caching which SwiftForth does not have.

The main benefit of Gforth's approach is portability: E.g., Gforth ran out of the box when AMD64, ARM A64, and RISC-V became available to us (in 2003, 2016, and 2017, respectively), and with a little work (typically less than an hour) all performance features could be activated. By contrast, SwiftForth's AMD64 port only started in 2020 and it is still in Beta testing, and the only other port is the IA-32 port. However, SwiftX, the cross-compiler for embedded systems, is available for 11 cores from 8 base architectures.

Gforth's approach comes at a cost in complexity, though. A rough estimate based on the sizes of various files is that about 5000 lines of code are spent on static superinstructions, dynamic superinstructions, stack caching, and IP-update optimizations. The IP-update optimizations alone have inserted 864 lines and deleted 316 lines [7].

One development interesting in the present context is that at one point we worked on completely eliminating the IP from Gforth [5], turning it into what was later called a copy-and-patch compiler [13]. However, that approach appeared to be too brittle (no fallback option that would work under all circumstances), so we did not turn it into a production feature.

3.3 Example

To make things more concrete, here's a piece of Forth source code that demonstrates the differences between SwiftForth's and Gforth's code generation. Consider the Forth definition:

```
: squared dup * ;
```

This definition defines the word `squared` to perform the primitives `dup` and `*`; then the definition ends (and at run-time the execution returns). The value to be squared is passed on the data stack, and `dup` pushes another copy of that value on the data stack. `*` then pops these two copies, multiplies them, and pushes the result.

For this example neither SwiftForth nor Gforth have static superinstructions, so we can look at the native code for the individual primitives (or, with stack caching, primitive variants):

Src	SwiftForth	Gforth
<code>dup</code>	<code>lea -8(%rbp),%rbp mov %rbx,0(%rbp)</code>	<code>mov %r8,%r15</code>
<code>*</code>	<code>mov 0(%rbp),%rax mul %rbx mov %rax,%rbx lea 8(%rbp),%rbp</code>	<code>imul %r15,%r8</code>
<code>;</code>	<code>ret</code>	<code>mov (%r14),%rbx add \$8,%r14 mov (%rbx),%rax jmp *%rax</code>

The big picture is that stack caching leads to much shorter code for `dup` and `*` for Gforth, but Gforth performs more instructions for the return from `squared` to its caller; i.e., we see the interpretive overhead in this return.

For SwiftForth the details are: the data-stack pointer is in `%rbp`, and the `lea` instructions update the data-stack pointer. The top of the data stack is in `%rbx`. `mul` multiplies `%rax` with the argument, and puts the result in `%rax`.

For Gforth, with one data stack item in a register (representation 1), the top-of-stack is in `%r8`; with two data stack items in registers (representation 2), the second item is in `%r8` and the top item is in `%r15`. In this example, `dup` starts out in representation 1, and finishes in representation 2; `*` starts in representation 2 and it finishes in representation 1.

For the return, the return-stack² pointer is in `%r14`, and IP is in `%rbx`. So the return first loads the IP from the return stack, updates the return-stack pointer, and then performs a threaded-code dispatch.

4 Conclusion

The Gforth interpreter shows performance in the same ballpark as the SwiftForth compiler. When looking at the details, we see that Gforth, despite using several optimizations that reduce the interpretive overhead, still suffers from the remainder of this overhead; in particular, the overhead on control flow and when dealing with immediate operands. However, apparently stack caching

² Forth stores return addresses on a separate stack so that they are not in the way of accessing data on the data stack.

(implemented in Gforth, but not in SwiftForth) provides enough speedup to compensate for the interpretive overhead slowdown.

In addition, with either approach one should avoid falling prey to microarchitectural pitfalls.

References

- [1] James R. Bell. “Threaded Code”. In: 16.6 (1973), pp. 370–372.
- [2] Robert B.K. Dewar. “Indirect Threaded Code”. In: 18.6 (June 1975), pp. 330–331.
- [3] M. Anton Ertl. “Threaded Code Variations and Optimizations (Extended Version)”. In: *Forth-Tagung 2002*. Garmisch-Partenkirchen, 2002. URL: <https://www.complang.tuwien.ac.at/papers/ertl02.ps.gz>.
- [4] M. Anton Ertl and David Gregg. “Combining Stack Caching with Dynamic Superinstructions”. In: *Interpreters, Virtual Machines and Emulators (IVME '04)*. 2004, pp. 7–14. URL: <https://www.complang.tuwien.ac.at/papers/ertl%26gregg04ivme.ps.gz>.
- [5] M. Anton Ertl and David Gregg. “Retargeting JIT compilers by using C-compiler generated executable code”. In: *Parallel Architecture and Compilation Techniques (PACT'04)*. 2004, pp. 41–50. URL: <https://www.complang.tuwien.ac.at/papers/ertl%26gregg04pact.ps.gz>.
- [6] M. Anton Ertl and David Gregg. “Stack Caching in Forth”. In: *21st EuroForth Conference*. Ed. by M. Anton Ertl. 2005, pp. 6–15. URL: <https://www.complang.tuwien.ac.at/papers/ertl%26gregg05.ps.gz>.
- [7] M. Anton Ertl and Bernd Paysan. “The Performance Effects of Virtual-Machine Instruction Pointer Updates”. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 14:1–14:26. ISBN: 978-3-95977-341-6. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.14>.
- [8] M. Anton Ertl et al. “vmgen — A Generator of Efficient Virtual Machine Interpreters”. In: *Software—Practice and Experience* 32.3 (2002), pp. 265–294. URL: <https://www.complang.tuwien.ac.at/papers/ertl+02.ps.gz>.
- [9] Octave Larose et al. “AST vs. Bytecode: Interpreters in the Age of Meta-Compilation”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). URL: <https://doi.org/10.1145/3622808>.
- [10] Bernd Paysan. “Constant Folding für Gforth”. In: *Vierte Dimension* 35.2 (2019), p. 17. URL: <https://wiki.forth-ev.de/lib/exe/fetch.php/vd-archiv:4d2019-02.pdf>.
- [11] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. “The Evolution of Forth”. In: *History of Programming Languages (HOPL-II) Preprints*. SIGPLAN Notices 28(3). 1993, pp. 177–199.
- [12] Thomas Würthinger et al. “Practical partial evaluation for high-performance dynamic language runtimes”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 662–676. ISSN: 0362-1340. URL: <https://doi.org/10.1145/3140587.3062381>.
- [13] Haoran Xu and Fredrik Kjolstad. “Copy-and-Patch Compilation”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021), 136:1–136:30. URL: <https://fredrikbk.com/publications/copy-and-patch.pdf>.

Generalizing Context-Free Subphrase Grammars

Björn Lötters
Technische Hochschule Mittelhessen
Wiesenstraße 14, D-35390 Gießen
bjoern.loetters@mni.thm.de

Abstract

Context-Free Subphrase Grammars (CFSGs) [4] are a novel grammar formalism that generate the full class of *Context-Free Languages* (CFLs), i.e., the languages recognized by push-down automata [3]. This leaves them as an equally powerful alternative to Chomsky's *Context-Free Grammars* (CFGs) [1]. Their main advantage is that they do not inherit the notorious lack of modularity CFGs are known for, which makes them well-suited for the modular development of *Domain-Specific Languages* (DSLs). The key to this improvement is a new rule scheme that models language inclusion via the notion of subphrases.

On the technical side, these rules induce a special kind of lattice, which is called *stone* [5] or *superalgebraic lattice* [2]. These lattices come with interesting properties, one of which is their *complete distributivity*, for example. However, although these properties promise to further improve their modularity, CFSGs do not make full use of them yet. With our present work we wish to change this and propose to generalize CFSGs. In this way, substructures can be composed more freely and offer new ways to encapsulate languages in grammars.

References

- [1] Noam Chomsky. "On certain formal properties of grammars". In: *Information and Control* 2.2 (1959), pp. 137–167. ISSN: 0019-9958.
- [2] Marcel Ern , Mai Gehrke, and Aleř Pultr. "Complete Congruences on Topologies and Down-set Lattices". In: *Applied Categorical Structures* 15 (2007), pp. 163–184.
- [3] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Addison-Wesley, 2006. ISBN: 0321455363.
- [4] Bj rn L tters. "Context-Free Subphrase Grammars – A Grammar Formalism for Modular Syntax Definitions". Paper submitted for publication. 2024.
- [5] Nik Weaver. *Lipschitz Algebras*. World Scientific, 1999.

Parametrisierung von Haskell-Programmieraufgaben

Leon Koth, Janis Voigtländer
Universität Duisburg-Essen
janis.voigtlaender@uni-due.de
previously presented in extended form as [3]

Abstract

Wir berichten über Design und Verwendung einer Template- und Generator-DSL zur Parametrisierung von Übungs- und Klausuraufgaben einer Lehrveranstaltung zur Haskell-Programmierung.

In [4] wurde ein in das E-Learning-System “Autotool” [5] integrierter Aufgabentyp zur Haskell-Programmierung vorgestellt. Es wurden vielfältige Konfigurationsmöglichkeiten beschrieben, etwa hinsichtlich der Feedback-Generierung mittels Compiler, Linter und dem deklarativen Testframework QuickCheck [1]. Eine konkrete Aufgabeninstanz wird definiert durch eine Textdatei mit diversen (teils YAML-basierten) Abschnitten zur Ansteuerung der verwendeten Tools, zur Bereitstellung einer Haskell-Programmdatei mit der eigentlichen Aufgabenstellung und Platz zum Lösen sowie zur Angabe einer (gegebenenfalls versteckten) Testsuite. Zusätzlich wird eine Musterlösung angelegt, die alle konfigurierten Überprüfungen und Tests besteht. Aus diversen Gründen kann es attraktiv sein, Aufgaben zunächst zu parametrisieren, um dann verschiedene Ausprägungen irgendwie “ähnlicher” Aufgabeninstanzen erhalten zu können. Damit ist hier nicht gemeint, dass Tests mit zufälligen Werten durchgeführt werden, sondern eine tatsächliche Variabilität in der Aufgabenstellung selbst, in anderen Teilen der Konfiguration und unter Umständen notwendigerweise auch in der passenden Musterlösung.

In einer Bachelorarbeit [2] wurden entsprechende Funktionalitäten entworfen und implementiert, die seitdem sowohl im Übungsbetrieb als auch in einer Online-Klausur verwendet wurden. Der Vortrag bespricht diese Funktionalitäten und diskutiert einige Aspekte beispielgestützt.

References

- [1] K. Claessen and J. Hughes. “QuickCheck: A lightweight tool for random testing of Haskell programs”. In: *Proc. ICFP*, ACM, 2000, pp. 268–279.
- [2] L. Koth. “Parametrisierte Konfiguration von Haskell-Übungsaufgaben”. Bachelorarbeit. Universität Duisburg-Essen, 2022.
- [3] L. Koth and J. Voigtländer. “Parametrisierung von Haskell-Programmieraufgaben”. In: *Proc. ABP’23*. GI, 2023.
- [4] M. Siegburg, J. Voigtländer, and O. Westphal. “Automatische Bewertung von Haskell-Programmieraufgaben”. In: *Proc. ABP’19*. GI, 2019.
- [5] J. Waldmann. “Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen”. In: *Proc. ABP’17*. CEUR, 2017.

Artificiosa intelligentia ad portas! Or: *The end of programming, the beginning of prompting?*

Jens Knoop
Compilers and Languages
Institute of Information Systems Engineering
Faculty of Informatics
Vienna University of Technology
knoop@complang.tuwien.ac.at

Abstract

“The apparent ability of LLMs to write functioning source code has caused celebration over the potential for massive increase in programmer productivity and consternation among teachers.”

Chris Edwards. *Teaching Transformed*. CACM 64(2):12-13, 2024.

“Prompting is Programming.

A Query Language for Large Language Models.”

Luca Beurer-Kellner, Marc Fischer, Martin Vechev.
Proc. ACM Program. Lang., Vol. 7, No. PLDI, Article 186,
June 2023, 24 pages.

Prompting is programming – our students, even (especially!) our first students amend the inverse of this claim is true, too: *Programming is prompting!* And they do act as if it is true! Programming assignments commonly given to first year students designed for introducing them into the world of programming are successfully solved by *state-of-the-art* LLMs when prompted with the plain assignment text!

Is this the next evolutionary step in the eternal quest for more powerful productivity enhancing programming tools? Or is this a disruptive game changer marking a turning point which splits the time and the era of programming into a *‘before’* and an *‘after’*?

Striving for answering this question raises an array of further awkward, annoying, nasty but necessarily to be answered questions we as programming teachers have to face up:

1. **What** shall we (henceforth) teach in computer science **in** programming and **as** programming?
2. **How** shall we (henceforth) teach programming in computer science? For majors? For minors?
3. **What** shall we (henceforth) consider **‘success’** of our teaching of programming? How can we **verify** it?
4. **How** is (henceforth) computer science educated traditional and ‘prompting’ programming distinguished from non computer science educated amateur and layman DIY ‘chat away’ programming?
5. **What** distinguishes (henceforth) in terms of programming skills a computer science graduate from a non graduate? Whereby is s/he distinguished from a non graduate?

6. **How, whereby, by what kind of teaching** can we 'produce' this in such a manner distinguished computer science graduate?
7. **Last but not least, will** an in such a manner distinguished graduate be(come) the valued and respected member of the workforce academia and industry are appreciating and awaiting for?

Following an introduction into the challenges of teaching programming after the advent of LLMs, a joint effort in terms of a moderated plenary discussion shall shed more profound light on appropriate and adequate answers to the above follow-up questions of the question of "*evolution or disruption*" raised by the advent of LLMs and their easy availability.

Compileroptimierung von Curry-Code mit Haskell als Zielsprache

Kai-Oliver Prott
Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Christian-Albrechts-Platz 4, 24118 Kiel, Germany
kpr@informatik.uni-kiel.de

Abstract

Die Kompilierung von Curry-Code in Haskell stellt aufgrund der call-by-need Auswertungsstrategie von Curry eine Herausforderung dar. In diesem Vortrag präsentieren wir einen neu entwickelten Compiler, der sowohl die logische als auch die funktionale Semantik effizient vereinen soll. Ein Programm wird in Haskell mithilfe einer monadischen Abstraktion modelliert, wobei die logische Komponente der Semantik in der Implementierung der Monade abgebildet wird. Dadurch entsteht eine klare Trennung zwischen dem erzeugten Code und der Semantik. Dies erhöht die Wartbarkeit, stellt jedoch auch eine Herausforderung für die Optimierung rein funktionaler Programmteile dar.

Wir stellen eine Schnittstelle vor, die es ermöglicht, unoptimierten monadischen Code für logische Programmanteile mit effizienten, rein funktionalen Codeabschnitten zu verbinden. Dadurch wird die Leistung der generierten Programme optimiert.

A Report on Automatic Generation of Petri Net Exercise and Exam Task Instances

Marcellus Siegburg
Universität Duisburg-Essen
Fakultät für Informatik
Lotharstraße 65 (LF), D-47057 Duisburg
marcellus.siegburg@uni-due.de

Abstract

We report on generators for different task types addressing Petri net concepts from a modeling lecture for undergraduate students. A focus is on how to control difficulty and intended insights about the subject matter on the learners' side. We explain the influence of provided configuration parameters for several task types on an exemplary instance each, and comment on presentation and implementation, as well as very briefly on exam experience.

This work has previously been published in Brandt et al. [1].

References

- [1] André Brandt et al. "A Report on Automatic Generation of Petri Net Exercise and Exam Task Instances". In: *Modellierung 2022 Satellite Events*. Bonn: Gesellschaft für Informatik e.V., 2022, pp. 197–204.

Morton Feldman's "Projections One to Five" — Exploring a Classical Avant-Garde Notation by Mathematical Remodelling

Markus Lepper
semantics gGmbH, Berlin
post@mlepper.de

Baltasar Trancón y Widemann
Hochschule Brandenburg

Comprehension

The tool *tscore* allows the direct notation of data models of very different kinds which are organized along some type of time axis. This notation is organized similar to the well-proven format of musical scores from conventional *Common Western Notation (CWN)*: time flows from left to right, broken into fragments (*staves*) top down. The central notion is that of an *event*, which is uniquely defined by a pair of *time point* and *voice*. Arbitrary *parameters* can be attached to these events— for each new application, a dedicated *meta-model* explicitly defines small-scale parsers for these parameters, which are grouped into distinct lines of the source text, called *parameter tracks*. This leads to a lean bespoke input format, which makes data representations easily readable and maintainable by humans, including non-experts in computer science. *Tscore* is thus a *meta-meta-model* for musical notation. [2, 4]

Tscore has already been employed for very different use cases. With conventional CWN, it allowed the input of the complete *Die Kunst der Fuge* in just four days. It has been used to control moving abstract graphics and to construct didactic presentations of musical forms. [3]

For the TENOR 2023 conference, *tscore* has been used to re-model the compositions *Projection 1* to *Projection 5* by Morton Feldman. [5] Written in 1959, these belong to the earliest and most important pieces of pure graphical notation in classical avantgarde music.

Our re-modelling led to a computer model which allows (a) to generate the graphic representation according to the rules given by Feldman in the

prefaces of the published scores; (b) to apply automated analyses, which confirmed many of the results from the literature, obtained by manual analyses [1, 6]; (c) to acoustically realize the compositions with different pitch material and interpretation rules, by an algorithm which simulates the interactions of improvising musicians.

(The software containing and interpreting our model can be downloaded from <http://bandm.eu/feldmanProjections.html> and may, by courtesy of the publishers, freely be used for research and education.)

But the effort of pouring the informally given historic information into precise mathematics also brought new insights into the structure of the historic settings themselves, namely about (A) missing precision in the verbal instructions given by the prefaces of the published scores; (B) the derivations from his own instructions when Feldman constructed the graphical appearance of the scores; (C) the techniques and different cases when three overlapping chords must be realized on a keyboard with only two hands, etc.

Once again we showed that the technique of *mathematical remodelling* is not only an indispensable prerequisite for the construction of software tools, but also a valuable means for clarifying historical and structural facts for mere inter-human discourse.

References

- [1] David Cline. *The Graph Music of Morton Feldman*. Cambridge, UK: Cambridge University Press, 2016. ISBN: 9781316271452.

- [2] Markus Lepper and Baltasar Trancón y Widemann. “tscore: Makes Computers and Humans Talk About Time”. In: *Proc. KEOD 2013, 5th Intl. Conf. on Knowledge Engineering and Ontology Development*. Ed. by Joaquim Filipe and Jan Dietz. instincc. Portugal: scitePress, 2013, pp. 176–183. ISBN: 978-989-8565-81-5. URL: <http://markuslepper.eu/papers/tscore2013.pdf>.
- [3] Markus Lepper and Baltasar Trancón y Widemann. *Example Instances of the TScore Project Infrastructure*. <http://markuslepper.eu/sempart/tscoreInstances.html> [20-09-2023].
- [4] Markus Lepper and Baltasar Trancón y Widemann. “Translets — Parsing Diagnosis in Small DSLs, with Permutation Combinator and Epsilon Productions”. en. In: *Tagungsband des 35ten Jahrestreffens der GI-Fachgruppe “Programmiersprachen und Rechenkonzepte*. Vol. 482. IFI Reports. University of Oslo, 2018, pp. 114–129. ISBN: 978-82-7368-447-9. URL: <http://urn.nb.no/URN:NBN:no-65294>.
- [5] Markus Lepper and Baltasar Trancón y Widemann. “Morton Feldman’s “Projections One to Five” — Exploring a Classical Avant-Garde Notation by Mathematical Remodelling”. In: *Proceedings of TENOR 2024 conference*. 2024.
- [6] Ryan Vigil. “Compositional Parameters: “Projection 4” and an Analytical Methodology for Morton Feldman’s Graphic Works”. In: *Perspectives of New Music* 47.1 (2009), pp. 233–267.

Precise to Universal Configurations: Elektra to PermaplanT

Markus Raab
Compilers and Languages, TU Wien
markus.raab@complang.tuwien.ac.at

Yvonne Markl
PermaplanT
yvonne@permaplant.net

1 Problem

Pflanzen haben stark unterschiedliche Bedürfnisse an Umweltfaktoren, von welchen die Vitalität abhängt, z.B.:

- Sonne
- Feuchtigkeit
- örtliche und zeitliche Wechselwirkungen mit anderen Pflanzen

Die Schwierigkeit ist, diese Fülle an Faktoren für jede Pflanze richtig zu berücksichtigen.

Dieses Konfigurationsproblem wird umso besser gelöst, je besser die Bedürfnisse der einzelnen Pflanzen erfüllt werden.

2 Lösung

Die Bedürfnisse der Pflanzenarten werden durch eine Konfigurationsspezifikationsprache in einer umfassenden Datenbank beschrieben. Dabei wird für geometrische Berechnungen PostGIS verwendet.

PermaplanT ist eine Web-App zur Planung von biodiversen und ertragreichen Grünflächen. Sie greift via Endpoints und Server-sent events (SSE) auf die Datenbank zu.

PermaplanT ist eine Open-source-Software für alle, die auf Grünflächen sowohl Biodiversität fördern als auch einen gesteigerten Ertrag erzielen wollen. Es ist die erste Software, die holistische, gärtnerische Entscheidungshilfen anbietet.

Dabei wird folgendermaßen vorgegangen:

1. User*innen erstellen ihre virtuelle Karte (Grundriss). Dort verzeichnen sie alle ihnen bekannten Umweltfaktoren auf verschiedenen Ebenen.
2. Aus der Datenbank von über 10.000 Einträgen werden die gewünschten Pflanzen gewählt und platziert.
3. User*innen erhalten sofortiges Feedback für bestmögliche Standorte, Wechselwirkungen, negative Einflüsse, ökologischen Nutzen, etc.

Neuartig ist dabei, dass PermaplanT:

- Pflanzen zu beliebigen Zeitpunkten/Orten zulässt,
- unter Berücksichtigung der Vor-/Nachkulturen über mehrere Jahre hinweg,
- Diversität auch jenseits von Gemüse/Obst fördert,
- kollaboratives Planen ermöglicht,
- Umweltfaktoren berücksichtigt und
- In-situ-Forschungsvorhaben unterstützt.

3 Mehrwert

In Krisenzeiten erleben wir ein Come-back von Selbstversorgung und Eigenständigkeit. Diese Entwicklung ist wichtig und zeigt einen bewussteren Umgang mit dem eigenen Ressourcenverbrauch auf.

PermaplanT fördert dieses Bewusstsein, führt Gleichgesinnte zusammen und prägt die Gesellschaft nachhaltig.

Durch den Einsatz von PermaplanT wird u.a. die Wiedereinführung von "Trittsteinbiotopen", klein

strukturierte Ökosysteme für heimische Pflanzen und Tiere, gefördert und Artenschwund sowie Klimawandel entgegengewirkt.

Die eigene Obst- und Gemüseernte spart Transportwege und spart CO_2 -Emissionen.

4 Zielgruppe

Für ein realistisches Bild der Zielgruppe haben wir eine Umfrage durchgeführt, bei welcher 311 Personen teilgenommen haben. Darunter waren 157 Österreicher*innen mit Zugang zu Grünflächen. Das Medianalter lag bei 37 Jahren und die durchschnittliche Gartenerfahrung bei 10 Jahren.

Innerhalb der interessierten Zielgruppe soll die App für folgende Zwecke hauptsächlich verwendet werden:

- zur Steigerung der Biodiversität (69 %)
- für einen besseren Überblick im Garten (55 %)
- zur Verbesserung des Ertrags (45 %)

5 Verwandte Arbeiten

Für das Anlegen eines Grünflächen-Konzepts ist bislang entweder

- das händische Zeichnen von Plänen auf Transparentblättern oder
- Architektur-Software (bei CAD Vorkenntnissen)

üblich.

Obwohl diese Methoden zum Ziel führen können, sind sie sehr zeitaufwendig und fehleranfällig. Wälzen von vielen Sachbüchern ist zwingend notwendig. Die Übersicht geht nach ein paar Jahren oftmals verloren. PermaplanT kann als spezialisierte CAD Implementierung verstanden werden.

Mehrere Apps bieten das Planen von Gemüsebeeten an. Alle Anbieter berechnen jedoch hauptsächlich die Nachbarschaftsbeziehungen zwischen den Pflanzen, ein aus gärtnerischer Sicht niedrig zu priorisierender Faktor. Die viel ausschlaggebenderen Faktoren, nämlich Umwelteinflüsse wie Licht oder Wasser bleiben bei diesen Anbietern gänzlich unberücksichtigt, d.h. das

eigentliche Konfigurationsproblem wird kaum unterstützt.

Konfigurationsmanagement wird in mehreren Communities erforscht:

Benutzereinstellungen [2, 4, 5, 25]

Variabilitätspunkte [6, 20, 23, 24]

Ableitungsentscheidungen [3]

Entscheidungsprozess [19]

Ontology [21]

Produktlinien [22]

Massenanfertigung [1]

Semantisch [18]

Achtsamkeit bezüglich Kontext [7–17]

Dabei ist der Fokus üblicherweise domänenspezifisch. Mit PermaplanT wollen wir erforschen, ob und wie weit ehemalige Ergebnisse auf die möglicherweise umfangreichste und wichtigste Problemstellung unserer Zeit umzulegen ist: die diverse Bepflanzung der Erdoberfläche in Zeiten des Klimawandels.

References

- [1] Eric Arnold Anderson. "Researching system administration". PhD thesis. University of California at Berkeley, 2002.
- [2] Paul Anderson. "Towards a High-Level Machine Configuration System." In: *LISA*. Vol. 94. 1994, pp. 19–26.
- [3] Software Productivity Consortium Services Corporation. *Reuse-driven Software Processes Guidebook: SPC-92019-CMC, Version 02.00. 03*. Software Productivity Consortium Services Corporation, 1993.
- [4] Peng Huang et al. "ConfValley: a systematic configuration validation framework for cloud services." In: *EuroSys*. 2015, p. 19.
- [5] Dongpu Jin et al. "Configurations Everywhere: Implications for Testing and Debugging in Practice". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: ACM, 2014, pp. 215–224. ISBN: 978-1-4503-2768-8. URL: <http://dx.doi.org/10.1145/2591062.2591191>.

- [6] Kim Mens et al. “A Taxonomy of Context-Aware Software Variability Approaches”. In: *Workshop on Live Adaptation of Software Systems, collocated with Modularity 2016 conference*. 2016.
- [7] Markus Raab. “Elektra: universal framework to access configuration parameters”. In: *The Journal of Open Source Software* 1.8 (Dec. 2016), pp. 1–2. URL: <http://dx.doi.org/10.21105/joss.00044>.
- [8] Markus Raab. “Global and Thread-Local Activation of Contextual Program Execution Environments”. In: *Proceedings of the IEEE 18th International Symposium on Real-Time Distributed Computing Workshops (ISOR-CW/SEUS)*. Apr. 2015, pp. 34–41.
- [9] Markus Raab. “Improving System Integration Using a Modular Configuration Specification Language”. In: *Companion Proceedings of the 15th International Conference on Modularity*. MODULARITY Companion 2016. Málaga, Spain: ACM, 2016, pp. 152–157. ISBN: 978-1-4503-4033-5. URL: <http://dx.doi.org/10.1145/2892664.2892691>.
- [10] Markus Raab. “Persistent Contextual Values As Inter-process Layers”. In: *Proceedings of the 1st International Workshop on Mobile Development*. Mobile! 2016. Amsterdam, Netherlands: ACM, 2016, pp. 9–16. ISBN: 978-1-4503-4643-6. URL: <http://dx.doi.org/10.1145/3001854.3001855>.
- [11] Markus Raab. “Safe Management of Software Configuration”. In: *Proceedings of the CAISE'2015 Doctoral Consortium*. urn:nbn:de:0074-1415-4: <http://ceur-ws.org/Vol-1415/>, 2015, pp. 74–82. URL: <http://ceur-ws.org/Vol-1415/CAISE2015DC09.pdf>.
- [12] Markus Raab. “Sharing Software Configuration via Specified Links and Transformation Rules”. In: *Technical Report from KPS 2015*. Vol. 18. Vienna University of Technology, Complang Group. 2015.
- [13] Markus Raab. “Unanticipated Context Awareness for Software Configuration Access Using the getenv API”. In: *Computer and Information Science*. Cham: Springer International Publishing, 2016, pp. 41–57. ISBN: 978-3-319-40171-3. URL: http://dx.doi.org/10.1007/978-3-319-40171-3_4.
- [14] Markus Raab and Gergö Barany. “Challenges in Validating FLOSS Configuration”. In: *Open Source Systems: Towards Robust Practices: 13th IFIP WG 2.13 International Conference, OSS 2017, Buenos Aires, Argentina, May 22-23, 2017, Proceedings*. Ed. by Federico Balaguer et al. Cham: Springer International Publishing, 2017, pp. 101–114. ISBN: 978-3-319-57735-7. URL: http://dx.doi.org/10.1007/978-3-319-57735-7_11.
- [15] Markus Raab and Gergö Barany. “Introducing Context Awareness in Unmodified, Context-unaware Software”. In: *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE, INSTICC*. ScitePress, 2017, pp. 218–225. ISBN: 978-989-758-250-9.
- [16] Markus Raab and Franz Puntigam. “Program Execution Environments As Contextual Values”. In: *Proceedings of 6th International Workshop on Context-Oriented Programming*. Uppsala, Sweden: ACM, 2014, 8:1–8:6. ISBN: 978-1-4503-2861-6. URL: <http://dx.doi.org/10.1145/2637066.2637074>.
- [17] Markus Raab et al. “Unified Configuration Setting Access in Configuration Management Systems”. In: *Proceedings of the 28th International Conference on Program Comprehension*. 2020, pp. 331–341.
- [18] Steven Raemaekers, Arie Van Deursen, and Joost Visser. *Semantic versioning versus breaking changes: A study of the maven repository*. Tech. rep. uuid: 57a68419 - 8c92 - 445f - 9c23 - 0fb36ece0cde. Delft University of Technology, Software Engineering Research Group, 2014. URL: <http://resolver.tudelft.nl/uuid:57a68419-8c92-445f-9c23-0fb36ece0cde>.
- [19] Mark-Oliver Reiser. *Core Concepts of the Compositional Variability Management Framework (CVM): A Practitioner's Guide*. TU, Professoren der Fak. IV, 2009.
- [20] Alexander von Rhein et al. “Variability encoding: From compile-time to load-time variability”. In: *Journal of Logical and Algebraic Methods in Programming* 85.1, Part 2 (2016). Formal Methods for Software Product Line Engineering, pp. 125–145. ISSN: 2352-2208. URL: <http://www.sciencedirect.com/science/article/pii/S2352220815000577>.
- [21] Timo Soinen et al. “Towards a general ontology of configuration”. In: *AI EDAM* 12.04 (1998), pp. 357–372.

- [22] Tommi Syrjänen. *A Rule-Based Formal Model For Software Configuration*. 1999.
- [23] Jilles Van Gorp, Jan Bosch, and Mikael Svahnberg. "On the notion of variability in software product lines". In: *Software Architecture, 2001. Proceedings. Working IEEE/I-FIP Conference on*. IEEE. 2001, pp. 45–54.
- [24] Karina Villela et al. "A Survey on Software Variability Management Approaches". In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. SPLC '14. Florence, Italy: ACM, 2014, pp. 147–156. ISBN: 978-1-4503-2740-4. URL: <http://dx.doi.org/10.1145/2648511.2648527>.
- [25] Zuoning Yin et al. "An Empirical Study on Configuration Errors in Commercial and Open Source Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 159–172. ISBN: 978-1-4503-0977-6.

Inference and Verification of Arithmetic Non-Fail Conditions in Declarative Programs

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

Abstract

Functions containing arithmetic operations have often restrictions which cannot be expressed by the type system of the programming language. The division operation requires that the divisor is non-zero and the factorial function should not be applied to negative numbers. Such partial operations might lead to program crashes if they are applied to unintended arguments. Checking the arguments before each call is tedious and decreases the run-time efficiency. To avoid these disadvantages and support the safe use of partially defined operations, we present an approach to verify the correct use of operations at compile time. To reduce the efforts for the programmer, our approach automatically infers non-fail conditions of operations from their definitions and checks whether these conditions are satisfied for all uses of the operations. For the latter task, an SMT solver is used to verify arithmetic non-fail conditions. This approach is implemented for functional logic Curry programs so that it can also be used for purely functional or logic programs. Since it is combined with a technique to infer non-fail conditions for programs using algebraic data types, only a few arithmetic non-fail conditions need to be inferred to verify larger modules.

1 Overview

Programs often contain operations which are partially defined, i.e., which do not yield meaningful results for particular argument values. A typical example is the division operation which is not defined if the divisor is zero. User-defined operations might also have restrictions on argument values. For instance, consider the following definition of the factorial function in the functional language Haskell [3]:

```
fac :: Int -> Int
fac n | n == 0 = 1
      | n > 0  = n * fac (n - 1)
```

Due to the conditions “ $n == 0$ ” and “ $n > 0$ ”, a run-time error occurs if `fac` is applied to a negative number since there is no branch for this case. Such an error might be avoided at compile time

by restricting the argument type of `fac` to natural numbers and checking whether each call satisfy this restriction. Unfortunately, this restriction is not expressible in type systems of current strongly typed declarative programming languages, such as Haskell. This is also due to the fact that a call like `fac (m-n)` must be considered as ill-typed if m is smaller than n , i.e., the correct typing depends on values available at run time.

In order to avoid program crashes due to such errors, one could transform the factorial function into a total function that returns a specific value indicating a meaningless result. In Haskell, this could be expressed by using the predefined type of partial values

```
data Maybe a = Nothing | Just a
```

`Nothing` represents “no value” and `Just x` the value x . Using this type, we could define a “to-

talized” version of `fac` as follows:

```
facT :: Int → Maybe Int
facT n | n < 0 = Nothing
      | n == 0 = Just 1
      | n > 0 =
        case facT (n - 1) of
          Nothing → Nothing
          Just m  → Just (n * m)
```

This total programming style yields ugly and less comprehensible code (note that also each client of `facT` has to check and transform the computed result). Moreover, the code is less efficient due to the additional case distinction in each recursive call.

In order to use the partially defined function `fac` without the risk of run-time errors, one can check the value of the argument before the actual call. For instance, the following code snippet defines an operation to read a number and, if it is non-negative, prints its factorial (`readInt` reads a string from the user input until it is an integer):

```
printFac = do
  putStr "Factorial computation for: "
  n <- readInt
  if n < 0
    then do putStrLn "Negative number!"
            printFac
    else print (fac n)
```

By checking the value of `n` before evaluating `(fac n)`, it is ensured that `fac` does not fail.

In order to verify programs for the absence of failures, we developed a fully automatic tool¹ which can verify the non-failure of this program. For this purpose, our tool infers the *non-fail condition*

```
fac'nonfail :: Int → Bool
fac'nonfail n = (n == 0) || (n > 0)
```

from the definition of `fac`. Then it checks whether this condition is satisfied at all call sites of `fac`. For instance, it is satisfied for the recursive call `fac (n - 1)` since `n > 0`. This property is automatically checked by an SMT solver [2]. The entire process is iterative since a non-fail condition for some operation `f` might require new non-fail conditions for operations that use `f`. For instance, the inference of the non-fail condition of `fac` causes a new non-fail condition for the operation

```
fac2 n = n * fac (n + 2)
```

The techniques used in this tool are based on previous work on the inference of non-fail conditions for algebraic data types [1] and combines this work with new techniques to infer non-fail conditions for functions involving arithmetic operations.

References

- [1] M. Hanus. “Inferring Non-Failure Conditions for Declarative Programs”. In: *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*. Springer LNCS 14659, 2024, pp. 167–187.
- [2] L. de Moura and N. Bjørner. “Z3: An Efficient SMT Solver”. In: *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. Springer LNCS 4963, 2008, pp. 337–340.
- [3] S. Peyton Jones, ed. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

¹ The tool, available at <https://cpm.curry-lang.org/pkgs/verify-non-fail.html>, is implemented as a Curry package for easy installation.

Anwendung einer adapterbasierten Effektmodellierung zur Implementierung eines Interpreters

Niels Bunkenburg
CAU Kiel
nbu@informatik.uni-kiel.de

Abstract

Interpreter sind eine gängige Methode, die Semantik einer Programmiersprache zu definieren. Verglichen mit Compilern sind Interpreter einfacher zu implementieren und besonders nützlich für Prototyping. Abgesehen von Lexing und Parsing ist ein Interpreter eine Funktion, die den abstrakten Syntaxbaum eines Programms verarbeitet und einen Wert zurückgibt, der das Ergebnis der Programmausführung darstellt. Die Interpretationsfunktion entscheidet dabei, wie jedes einzelne syntaktische Konstrukt der Eingabesprache interpretiert werden soll. Wir definieren eine domänenspezifische Sprache, die algebraische Effekte, Effektadapter und Effekthandler nutzt, um den syntaktischen Konstrukten einer Sprache auf modulare Weise eine Semantik zuzuweisen. Am Beispiel der Implementierung eines Interpreters für die funktional-logische Programmiersprache Curry zeigen wir nicht nur, wie man Konzepte wie Literale, Funktionsdeklarationen oder Pattern-Matching implementiert, sondern auch wie, unüblichere Sprachfeatures wie Currys Nichtdeterminismus, Lazy Evaluation und freie Variablen umgesetzt werden können. Schließlich evaluieren wir die Vorteile unseres Ansatzes hinsichtlich Modularität und Erweiterbarkeit sowie die Auswirkung auf die Geschwindigkeit des Interpreters.

Java-TX Compiler in Java-TX

Julian Schmidt, Martin Plümicke
 Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
 Department of Computer Science
 Florianstraße 15, 72160 Horb
 m.pluemicke@hb.dhbw-stuttgart.de

Abstract

Seit rund 15 Jahren wird an der Dualen Hochschule Baden Württemberg Campus Horb an einer Java-Erweiterung namens Java-TX (Type eXtended) auf Basis von Java 8 geforscht. Java-TX zielt darauf ab, die Java-Programmiersprache durch funktionale Programmierkonzepte wie globale Typinferenz und echte Funktionstypen für Lambda-Ausdrücke zu erweitern. Im Bereich des Compilerbaus gilt die Selbstübersetzung eines Compilers als wichtiges Qualitätsmerkmal. Aus diesem Grund wird im Rahmen dieser Arbeit der in Java verfasste Java-TX-Compiler, so weit es möglich ist, in Java-TX übersetzt. Dabei werden der Funktionsumfang und die Praxis-tauglichkeit des aktuellen Java-TX-Zustands untersucht und bestehende Probleme sowie fehlende Funktionen aufgezeigt. Zusätzlich werden durch die Überarbeitung des Quellcodes die Vorteile von Java-TX im Vergleich zu Java demonstriert.

1 Einleitung

Java-Type eXtended (Java-TX) basiert auf Java 8 und wurde um zwei wesentliche Funktionen erweitert, die aus funktionalen Programmiersprachen wie Haskell bekannt sind: Globale Typinferenz und echte Funktionstypen. Im folgenden Abschnitt werden diese beiden Konzepte genauer erläutert.

1.1 Typinferenz in Java-TX

Trotz verschiedener Arten der lokalen Typinferenz, die in den letzten Jahren zu Java hinzugefügt wurden, müssen bis heute die Typen in Methodensignaturen und Feldern explizit vom Entwickler angegeben werden. In funktionalen Programmiersprachen wie Haskell existiert hingegen eine globale Typinferenz. Diese erlaubt es auch, die Typen der Parametern und des Rückgabewerts einer Methode vom Compiler ableiten zu

lassen [20][S.323 ff]. In Listing 1 ist die Funktion `add`, welche zwei numerische Werte addiert, in Haskell gegeben. Wie man sehen kann, beinhaltet der Quellcode keinerlei Typinformationen. Dennoch ist Haskell eine statisch typisierte Sprache [5][S.3]. Das bedeutet, dass sämtliche Typinformationen bereits zur Kompilierzeit überprüft werden [10][S.2]. Der Haskell Compiler inferiert die Typen also zur Kompilierzeit. Im Fall der Funktion `add`, wird der Typ `add :: Num a => (a, a) -> a` errechnet. Dieser Typ bedeutet, dass die Funktion `add` zwei Werte vom Typ `a` entgegennimmt und einen Wert vom selben Typ zurückgibt. `a` muss dabei von der Typklasse `Num` sein, was in Haskell die numerischen Typen einschließt [5][S.76, 81]. Die Funktion `add` kann also sowohl mit ganzen Zahlen als auch mit Gleitkommazahlen aufgerufen werden.

```
1 add (x, y) = x+y
```

Listing 1: Funktion `add` in Haskell

Eines der primären Ziele von Java-TX ist es daher, die globale Typinferenz in Java zu er-

möglichen, um auch Typen von Feldern und Methodensignaturen vom Compiler inferieren lassen zu können.

Ein Beispiel für die Funktion `add` in Java-TX ist in Listing 2 zu sehen.

```
1 import java.lang.Integer;
2 class Add{
3     public add(a, b) {
4         return a + b;
5     }
6 }
```

Listing 2: Untypisierte Methode `add`

In diesem Beispiel werden ähnlich wie in Listing 1 sowohl die Parametertypen als auch der Rückgabotyp nicht angegeben. Diese Typen werden zur Kompilierzeit anhand des Kontextes abgeleitet. Die resultierenden Typen der Methode sind in Listing 3 gegeben. Der Compiler weiß, dass der Operator `+` nur für numerische Datentypen und Zeichenketten definiert ist. Da der Algorithmus für die globale Typinferenz allerdings sehr rechenintensiv ist, werden nur explizit importierte Typen berücksichtigt. In diesem Beispiel wird nur `java.lang.Integer` importiert, daher spielen andere numerische Typen und Zeichenketten keine Rolle. Neben den Parametern berechnet der Compiler auch einen möglichen Rückgabotyp. In diesem Fall hat dieser auch den Datentyp `java.lang.Integer`, da die Addition von zwei Integern wieder einen Integer ergibt.

```
1 import java.lang.Integer;
2 class Add{
3     public Integer add(Integer a,
4                       Integer b) {
5         return a + b;
6     }
7 }
```

Listing 3: Von Typinferenz errechnete Typen für Listing 2

Ähnlich wie C++ Templates erlaubt dieses Vorgehen, Methoden für verschiedene Datentypen mit gleicher Implementierung nur einmal zu definieren. Der Unterschied besteht darin, dass Java-TX den Code für alle Typen, die importiert wurden und die verwendeten Funktionalitäten unterstützen, generiert. C++ generiert den Code nur für die Typen, mit denen das Template explizit aufgerufen wird [21][Abschnitt 2.1.2]. Dies ermöglicht es, den Code zu vereinfachen und die Wartbarkeit zu erhöhen. Ein Beispiel dazu ist in Listing 4 zu sehen.

```
1 import java.lang.Integer;
2 import java.lang.String;
3 import java.lang.Double;
4
5 public class Add{
6     public add(a, b) {
7         return a + b;
8     }
9 }
```

Listing 4: Methodenüberladungen durch Typinferenz

In diesem Fall erzeugt der Compiler die Methode `add` jeweils mit der Signatur für die Datentypen `java.lang.Integer`, `java.lang.String` und `java.lang.Double`, wie in Listing 5 gezeigt ist.

```
1 import java.lang.Integer;
2 import java.lang.String;
3 import java.lang.Double;
4 public class Add{
5     public Integer add(Integer a,
6                       Integer b) {
7         return a + b;
8     }
9
10    public Double add(Double a,
11                     Double b) {
12        return a + b;
13    }
14
15    public String add(String a,
16                     String b) {
17        return a + b;
18    }
19 }
```

Listing 5: Resultat der Typinferenz für Listing 4

1.2 Echte Funktionstypen in Java-TX

Die Funktionstypen werden in Java-TX laut [12] mit der folgenden Syntax definiert:

$$(\tau_1, \tau_2, \dots, \tau_N) \rightarrow \tau_0 \equiv \text{Fun}N\$\$ \langle \tau_1, \tau_2, \dots, \tau_N, \tau_0 \rangle$$

$$(\tau_1, \tau_2, \dots, \tau_N) \rightarrow \text{void} \equiv \text{FunVoid}N\$\$ \langle \tau_1, \tau_2, \dots, \tau_N \rangle$$

Außerdem sind Argumente der Funktionstypen automatisch kontravariant und der Rückgabewert kovariant. Dies erlaubt die Subtypisierung von Funktionstypen ohne use site Varianz, was zur besseren Lesbarkeit und weniger Fehleranfälligkeit führt. Tatsächlich ist die Verwendung von Wildcards für die Java-TX Funktionstypen nicht erlaubt [11][Abschnitt 6]. In Listing 6 ist

ein Beispiel für die Subtypisierung von Funktionstypen in Java-TX zu sehen ist. Hier lässt sich die Funktion `func1` der Funktion `func2` zuweisen, da das Argument der Funktion `func1` ein Supertyp des Arguments der Funktion `func2` ist und der Rückgabewert der Funktion `func1` ein Subtyp des Rückgabewerts der Funktion `func2` ist.

```
1 Fun1$$<Number, Integer> func1 =
2     x -> x.intValue() + 1;
3 Fun1$$<Integer, Integer> func2 =
4     func1;
```

Listing 6: Subtypisierung von Funktionstypen in Java-TX

Echte Funktionstypen ermöglichen in Java-TX die Definition von Funktionen in einem beliebigen Kontext. Ein Beispiel hierzu ist in Listing 7 zu sehen. In diesem Fall inferiert der Compiler den Typ `Fun2$$<Integer, Integer, Integer>`. In Java ist die Verwendung von `var` in diesem Kontext nicht möglich.

```
1 import java.lang.Integer;
2 public class Main{
3     main(){
4         var add = (x, y) -> x + y;
5     }
6 }
```

Listing 7: Lambda Ausdruck ohne Typkontext

1.3 Selbstkompilierende Compiler

Im Compilerbau gilt ein selbstkompilierender Compiler als Qualitätsmerkmal. Ein Compiler ist selbstkompilierend, wenn er in der Sprache geschrieben ist, die er kompiliert. Dies bedeutet, dass der Compiler in der Lage ist, seinen eigenen Quellcode zu kompilieren. In der Regel gibt es in jeder großen Programmiersprache mindestens einen Compiler, der selbstkompilierend ist. Bekannte Beispiele hierfür sind der `javac` Compiler, der in Java geschrieben ist [9] und Java Code kompiliert und der C++ Compiler der GNU Compiler Collection (GCC), der seit einigen Jahren in C++ geschrieben ist [3] und unter anderem C++ Code kompilieren kann [15][S.5]. Laut [18][Abschnitt 3.4] bieten selbstkompilierende Compiler im Wesentlichen 4 Vorteile:

- Sie stellen einen nicht trivialen Test der Funktionsfähigkeit des Compilers dar.

- Sobald der selbstkompilierende Compiler einmal implementiert ist, ist die Entwicklung ohne die Abhängigkeit anderer Übersetzungssysteme möglich.
- Alle Verbesserungen, die am Backend des Compilers vorgenommen werden, wirken sich sowohl auf den Compiler als auch auf den generierten Code aus.
- Sie bieten eine umfassende Überprüfung der Selbstkonsistenz des Compilers. Der Compiler sollte in der Lage sein, seinen eigenen Quellcode zu kompilieren.

Der Prozess, einen selbstkompilierenden Compiler zu erstellen, kann mit sogenannten T-Diagrammen visualisiert werden. In Abbildung 1 ist ein solches Diagramm für einen selbstkompilierenden Compiler in Java-TX zu sehen. Auf der linken Seite ist dabei die Eingabesprache zu sehen, auf der rechten Seite die Ausgabesprache und in der Mitte die Sprache, in der der Compiler geschrieben ist. In diesem Fall ist das rote T der bereits bestehende Compiler in Java-TX, der den Java-TX Code in Bytecode übersetzt. Der blaue T ist das langfristige Ziel des Projekts „Java-TX Compiler in Java-TX“, also ein Compiler der in Java-TX geschrieben ist und Java-TX Quelldateien in Bytecode kompiliert. Die Anordnung der Ts symbolisiert, dass der ursprüngliche Compiler den neuen Compiler initial kompiliert.

2 Aufbau der Umgebung

2.1 Voraussetzungen

Der aktuelle „Java-TX Compiler“ ist in Java implementiert. Da Java-TX ein Superset von Java 8¹ ist, kann der Quellcode im Wesentlichen übernommen werden. Es müssen lediglich die inferierbaren Typinformationen entfernt werden, um die Vorteile von Java-TX auch zu nutzen. Da sowohl Java, als auch Java-TX Java Bytecode generieren, ist es möglich, Java-TX und Java Dateien zu mischen. Dadurch kann der Compiler sukzessive in Java-TX übersetzt werden. Der Vorteil daran ist, dass man nach jeder übersetzten Datei einen funktionsfähigen Compiler hat, der zu einem gewissen Grad bereits aus Java-TX Dateien besteht. Auf diesem Zwischenstand können dann z.B. auch Tests aufgeführt werden, um die Funktionalität zu prüfen.

¹ Es existieren einige Einschränkungen vgl. [12][S.2]

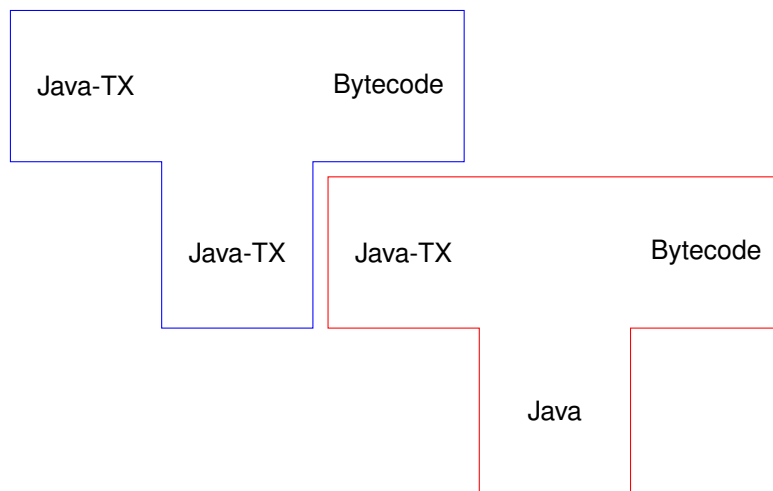


Figure 1: Selbstkompilierender Compiler in Java-TX

Ein Problem besteht darin, dass der „Java-TX Compiler“ `.jav2`-Dateien und `.class`-Dateien lesen kann, während der `javac` Compiler `.java`-Dateien und `.class`-Dateien einlesen kann. Jedoch kann der „Java-TX Compiler“ keine `.java`-Dateien lesen und der `javac` Compiler natürlich keine `.jav`-Dateien. Das bedeutet, dass zirkuläre Abhängigkeiten zwischen Java und Java-TX Dateien nicht ohne Weiteres möglich sind. Dieses Problem ist in Abbildung 2 visualisiert. Ein Pfeil symbolisiert dabei Abhängigkeiten zu einer anderen Datei. Wenn der „Java-TX Compiler“ zuerst aufgerufen wird, kann er die Java Datei nicht lesen und umgekehrt. Selbst wenn die Abhängigkeiten nicht zirkulär sind, muss die genaue Reihenfolge der Kompilierung definiert sein. Dies bedeutet das der Build-Prozess für den „Java-TX Compiler in Java-TX“ sehr komplex werden würde.

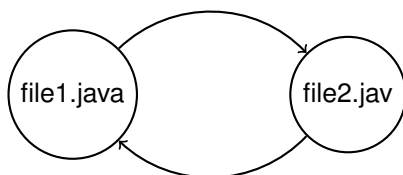


Figure 2: Zirkuläre Abhängigkeiten zwischen Java und Java-TX Dateien

Aktuell wird dieses Problem gelöst indem der gesamten Compiler zu Beginn einmal in seiner Ursprungsform (also nur `.java` Dateien) mit `javac` kompiliert wird. Dann liegt der komplette „Java-TX Compiler“ in Form von `.class` Dateien vor.

² Java-TX Quelldateien haben die Dateierdung `.jav`

Anschließend muss diese Hierarchie nur noch als Classpath des „Java-TX Compiler“ angegeben werden. Der „Java-TX Compiler“ kann nun die `.class` Dateien lesen, die zur jeweiligen `.java` Datei korrespondieren. Wichtig ist an dieser Stelle natürlich, dass die `.class`-Dateien den gleichen Stand wie die Quelldateien haben. Dieses Verhalten ist in Abbildung 3 gezeigt. Die Grafik zeigt, dass der „Java-TX Compiler“ zur Kompilierung der `.jav` Datei die zuvor generierte `.class` Datei von `file1` verwendet. Wenn später dann `file1` von `javac` kompiliert wird, wurde `file2.class` bereits vom „Java-TX Compiler“ erstellt. Der `javac` Compiler benötigt die vorkompilierten Klassen also nicht mehr im Classpath und kann die vom „Java-TX Compiler“ generierten `.class` Dateien verwenden. Natürlich ist es für diese Lösung notwendig, dass der „Java-TX Compiler“ zuerst alle `.jav` Dateien kompiliert, bevor der `javac` Compiler auf Basis der generierten `.class` Dateien, alle `.java` kompiliert. Diese Reihenfolge ist aber ohnehin notwendig, damit `javac` seinen Bytecode mit den korrekten Typen generieren kann, die der „Java-TX Compiler“ inferiert hat. Diese Lösung ist zwar nicht ideal, da man eine Abhängigkeit zum gesamten vorkompilierten Projekt benötigt. Da der „Java-TX Compiler“ aber keine `.java` Dateien lesen kann, ist es vermutlich die einzige Möglichkeit, zirkuläre Abhängigkeiten zwischen `.java` und `.jav` Dateien zu lösen.

Die aktuelle „Java-TX Compiler“ Implementierung automatisiert den Build-Prozess mit Maven. Maven ist für die Verwendung von Java basierten Sprachen entwickelt worden [6]. Da Maven keine Java-TX Unterstützung bietet, muss der Build-Prozess für den „Java-TX Compiler in Java-TX“

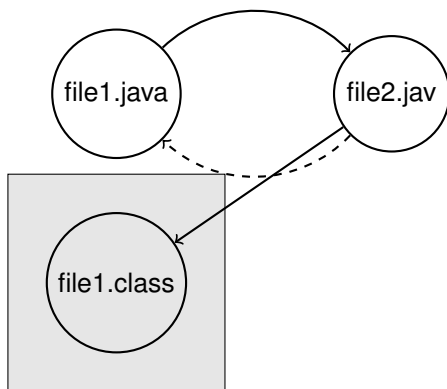


Figure 3: Zirkuläre Abhängigkeiten zwischen Java und Java-TX Dateien behoben

manuell angepasst werden. Dazu wurden zwei Lösungsansätze erarbeitet, die in den folgenden Abschnitten beschrieben werden.

2.2 Kompilierung mit Make

Der erste Ansatz der verfolgt wurde, war die Kompilierung mittels GNU's Not Unix (GNU) make umzusetzen. Make ist sehr flexibel und ist vielseitig einsetzbar. Um nicht alle Quelldateien einzeln angeben zu müssen, werden Wildcards verwendet. Außerdem bietet Make den Vorteil, dass nur geänderte Dateien neu kompiliert werden. Dies ist besonders bei großen Projekten von Vorteil, da nicht alle Dateien neu kompiliert werden müssen. Das verwendete Make-File ist in Listing 8 zu sehen.

Im ersten Abschnitt werden einige globale Variablen, wie der Ort des javac und „Java-TX Compiler“ und der Quell- und Zielordner definiert. Dann werden mit dem shell Befehl `find` alle Dateien, die auf `.java` und `.jav` enden, in den Variablen `JAVASOURCES` und `JAVSOURCES` gespeichert. Der `find` Befehl geht dabei rekursiv vor und beachtet auch alle Unterverzeichnisse des Quellverzeichnis. In diesem Fall findet der Befehl also alle Dateien in der Packethierarchie. Danach wird der `patsubst` Befehl von Make verwendet, um den Pfand, an dem die korrespondierende `.class` Datei liegen wird, zu substituieren. So wird z.B. die Liste der Quelldateien

```
javatx-src/main/java/de/dhbwstuttgart/
  typeinference/assumptions/
  Assumption.java
javatx-src/main/java/de/dhbwstuttgart/
  typeinference/assumptions/
  FieldAssumption.java
```

zu einer Liste mit folgenden korrespondierenden Zieldateien:

```
out/de/dhbwstuttgart/typeinference/
  assumptions/Assumption.class
out/de/dhbwstuttgart/typeinference/
  assumptions/FieldAssumption.class
```

Die genaue Syntax des `patsubst` Befehls kann in [16][S.92] nachgelesen werden. In der nächsten Zeile wird die Standardregel für das Makefile definiert. In GNU Make ist die erste Regel diejenige, die aufgerufen wird, wenn der Make-Befehl ohne Parameter aufgerufen wird [16][S.109]. Diese Regel versucht, alle Dateien in `$(JAVACLASSES)` und `$(JAVSOURCES)` zu kompilieren. Dazu werden die nächste beiden Regeln verwendet.

Die erste Regel kompiliert Java-TX Dateien. Dazu liegt der Compiler als JAR-Archiv vor. Der Compiler wird mit dem Befehl `java -jar` aufgerufen, welcher die Ausführung von JAR-Archiven ermöglicht. Der `-d` Parameter gibt den Zielordner an, in dem die `.class` Datei gespeichert wird. Der `-cp` Parameter gibt die Klassenpfade an, die der Compiler benötigt. In diesem Fall sind dies der Quellordner, der Zielordner und die Abhängigkeiten des Projekts. Der `%` Operator gibt den Pfad der ersten Abhängigkeit an [16][S.131]. In diesem Fall ist das die `.jav` Quelldatei. Das `%` Symbol dient als Wildcard und steht für eine beliebige Zeichenkette. Eine solche Regel nennt man auch Musterregel (engl. Pattern Rule) [16][S.129].

Die zweite Regel kompiliert `.java` Quelldateien. Sie unterscheidet sich nur marginal von der ersten Regel. Lediglich der Aufruf des Compilers und die Dateiendung der Regel wurden angepasst, da hierzu der javac Compiler verwendet wird. Die Parameter `-d` und `-cp` sind identisch. Lediglich einige weitere Parameter, die anfangs in die Variable `JFLAGS` gespeichert wurden, werden dem Compiler übergeben. Der Parameter `|g:none|` bedeutet, dass keine Debuginformationen generiert werden sollen. Das macht den generierten Bytecode in solch einem Entwicklungsumfeld leichter lesbar. Der Parameter `|implicit:none|` bedeutet, dass Abhängigkeiten nicht impliziert kompiliert werden sollen. Weiteres dazu in Unterabschnitt 2.2.1. Der Parameter `nowarn` bedeutet schließlich, dass keine Warnungen ausgegeben werden sollen.

Die letzte Regel des Makefiles ist eine Phony-Regel. Sie löscht den Inhalt des Zielordners.

```
1 JFLAGS = -g:none -implicit:none -nowarn
2 JC = javac
3 JTX = JavaTXcompiler-1.0-jar-with-dependencies.jar
4 SRCDIR = javatx-src/main/java
5 DESTDIR = out
6
7 # Use find to locate all .java and .jav files recursively
8 JAVASOURCES := $(shell find $(SRCDIR) -name '*.java')
9 JAVSOURCES := $(shell find $(SRCDIR) -name '*.jav')
10
11 # Convert .java files to .class files with the same directory structure
12 JAVACLASSES := $(patsubst $(SRCDIR)/%.java,$(DESTDIR)/%.class,$(JAVASOURCES))
13 JAVCLASSES := $(patsubst $(SRCDIR)/%.jav,$(DESTDIR)/%.class,$(JAVSOURCES))
14
15 default: $(JAVCLASSES) $(JAVACLASSES)
16
17 # Rule for compiling .jav files
18 $(DESTDIR)/%.class: $(SRCDIR)/%.jav
19     java -jar $(JTX) -d "$(DESTDIR)" -cp "$(SRCDIR):$(DESTDIR):target/
20         dependencies/" $<
21
22 # Rule for compiling .java files
23 $(DESTDIR)/%.class: $(SRCDIR)/%.java
24     $(JC) -d $(DESTDIR) -cp "$(SRCDIR):$(DESTDIR):target/dependencies/*" $(
25         JFLAGS) $<
26
27 .PHONY: clean
28 clean:
29     $(RM) -r $(DESTDIR)/*
```

Listing 8: Makefile für die Kompilierung des „Java-TX Compiler in Java-TX“

Dazu wird der `rm` Befehl mit dem `-r` Parameter verwendet, um rekursiv alle Dateien und Ordner zu löschen. Der `*` Operator steht für alle Dateien und Ordner im Zielordner. Die Variable `RM` ist eine vordefinierte Variable in `Make`, die den Befehl zum Löschen von Dateien definiert [16][S. 125 ff.]. Die Regel wird ausgeführt, wenn der `make` Befehl mit dem Parameter `clean` aufgerufen wird.

2.2.1 Performanceprobleme

Das Makefile ist funktionstüchtig und kompiliert das Projekt korrekt, allerdings dauert eine komplette Kompilierung des Projekts mehrere Minuten, während die originale Java Version des Compilers nur wenige Sekunden benötigt. Dabei ist zu erwähnen, dass ein Großteil der Performanceeinbußen nicht durch die Typinferenz des „Java-TX Compiler“ entsteht, sondern schon der `javac` Compiler deutlich langsamer als beim ursprünglichen Compiler ist. Dies hängt damit zusammen, dass der `javac` Compiler für jede Datei einzeln aufgerufen wird. Dies resultierte vor allem auf Grund der folgenden beiden Gründe in einer schlechteren Performance:

- Der `javac` Compiler kompiliert standardmäßig alle Abhängigkeiten implizit mit. Da der Compiler aber ohnehin für jede Datei aufgerufen wird, bedeutet das, dass viele Dateien mehrfach kompiliert werden.
- Der Overhead, den Compiler zu starten ist relativ hoch. Das liegt zu großen Teilen wahrscheinlich daran, dass der `javac` Compiler in Java implementiert ist und somit die Java Virtual Machine (JVM) gestartet werden muss. Dadurch ist es sehr ineffizient, den Compiler für jede Datei neu zu starten.

Die implizite Kompilierung kann beim Aufruf von `javac` durch den Parameter `-implicit:none` deaktiviert werden. Dieser sorgt dafür, dass Abhängigkeiten nicht implizit kompiliert werden. In Tabelle 1 sind die Kompilierzeiten des „Java-TX Compiler“ mit `javac` aufgeführt.

	Einzel	Einzel (Nicht Im- plizit)	Gemeinsam
Min	04:52,44	02:23,28	00:02,50
Max	05:10,69	02:25,78	00:02,73
Avg	04:58,83	02:24,15	00:02,57

Table 1: Kompilierzeiten des „Java-TX Compiler“ mit `javac`³

Die erste Spalte beschreibt die Kompilierzeiten, wenn der Compiler für jede Datei einzeln aufgerufen wird und die implizite Kompilierung von Abhängigkeiten aktiviert ist. Die zweite Spalte beschreibt die Kompilierzeiten, wenn der Compiler für jede Datei einzeln aufgerufen wird, aber die Abhängigkeiten nicht implizit kompiliert werden. Es werden also mehrfache Kompilierungen vermieden. Die dritte Spalte beschreibt die Kompilierzeiten, wenn der Compiler für alle Dateien auf einmal aufgerufen wird. Zu sehen ist, dass sich die Kompilierzeit in etwa halbiert, wenn die Abhängigkeiten nicht implizit kompiliert werden. Allerdings ist die letzte Variante, die alle Dateien auf einmal an den Compiler übergibt, immer noch etwa um den Faktor 56 schneller.

Aus diesem Grund wurde entschieden, dass die Kompilierung des „Java-TX Compiler“ mit `Make` nicht praktikabel ist. Ein alternativer Ansatz, der dieses Problem umgeht, wird in Unterabschnitt 2.3 vorgestellt.

2.3 Kompilierung mit Bash

Aufgrund der Laufzeiteinbußen, die beim Verwenden des Makefiles zustande kommen, musste eine alternative Lösung gefunden werden. Die Idee ist, zuerst alle Dateien, die kompiliert werden müssen, zu sammeln und dann den Compiler nur einmal aufzurufen. Da sich die Komplexität für solch ein Skript in Grenzen hält und die größtmögliche Flexibilität bietet, haben wir uns für ein eigenes Skript entschieden. Als Skriptsprache wurde Bourne Again Shell (Bash) ausgewählt. Diese ist sowohl unter Linux, als auch unter MacOS nativ ohne weitere Abhängigkeiten lauffähig. Zur Ausführung auf Windows müsste man auf das Windows Subsystem for Linux (WSL) zurückgreifen. Das Skript hat das gleiche Verhalten wie das Makefile. Der einzige Unterschied besteht darin, dass alle Dateien die kompiliert werden müssen, in eine Liste gespeichert werden und der Compiler am Ende nur einmal aufgerufen wird. Dadurch wird der Overhead des Compilers minimiert und die Kompilierzeit deutlich reduziert, sodass man ohne Java-TX Dateien mit dem Skript eine Zeit von etwa 2 Sekunden erreicht, was in etwa der Zeit in Tabelle 1 entspricht. Wenn das Skript mit dem Argument `clean` aufgerufen wird, wird der Inhalt des Zielordners gelöscht.

In Abbildung 4 ist die Ordnerstruktur mit allen notwendigen Ressourcen des Projekts

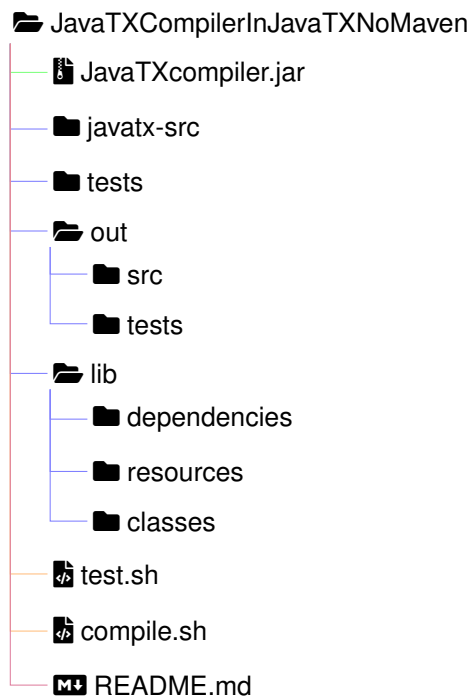


Figure 4: Dateistruktur des Projekts

dargestellt. Das Projekt besteht aus folgenden Ordnern und Dateien:

- **JavaTXCompiler.jar**: Die aktuelle Version des „Java-TX Compiler“ in Form eines JAR-Archivs. Es wird verwendet, um die Java-TX Dateien des „Java-TX Compiler in Java-TX“ zu kompilieren.
- **javatx-src**: Enthält den Code des „Java-TX Compiler in Java-TX“. Besteht zum Großteil aus den Java Quelldateien des „Java-TX Compiler“ und einigen Java-TX Dateien, die bereits migriert werden konnten.
- **tests**: Enthält die Testsuite des „Java-TX Compiler“ in Form von JUnit Tests (vgl. Unterabschnitt 2.4).
- **out**: Enthält das kompilierte Projekt sowie die kompilierten Tests.
 - **src**: Enthält das kompilierte Projekt in Form von `.class` Dateien.
 - **test**: Enthält die kompilierten Tests in Form von `.class` Dateien.
- **lib**: Enthält zusätzliche statische Dateien, die für das Projekt benötigt werden.
 - **dependencies**: Enthält externe Bibliotheken, die für das Projekt benötigt werden in Form von JAR-Archiven. Werden

im „Java-TX Compiler“ durch Maven verwaltet.

- **resources**: Enthält zusätzliche Ressourcen, die für die Tests benötigt werden. Im Wesentlichen sind dies die Beispielprogramme, die getestet werden.
- **classes**: Enthält den gesamten „Java-TX Compiler“ in kompilierter Form, um Abhängigkeiten zwischen `jav` und `java` Dateien zu ermöglichen (vgl. Unterabschnitt 2.1)
- **test.sh**: Dieses Bash Skript ist für die Kompilierung und Ausführung der Tests zuständig.
- **compile.sh**: Dieses Bash Skript ist für die Kompilierung des Projekts zuständig.
- **README.md**: Eine Markdown Datei, die Informationen zur Verwendung des Projekts enthält.

Da das gesamte Skript weder besonders spannend, noch komplex und zusätzlich relativ lang ist, wird es hier nicht im Detail beschrieben.

2.4 Tests

Da es bei der Entwicklung eines eigenen Compilers schnell zu fehlerhaftem Bytecode kommen kann, welcher ggf. erst zur Laufzeit auffällt, ist es sinnvoll den generierten Bytecode zu testen. Der große Vorteil ist an dieser Stelle, dass es bereits eine Testsuite mit über 200 Unittests für den „Java-TX Compiler“ gibt. Diese Tests müssen lediglich auf den generierten Bytecode des „Java-TX Compiler in Java-TX“ angewandt werden, um fehlerhafter Bytecode oft frühzeitig zu erkennen.

Die Tests des „Java-TX Compiler“ werden über Maven verwaltet und ausgeführt. Außerdem bieten die meisten Integrated Development Environment (IDE)s eine direkte Integration für JUnit Tests. Diese automatischen Verfahren sind für reine Java Projekte geeignet, da sie die Testsuite automatisch ausführen und die Ergebnisse anzeigen. Da der „Java-TX Compiler in Java-TX“ aus einer gemischten Codebasis mit Java und Java-TX Quelldateien besteht, die mit dem eigenen Skript kompiliert wird, ist es nicht möglich diese Tools zu verwenden.

Daher muss die Testsuite manuell auf die kompilierten Dateien angewandt werden. Um diesen Prozess zu automatisieren, wurde ein weiteres Skript geschrieben. Dieses Skript kompiliert und kopiert die notwendigen Dateien zur Ausführung der Tests in den korrekten Ordner. Danach können die Tests mit einem Aufruf des JUnit 4 JAR-Archivs ausgeführt werden. Das gesamte Skript ist in Listing 9 zu sehen.

Zu Beginn werden einige Variablen initialisiert. `DESTDIR` gibt den Ordner an, an dem die bereits kompilierten Dateien des „Java-TX Compiler in Java-TX“ liegen. Auf diesen Dateien werden die Tests ausgeführt. `TESTDESTDIR` gibt den Ordner an, in dem die kompilierten JUnit Testdateien abgelegt werden sollen. `DEPENDENCIES` gibt den Ordner mit den externen Abhängigkeiten des Projekts an. Diese sind identisch mit den Abhängigkeiten des `compile.sh` Skripts und damit mit den Dateien, die im ursprünglichen Compiler über Maven verwaltet werden. `TESTFILES` gibt die Testklassen an, die ausgeführt werden sollen. Zuletzt verweist `RESOURCES` auf den Ordner, in dem die Testdateien, d.h. `.jav` Quelldateien, die von den Tests kompiliert werden, liegen. Dann werden alle JUnit Test Dateien kompiliert, wenn sie nicht bereits vorhanden sind. Im Anschluss werden sämtliche Ressourcen, die von den Tests benötigt werden, z.B. die Testdateien, deren Code kompiliert werden soll, an die korrekte Position kopiert, sodass die Tests diese zur Laufzeit finden können. In den nächsten Zeilen wird in den Testordner gewechselt und die Tests mit der Klasse `JUnitCore` ausgeführt. Die Klasse `org.junit.runner.JUnitCore` ist die Hauptklasse von JUnit 4, die die Tests ausführt. Sie ist im JUnit4 JAR-Archiv enthalten, welches bereits bei den Abhängigkeiten in `DEPENDENCIES` vorhanden ist.

3 Aufgetretene Probleme

Beim Versuch, im Zuge dieser Studienarbeit Teile des „Java-TX Compiler“ in Java-TX zu konvertieren, sind viele Bugs und fehlende Features aufgefallen. Einige interessante Probleme werden in diesem Kapitel genauer beschrieben. In Abbildung 5 ist eine Gesamtübersicht über die Anzahl der gefundenen Probleme zu sehen. Die Grafik ist in zwei Kategorien unterteilt: Bugs und Feature-Anfragen. Bugs beschreiben hierbei Fehler in bereits implementierten Funktionen,

während Feature-Anfragen gänzlich neue Funktionen beschreiben, meistens handelt es sich hierbei um Standard-Java-Funktionen, die bislang noch nicht implementiert wurden und keine neuen Funktionen im Bezug auf Java-TX Sprachfeatures. Weiter sind die Kategorien nach offenen und geschlossenen Problemen unterteilt.

3.1 Neue Funktionen

Um den „Java-TX Compiler in Java-TX“ umzuschreiben, sind einige Funktionen notwendig, die zu Beginn der Studienarbeit noch nicht implementiert waren. Über den Zeitraum der Studienarbeit wurden insgesamt 20 neue Funktionen implementiert. Da die For-Each Schleife in Java-TX minimal von der Java Syntax abweichen darf, wird sie im folgenden etwas ausführlicher beschrieben. Einige der anderen Funktionen werden nur kurz aufgeführt.

3.1.1 For-Each Schleife

Die For-Each Schleife ermöglicht es, mit einer eleganten Syntax über eine Variable vom Typ `java.lang.Iterable` zu iterieren [19]. Ein Beispiel zur Verwendung der For-Each Schleife in Java ist in Listing 10 zu sehen.

```

1 List<String> list = new ArrayList
   <>();
2 //entweder
3 for (String s : list) {
4     System.out.println(s);
5 }
6 // ... oder
7 for(var s : list) {
8     System.out.println(s);
9 }

```

Listing 10: For-Each Schleife in Java

Dabei kann der Typ der Variable entweder explizit angegeben oder durch Angabe des Schlüsselworts `var` durch den Compiler inferiert werden. In Java-TX kann diese Typangabe Angabe auch gänzlich weggelassen werden. Ein Beispiel zur Verwendung der For-Each Schleife in Java-TX ist in Listing 11 zu sehen.

```

1 #!/bin/bash
2 ##TEST ENVIRONMENT##
3
4 DESTDIR="out/src"
5 TESTDESTDIR="out/tests"
6 DEPENDENCIES="dependencies/*"
7 TESTFILES="TestComplete TestPackages GenericParserTest TestTypeDeployment
   finiteClosure.SuperInterfacesTest astfactory.ASTFactoryTest targetast.
   ASTToTypedTargetAST targetast.GreaterEqualTest targetast.GreaterThanTest
   targetast.InheritTest2 targetast.InheritTest targetast.LessEqualTest
   targetast.LessThanTest targetast.OLTest targetast.PostIncTest targetast.
   PreIncTest targetast.PutTest targetast.TestCodegen targetast.TestGenerics
   targetast.TphTest targetast.WhileTest"
8 RESOURCES="lib/resources"
9
10 #recompile all necessary test files
11 javac -cp "$TESTDESTDIR:$DESTDIR:$DEPENDENCIES" -d $TESTDESTDIR tests/**/*.java
12 javac -cp "$TESTDESTDIR:$DESTDIR:$DEPENDENCIES" -d $TESTDESTDIR tests/*.java
13
14 cp -r $RESOURCES $TESTDESTDIR/resources/
15
16 cd "$TESTDESTDIR"
17
18 #run tests with junit
19 java -cp "../src:../../dependencies/*" org.junit.runner.JUnitCore $TESTFILES

```

Listing 9: Skript zum Kompilieren und Ausführen der Tests

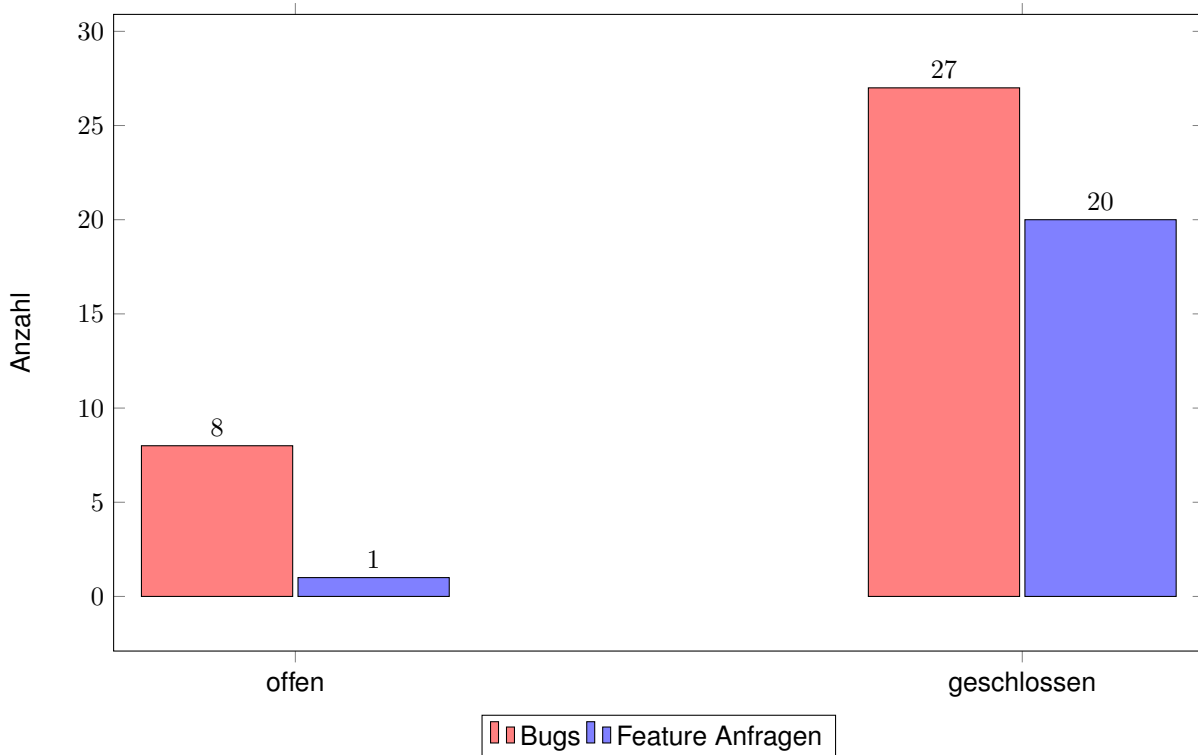


Figure 5: Gefundene Probleme und neue Funktionen

```

1 List<String> list = new ArrayList
   <>();
2 //Kein Typ vor der Variable s
   notwendig
3 for (s : list) {
4     System.out.println(s);
5 }

```

Listing 11: For-Each Schleife in Java-TX

3.1.2 Weitere neue Funktionen

Neben der For-Each Schleife wurden noch weitere Funktionen implementiert. Diese unterscheiden sich allerdings nicht von der Java Syntax und werden daher nur kurz aufgelistet:

- `super()` Konstruktoraufruf
- Methodenaufruf der Superklasse mittels `super`
- `this()` Konstruktoraufruf
- `instanceof` Schlüsselwort
- `throw` Schlüsselwort
- Verwendung des `null` Literals
- Character Literal z.B. `'a'`
- Long und Float Literals z.B. `1L` oder `1.0f`
- Type Casts
- Negation Operator `!`
- Bitweise AND `&` und OR `|` Operatoren
- Modulo Operator `%`
- Ternary Operator `? :`
- Do-While Schleife
- Verwenden von JAR-Archiven im Classpath

3.2 Bugs

Insgesamt wurden 27 Bugs gefunden und behoben, einige weitere sind noch offen. Im folgenden Abschnitt wird eine kleine Auswahl dieser Bugs ausführlicher beschrieben.

3.2.1 JVM Classpath wird von „Java-TX Compiler“ beachtet

Zur Beschreibung dieses Bugs müssen zunächst einige Grundlagen zum Classpath und dem Classloader geklärt werden. Damit der „Java-TX Compiler“ andere Klassen verwenden kann (darunter zählen z.B. auch JAR-Archive), müssen sie im Classpath vorhanden oder Teil der Standardbibliothek sein. Der Classpath kann, wie auch beim `javac` Compiler, mit dem Flag `-cp` oder `-classpath` angegeben werden. Ein Beispielaufruf könnte folgendermaßen aussehen: `java -jar JavaTXCompiler.jar -cp /path/to/classes /path/to/source.jav`⁴. Wenn kein Classpath angegeben wird, wird das aktuelle Arbeitsverzeichnis des Nutzers verwendet. Die angegebenen Pfade werden dann vom Compiler durchsucht, um die benötigten Klassen zu finden. Wenn die Klasse nicht gefunden werden konnte, wird ein Fehler ausgegeben. Nur Klassen, die im Classpath vorhanden sind, können importiert und verwendet werden. Ausnahme sind die Klassen aus der Java Standard Library, wie z.B. alle Klassen des Pakets `java.lang`, die standardmäßig vorhanden sind. Das Paket `java.lang` bietet die Klasse `ClassLoader`, die es ermöglicht, Klassen dynamisch in die JVM zu laden [1]. Der „Java-TX Compiler“ verwendet auch diesen Classloader, um die angegebenen Klassen zu suchen und einzulesen. So muss kein eigener Parser für Java Bytecode implementiert werden.

Um genau zu sein, gibt es nicht einen Classloader, sondern mehrere, die nacheinander ausgeführt werden. Die oberste Schicht ist der Bootstrap Classloader, der Java Development Kit (JDK) interne Klassen lädt. Darunter befindet sich der Extension Classloader, der die Klassen aus den JAR-Dateien im `jre/lib/ext` Verzeichnis lädt. Der Application Classloader lädt die Klassen aus dem Classpath und zuletzt gibt es eine eigene Implementierung des `java.net.URLClassLoader` namens Directory Classloader, der die Pfade, die dem Compiler mit `-cp` übergeben werden, durchsucht [1, 8]. Eine Visualisierung dieser Schichten ist in Abbildung 6 zu sehen. Das Programm beginnt mit dem untersten Classloader (Directory Classloader), versucht also die gesuchte Klasse in diesen Ressourcen zu finden. Wenn die Klasse nicht gefunden wird, wird der nächste Classloader (Application Classloader) verwendet. Dieser Prozess wird so lange wiederholt,

⁴ `JavaTXCompiler.jar` ist in diesem Fall der „Java-TX Compiler“ in Form eines JAR-Archivs

bis die Klasse gefunden wurde oder alle Classloader durchsucht wurden. Zu beachten ist, dass der Classpath des Application Classloaders nicht derselbe ist wie der Classpath, der dem Compiler übergeben wird und vom DirectoryClassloader verwendet wird. Der Classpath des Application Classloaders ist der Classpath, mit dem die JVM gestartet wurde. Dieser beinhaltet in diesem Fall z.B. sämtliche Maven-Abhängigkeiten und Bytecode-Dateien des „Java-TX Compiler“.

Da durch dieses Vorgehen auch die Klassen aus dem JVM Classpath vom Classloader und damit vom „Java-TX Compiler“ berücksichtigt werden, kann man diese importieren, ohne sie im Classpath angeben zu müssen. Dies ist ein unerwünschtes Verhalten, da der Compiler so Klassen akzeptiert, die gegebenenfalls nicht vom Programmierer gewünscht sind, was zu Verwirrung führen kann. Ein Beispiel hierzu ist in Listing 12 zu sehen. Dieser Code kompiliert mit dem Befehl `java -jar JavaTXCompiler.jar Main.jav`⁵ korrekt, obwohl die Klasse `com.google.common.math.IntMath` weder im Classpath angegeben noch in der Standardbibliothek vorhanden ist. Sie ist jedoch im JAR Archiv des „Java-TX Compiler“ und damit im JVM Classpath zur Ausführung des Compilers vorhanden, da der „Java-TX Compiler“ die Google Guava Bibliothek verwendet.

```

1 import com.google.common.math.IntMath;
2 import java.lang.String;
3
4 class Main{
5     return2(){
6         return IntMath.checkedAdd(1,
7             1);
8     }
9 }

```

Listing 12: Verwenden von Klassen im JVM Classpath

Dieses Verhalten ist vor allem auch für das Projekt „Java-TX Compiler in Java-TX“ problematisch, da der „Java-TX Compiler“ und der „Java-TX Compiler in Java-TX“ die gleichen Klassen/Klassenhierarchie verwenden und somit die Möglichkeit besteht, dass der Compiler die Klassen des „Java-TX Compiler“ verwendet anstatt die des „Java-TX Compiler in Java-TX“.

Die Lösung dieses Problems ist glücklicherweise unkompliziert. Es muss lediglich die ClassLoader Hierarchie so angepasst werden, dass der Application Classloader übersprungen wird. Diese Änderung ist in Abbildung 7 zu sehen. Der Directory

⁵ JavaTXCompiler.jar ist hier der „Java-TX Compiler“ in Form eines JAR Archivs

ClassLoader ruft nun direkt den Extension Classloader auf. So können weiterhin Klassen aus der Java Standard Bibliothek verwendet werden, jedoch nicht mehr die Klassen im JVM Classpath.

3.2.2 Kompatibilität von Java-TX Funktionstypen und funktionalen Interfaces

Als Problem stellte sich die Kompatibilität von Java-TX Funktionstypen und funktionalen Interfaces heraus, die seit Java 8 als Zieltypen für Lambda Ausdrücken dienen. Ziel ist es, bestehende Java Bibliotheken, die mit funktionalen Interfaces arbeiten, mit Java-TX Funktionstypen verwenden zu können. Die theoretische Lösung für dieses Problem wurde bereits 2017 in [11][Abschnitt 6] beschrieben. Die praktische Umsetzung gestaltet sich jedoch komplizierter als gedacht. Ein Beispiel für eine Bibliothek, die ausgiebig funktionale Interfaces verwendet, ist die sehr verbreitete Stream API, welche mit Java 8 eingeführt wurde [2]. Streams erlauben es, Daten mit deklarativem Code zu verarbeiten, was die Lesbarkeit und Wartbarkeit des Codes erhöht [14].

```

1 import java.util.List;
2 import java.util.stream.Stream;
3 import java.util.function.Predicate;
4
5 public class ListUtils{
6     static List<Integer>
7     getAllEvenNumbers(List<Integer> list)
8     {
9         List<Integer> result =
10             list.stream()
11                 .filter(x -> x % 2 == 0)
12                 .toList();
13         return result;
14     }
15 }

```

Listing 13: Verwendung der Stream API in Java

In Listing 13 ist ein Beispielprogramm in Java zu sehen, welches die Stream API verwendet. Die Methode `getAllEvenNumbers` filtert alle ungeraden Zahlen aus einer Liste. Dazu verwendet sie die Lambda Funktion `x -> x % 2 == 0`, die folgendermaßen definiert ist:

$$\text{isEven}(n)^6 = \begin{cases} true, & \text{wenn } n \bmod 2 = 0 \\ false, & \text{sonst} \end{cases}$$

⁶ Im Quellcode hat der Lambda-Ausdruck keinen Namen

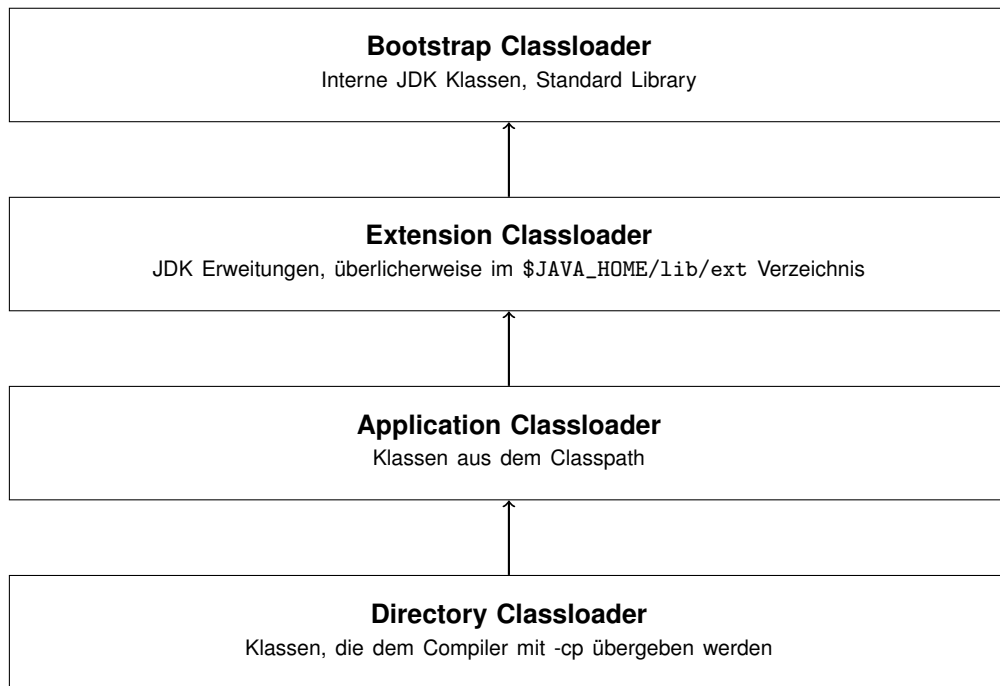


Figure 6: Die Classloader Hierarchie des „Java-TX Compiler“ (vgl. [8])



Figure 7: Die Classloader Hierarchie des „Java-TX Compiler“ ohne den ApplicationClassLoader

Die `filter` Methode von `java.util.stream.Stream` lässt nur Werte passieren, für die die angegebene Funktion zu „true“ evaluiert. Zuletzt werden diese Werte in einer Liste gesammelt und zurückgegeben.

Der `javac` Compiler inferiert für den Lambda-Ausdruck `x -> x % 2 == 0` den Typ `java.util.function.Predicate<Integer>`, da explizit dieser Typ von der `filter` Methode erwartet wird.

```
1 public interface Stream<T>
2     extends BaseStream<T, Stream<T>> {
3     Stream<T>
4         filter(Predicate<? super T>
5             predicate);
6     ...
7 }
```

In Java-TX kompilierte dieser Code zu Beginn der Studienarbeit nicht, da der „Java-TX Compiler“ den Typ `Fun1$$<Integer, Boolean>` für den Lambda Ausdruck inferiert hat. Der „Java-TX Compiler“ würde also den Lambda Ausdruck als Funktionstypen interpretieren anstatt als funktionalen Interface, was soweit korrekt ist. Obwohl die beiden Typen semantisch äquivalent sind, sind sie aufgrund des nominalen Typsystems von Java und der JVM jedoch nicht austauschbar. Es kommt also zu einem Laufzeitfehler. Dieses Problem ist von hoher Relevanz, da sämtliche Methoden aus Java Bibliotheken, die funktionalen Interfaces verwenden, so in Java-TX nicht verwendet werden können.

Im Laufe der Arbeit wurde zumindest eine Teillösung dieses Problems implementiert, so dass der Code in Listing 13 kompiliert und lauffähig ist. Der „Java-TX Compiler“ generiert in solch einem Fall nun den Code für das korrekte funktionale Interface statt dem `FunN$$` Typen. Das funktioniert allerdings nur, wenn der Lambda Ausdruck wie bei Listing 13 direkt im Funktionsaufruf steht. Der Beispielcode in Listing 14 funktioniert also nicht und stellt aktuell noch ein Problem dar.

```
1 import java.util.List;
2 import java.lang.Integer;
3 import java.lang.Boolean;
4 import java.util.stream.Stream;
5 import java.util.function.Predicate;
6
7 public class ListUtils{
8     static getAllEvenNumbers(list){
9         // Java-TX inferiert hier
10            Fun1$$<Integer, Boolean>
11            var func = x -> x % 2 == 0;
```

```
11 // An dieser Stelle wird aber
12 // ein Predicate<Integer>
13 // erwartet -> Laufzeitfehler
14 var result = list.stream().
15     filter(func).toList();
16 }
17 }
```

Listing 14: Aktuell nicht lauffähiger Java-TX Code I

Das liegt daran, dass der Compiler zum Zeitpunkt der Initialisierung des Lambda Ausdrucks nicht weiß, in welchem Kontext der Ausdruck später verwendet werden soll. Betrachten wir zur Verdeutlichung des Problems die etwas komplexere Funktion `uselessFunction` in Listing 15. Über die Sinnhaftigkeit dieses Codes lässt sich streiten. Die Funktion sollte im Endeffekt das gleiche Resultat liefern wie die Funktion in Listing 13, mit dem Unterschied, dass in der Resultatliste für jede Zahl der Wert `true` zurückgegeben wird. Das Problem ist nun, dass die selbe Lambda Funktion im Laufe der Funktion mit verschiedenen funktionalen Interfaces verwendet wird, was die Sache verkompliziert. So würde die `map` Methode in Zeile 10 den Typ `Function<Integer, Boolean>` erwarten, während die `Filter` Methode in Zeile 9 weiterhin `Predicate<Integer>` erwartet. Hier müsste vermutlich für alle Aufrufe eine separate Lambda Funktion mit dem richtigen Target Type im Bytecode erstellt werden. Allerdings erfordert dies vermutlich eine größere Änderung am Compiler und konnte aktuell noch nicht umgesetzt werden.

```
1 import java.util.List;
2 import java.lang.Integer;
3 import java.lang.Boolean;
4 import java.util.stream.Stream;
5 import java.util.function.Predicate;
6 import java.util.function.Function;
7
8 public class ListUtils{
9     static uselessFunction(list){
10        // Java-TX inferiert hier
11        Fun1$$<Integer, Boolean>
12        var func = x -> x % 2 == 0;
13        // An dieser Stelle wird ein
14        Predicate<Integer>
15        erwartet
16        var result1 = list.stream().
17            filter(func);
18        // An dieser Stelle wird ein
19        Function<Integer, Boolean>
20        erwartet
21        var result2 = result1.map(func)
22            .toList();
23        return result2;
24    }
25 }
```

18 }
}**Listing 15:** Aktuell nicht lauffähiger Java-TX Code II

3.2.3 Überschreiben von Methoden mit primitiven Datentypen

Ein weiteres Problem ist das Überschreiben von Methoden mit primitiven Datentypen. Im Vergleich zu Java unterstützt Java-TX nur Referenztypen, keine primitiven Datentypen. Die Grammatik erlaubt zwar die Verwendung von primitiven Datentypen, diese werden jedoch intern in Referenztypen umgewandelt. In Abbildung 8 sind auf der linken Seite einige Initialisierungen von primitiven Datentypen in Java-TX gezeigt. Auf der rechten Seite gegenübergestellt ist der Code, wie er im Compiler intern verarbeitet und später auch im Bytecode generiert wird. Der Compiler generiert also für jeden primitiven Datentypen den entsprechenden Wrappertyp. Daher ist es auch bei der Verwendung von primitiven Typen notwendig, die entsprechenden Wrapperklasse zu importieren.

```

1 import java.lang.Integer;
2 import java.lang.Boolean;
3 import java.lang.Float;
4 class PrimitiveTypes{
5     int i = 5;
6     boolean b = true;
7     float f = 10.5f;
8 }

```

```

1 import java.lang.Integer;
2 import java.lang.Boolean;
3 import java.lang.Float;
4 class PrimitiveTypes{
5     Integer i = 5;
6     Boolean b = true;
7     Float f = 10.5f;
8 }

```

Figure 8: Primitive Datentypen in Java-TX

Diese Eigenschaft von Java-TX führte im Zusammenhang mit der Überladung von Java Methoden, deren Rückgabewert oder Parameter primitive Datentypen sind, zu einem Problem. Nehmen wir als Beispiel den Code in Listing 16. Die Methode `hashCode` wird von der Mutterklasse `Object`⁷ geerbt. Sie hat folgende Signatur:

```
int hashCode();
```

⁷ In Java erben alle Klassen implizit von `Object`

Da sowohl der Name als auch die Parameterliste der Methode übereinstimmen, würde man erwarten, dass diese von der Klasse `Foo` überschrieben wird.

```

1 import java.lang.Integer;
2
3 public class Foo{
4     public hashCode(){
5         return 42;
6     }
7 }

```

Listing 16: Überschreiben von Methoden mit primitiven Datentypen in Java-TX

Stattdessen inferiert der Compiler allerdings die Typen in Listing 17 für die Methode `hashCode` in Listing 16. Dies ist aufgrund der Tatsache, dass primitive Datentypen in Java-TX automatisch mit dem Wrappertyp ersetzt werden, logisch.

```

1 import java.lang.Integer;
2
3 public class Foo{
4     public java.lang.Integer hashCode
5     (){
6         return 42;
7     }
8 }

```

Listing 17: Ergebnis der Typinferenz für die Methode `hashCode` in Java-TX

Im Bytecode führt dies allerdings anstatt einer Überschreibung zu einer Überladung der Methode `hashCode`, da der Rückgabotyp in Listing 17 nicht mit dem Rückgabotyp in Listing 16 übereinstimmt, schließlich sind `int` und `java.lang.Integer` trotz Autoboxing unterschiedliche Typen (mehr dazu in [7][S.6 ff]). Denn obwohl Java die Überladung von Methoden anhand des Rückgabetyps nicht unterstützt, ist es in Java Bytecode durchaus möglich. Dies macht sich Java z.B. für das kovariante Überladen von Methoden zunutze. Seit Java 5 ist es möglich, dass eine Methode in einer Subklasse einen Rückgabotyp hat, der ein Subtyp des Rückgabetyps der Methode in der Superklasse ist [7][S. 49]. Dazu sei zunächst die Signatur der Methode `clone` in der Klasse `Object` gegeben, welche kein Parameter hat und ein Objekt vom Typ `Object` zurückgibt:

```

class Object{
    ...
    protected Object clone(){...}
}

```

Es ist nun möglich, die Methode `clone` in einer Subklasse zu überschreiben und den Rückgabebetyp zu spezialisieren. In Listing 18 wird die Methode `clone` in der Klasse `A` überschrieben und der Rückgabebetyp auf `A` spezialisiert.

```

1 class A{
2     ...
3     @Override
4     public A clone(){...}
5 }

```

Listing 18: Kovariante Methodenüberladung in Java

Dies wird durch eine Bridge Methode ermöglicht, die sich den Fakt zunutze macht, dass die JVM Methoden anhand des Rückgabetyps unterscheidet und somit überladen kann. Der Compiler erzeugt also eine Bridge Methode, mit der Signatur `Object clone()`, die die Methode `A clone()` aufruft. In Listing 19 ist dazu der dekomplilierte Bytecode der Klasse `A` gegeben.

```

1 class A{
2     ...
3     public A clone(){...}
4
5     public Object clone(){
6         return this.clone(); //Aufruf
7         der Methode clone():A
8     }
9 }

```

Listing 19: Dekompilierter Bytecode der Klasse `A`

In unserem Bug ist dies allerdings nicht das gewünschte Verhalten, da eine Überladung anhand des Rückgabewerts nicht möglich sein sollte. In Java werden Überladungen anhand des Rückgabewerts vom Compiler abgelehnt, da nicht immer ersichtlich ist, welche Methode aufgerufen werden soll. Außerdem wären Überladungen von Methoden, welche primitive Typen verwenden, in Java-TX gänzlich unmöglich, weil selbst die explizite Angabe des Typs `int` vom Compiler in `java.lang.Integer` umgewandelt werden würde.

Um dieses Problem zu lösen, wird aktuell eine einfache Substitution verwendet. Wenn der Compiler eine Überladung erkennt und der Rückgabebetyp oder ein Parametertyp der Superklasse ein primitiver Datentyp ist, wird der dazugehörige Wrappertyp durch den jeweiligen primitiven Typen ersetzt. Dadurch funktioniert die Überschreibung von Methoden mit primitiven Datentypen korrekt. Da es aber noch einige Bugs mit dieser Implementierung gibt, bleibt abzuwarten, ob dies die endgültige Lösung ist.

3.2.4 Korrekter Methodenaufruf für überladene Methoden mit Subtypen als Parameter

Dieser Bug ist zum Zeitpunkt der Abgabe dieser Arbeit noch nicht behoben. Er tritt vor allem im Zusammenhang mit dem Visitor-Pattern auf, welches im „Java-TX Compiler“ ausgiebig genutzt wird. Der Compiler ruft nicht immer die korrekte Methode auf, wenn mehrere potenziell korrekte Methoden zur Auswahl stehen. Sehen wir uns dazu an, wie Java mit diesem Problem umgeht. Dazu sei der Code in Listing 20 gegeben. Die `Main` Funktion ruft dabei die Methode `visit` der Klasse `Visitor` mit einer Instanz der Klasse `java.lang.Integer` (bzw. `int`, was aber geboxed wird [7][S.6 ff]) auf. Die `visit` Methode ist dreimal überladen, einmal mit `java.lang.Object`, einmal mit `java.lang.Number` und einmal mit einem `java.lang.Integer`. Die Frage ist nun, welche dieser Überladungen aufgerufen werden sollte. Theoretisch wäre jeder Aufruf korrekt, da `java.lang.Integer` sowohl von `java.lang.Number` als auch von `java.lang.Object` erbt. Das Verhalten in so einem Fall ist in [4][Abschnitt 15.12.2.5] beschrieben. Java wählt in einem solchen Fall die spezifischste Methode, also die Methode, die den spezifischsten Typen als Parameter hat. In diesem Fall wäre das die Methode, mit der Signatur `void visit(Integer i)`. Dies bestätigt sich auch, wenn der Code in Listing 20 kompiliert und ausgeführt wird. Die Ausgabe ist wie erwartet **"Integer"**.

```

1 class Main{
2     public static void main(String[]
3         args){
4         Visitor v = new Visitor();
5         v.visit(1);
6     }
7
8     class Visitor{
9         public void visit(Object o){
10            System.out.println("Object");
11        }
12        public void visit(Number n){
13            System.out.println("Number");
14        }
15        public void visit(Integer i){
16            System.out.println("Integer");
17        }
18    }

```

Listing 20: Überladene Methoden in Java

Dieses Verhalten wäre vor allem aus Kompatibilitätsgründen auch in Java-TX wünschenswert.

Hier kommt es aktuell aber zu einem Fehler. Wenn der Code mehrmals kompiliert wird, wählt der Compiler jedes Mal eine andere Methode. Die Ausgabe ist also nicht deterministisch. Dies ist dadurch zu ergründen, dass der Typinferenzalgorithmus alle 3 Lösungen für den Methodenaufruf findet, aktuell aber nicht berücksichtigt, dass die Parameter der Methoden Subtypen sein können. Die Lösungen werden also als gleichwertig angesehen. Der Bytecodegenerator wählt dann aktuell die erste Lösung und verwirft die restlichen. Da der Typinferenzalgorithmus auf mehreren Threads parallel ausgeführt wird, kann sich die Reihenfolge der Ergebnisse ändern und somit auch das Ergebnis des Bytecodegenerators. Daher werden bei mehreren Kompilierungen unterschiedliche Ergebnisse erzielt.

Leider konnte das Problem bisher nicht gelöst werden, da es erst relativ spät bemerkt wurde. In einer späteren Version des Compilers sollte sich Java-TX in diesem Punkt aber wie Java verhalten, so dass auch das Visitor Pattern in Java-TX korrekt funktioniert.

3.2.5 Weitere Bugs und fehlende Features

Neben diesen umfangreich beschriebenen Problemen, gab es noch viele weitere Probleme, die hier nur kurz aufgeführt werden.

1. Die Access Modifier wurden nicht korrekt auf die Methoden angewendet. Alle Methoden wurden mit dem Access Modifier `public` deklariert. Wenn ein anderer Access Modifier verwendet wurde, wurde dieser einfach hinzugefügt. So konnte es zu Signaturen wie `public private void foo()` kommen.
2. Die Fehlermeldungen des Compilers wurden verbessert. Es wird nun angezeigt, in welcher Zeile und Datei ein Constraint erstellt wurde.
3. Überladene Konstruktoren konnten nicht aufgerufen werden.
4. If-Statements ohne Blöcke wurden teilweise nicht korrekt verarbeitet.
5. `toString()` und andere Methoden von `Object` konnten auf Interfaces nicht aufgerufen werden.
6. Die `@Override` Annotation bei Methoden führte zu einem Fehler. Annotations werden nun ignoriert.

7. Die Typinferenz war für die neu hinzugefügte `ForEach` Schleife fehlerhaft.
8. Es gab einige Probleme mit Interfaces, welche behoben wurden.
9. Der Bytecode beim Aufruf von statischen Methoden war fehlerhaft.
10. Bei der Überschreibung von vererbten Methoden mussten die Parametertypen den gleichen Namen haben.

Zusätzlich sind auch einige Features aufgefallen, die bis zum aktuellen Stand noch nicht implementiert wurden.

1. Arrays und damit auch die Main-Funktion sind nicht implementiert.
2. Exceptions sind nur sehr rudimentär implementiert, z.B. sind checked Exceptions aktuell nicht möglich.
3. Subtypisierung bei Funktionstypen funktioniert noch nicht wie es sollte
4. Der Funktionstyp `FunVoidN$$` für Funktionstypen, die `void` zurückgeben ist aktuell noch nicht umgesetzt.

4 Fazit und Ausblick

Im Laufe dieser Studienarbeit konnte bisher nur ein Bruchteil des Quellcodes übersetzt werden. Abbildung 9 zeigt das aktuelle Verhältnis von Java zu Java-TX Quelldateien. Bisher konnten nur 18 von 251 Quelldateien erfolgreich übersetzt werden. „Erfolgreich“ bedeutet in diesem Kontext, dass alle Tests der Testsuite erfolgreich durchlaufen wurden.

Die übersetzten Dateien beschränken sich aktuell auf die Pakete `de.dhbwstuttgart.typeinference` und `de.dhbwstuttgart.syntaxtree`. Der Umfang der übersetzten Dateien ist vergleichsweise gering. Dennoch konnten durch die Studienarbeit viele Bugs im Compiler gefunden und behoben werden, wodurch sich die generelle Qualität des Compilers verbessert hat. Einige neue Features, welche zum Übersetzen der Quelldateien in Java-TX notwendig waren, wurden ebenfalls hinzugefügt. Die größte Schwachstelle des Compilers sind aktuell wohl die Fehlermeldungen. Diese sind oft schwer zu verstehen und einzugrenzen. Hier besteht definitiv noch Verbesserungspotential. Außerdem fehlen auch

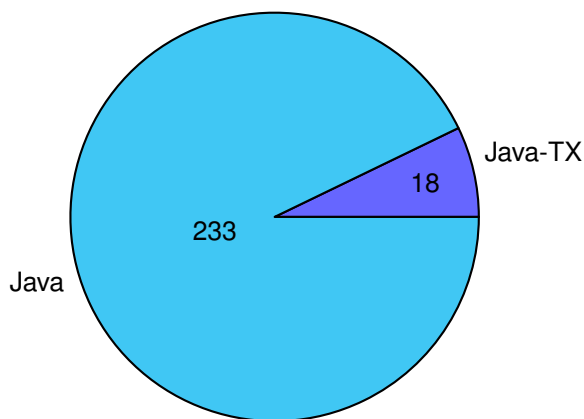


Figure 9: Verhältnis der Java und Java-TX Dateien im „Java-TX Compiler in Java-TX“

aktuell noch einige der grundlegenden Funktionen von Java, wie z.B. die Main Methode, die wegen der fehlenden Unterstützung für Arrays noch nicht implementiert wurde.

Das langfristige Ziel wird sicherlich sein (ggf. in nachfolgenden Studienarbeiten), den gesamten Quellcode des „Java-TX Compiler“ in Java-TX zu übersetzen⁸, um die Qualität und den Funktionsumfang des Compilers sicherzustellen. Der Weg dahin ist jedoch noch weit. Es gibt sicherlich noch viele unentdeckte Bugs und Probleme, die es zu lösen gilt. Zudem müsste man an bestimmten Stellen nicht nur die Typinformationen entfernen, sondern den Code anpassen. Der „Java-TX Compiler“ verwendet, historisch bedingt, teilweise noch alte Java-Features, die in Java-TX nicht mehr unterstützt werden, z.B. Raw Types [13]. Außerdem müsste man zum aktuellen Stand des Compilers sämtliche Arrays in Listen umwandeln, da Java-TX aktuell keine Arrays unterstützt. Dies ist darauf zurückzuführen, dass Arrays auch in Java eine gewisse Sonderstellung haben, da sie aus einer Zeit vor generischen Typen stammen und im Vergleich zu Collections einige Nachteile haben [7][Abschnitt 2.5, 6.9]. Doch auch wenn Arrays im Quellcode durch Listen ersetzt werden können, gibt es gewisse Methoden in Java Bibliotheken, die Arrays verwenden. So z.B. auch die Methode `split` der Klasse `java.lang.String`, die ein Array von Strings zurückgibt [17]:

```
public final class String implements ... {
    ...
    public String[] split(String regex){...}
    public String[] split(String regex,
                          int limit){...}
    ...
}
```

Diese Methoden sind z.B. aus Java-TX nicht aufrufbar, da sie Arrays zurückgeben. Langfristig wird es also notwendig sein, Arrays in Java-TX zu unterstützen, um die Kompatibilität mit Java Klassen zu gewährleisten. Arrays sind in Java auch essenziell für die Main Funktion, die daher in Java-TX auch noch fehlt.

Erstrebenswert wäre es auch, die Fehlermeldungen des Compilers weiter zu verbessern. Dies würde sicherlich auch die Fehlersuche beim Übersetzen der Quelldateien erleichtern. Auch eine vollständige Kompatibilität von Java-TX Funktionstypen und funktionalen Interfaces wäre wünschenswert, um den Vorteil der echten Funktionstypen vollumfänglich mit bestehenden Java Bibliotheken ausnutzen zu können.

Java-TX	Java-Type eXtended
GNU	GNU's Not Unix
JVM	Java Virtual Machine
JDK	Java Development Kit
WSL	Windows Subsystem for Linux
IDE	Integrated Development Environment
GCC	GNU Compiler Collection
Bash	Bourne Again Shell

References

- [1] Baeldung. *Class Loaders in Java | Baeldung*. May 11, 2024. URL: <https://www.baeldung.com/java-classloaders> (visited on 05/25/2024).
- [2] Baeldung. *The Java 8 Stream API Tutorial | Baeldung*. June 15, 2016. URL: <https://www.baeldung.com/java-8-streams> (visited on 05/25/2024).
- [3] *gcc-in-cxx - GCC Wiki*. URL: <https://gcc.gnu.org/wiki/gcc-in-cxx> (visited on 06/01/2024).
- [4] James Gosling et al. *The Java Language Specification, 3rd Edition*. 3rd ed. Upper Saddle River, NJ: Addison Wesley, June 14, 2005. 684 pp. ISBN: 978-0-321-24678-3.

⁸ Externe Tools wie Antlr oder ASM, die zur Implementierung des Compilers verwendet wurden, werden natürlich weiterhin Java Code verwenden.

-
- [5] Simon Marlow et al. “Haskell 2010 language report”. In: *Available online* [\(http://www.haskell.org/\(May 2011\)\)](http://www.haskell.org/(May 2011)) (2010).
- [6] *Maven – Introduction*. URL: <https://maven.apache.org/what-is-maven.html> (visited on 05/20/2024).
- [7] Maurice Naftalin and Philip Wadler. *Java generics and collections*. OCLC: ocm76810468. Beijing ; Sebastopol, CA: O’Reilly, 2007. 273 pp. ISBN: 978-0-596-52775-4.
- [8] Rafael del Nero. *All about Java class loaders*. InfoWorld. June 29, 2023. URL: <https://www.infoworld.com/article/3700054/all-about-java-class-loaders.html> (visited on 05/25/2024).
- [9] *OpenJDK Repository - Javac*. GitHub. URL: <https://github.com/openjdk/jdk/blob/master/src/jdk.compiler/share/classes/com/sun/tools/javac> (visited on 06/01/2024).
- [10] Benjamin C. Pierce. *Types and programming languages*. Cambridge, Massachusetts London, England: The MIT Press, 2002. 623 pp. ISBN: 978-0-262-16209-8.
- [11] Martin Plümicke and Andreas Stadelmeier. “Introducing Scala-like function types into Java-TX”. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. New York, NY, USA: Association for Computing Machinery, Sept. 27, 2017, pp. 23–34. ISBN: 978-1-4503-5340-3. URL: <https://doi.org/10.1145/3132190.3132203> (visited on 04/24/2024).
- [12] Martin Plümicke and Etienne Zink. *Java-TX: The language*. Fakultät Technik der Dualen Hochschule Baden-Württemberg Stuttgart, Jan. 2022. URL: <https://www.dhbw-stuttgart.de/forschung-transfer/technik/schriftenreihe-insights/>.
- [13] *Raw Types (The Java™ Tutorials > Learning the Java Language > Generics (Updated))*. URL: <https://docs.oracle.com/javase/tutorial/java/generics/rawTypes.html> (visited on 06/02/2024).
- [14] SoftwareAlchemy. *Streamline Your Java Code: A Cheat Sheet for Mastering Streams*. Javarevisited. Apr. 23, 2024. URL: <https://medium.com/javarevisited/streamline-your-java-code-a-cheat-sheet-for-mastering-streams-e8500f4495fe> (visited on 05/25/2024).
- [15] Richard Stallman and GCC Developer Community. *Using the GNU Compiler Collection For gcc version 14.1.0*. 2024. URL: <https://gcc.gnu.org/onlinedocs/gcc-14.1.0/gcc.pdf>.
- [16] Richard Stallman, Roland McGrath, and Paul D. Smith. *GNU Make: a program for directing recompilation ; GNU make version 3.81*. Boston, Mass: Free Software Foundation, 2004. 184 pp. ISBN: 978-1-882114-83-2.
- [17] *String (Java Platform SE 8)*. URL: <https://docs.oracle.com/javase/8/docs/api/java/lang/String.html> (visited on 06/02/2024).
- [18] Patrick D. Terry. *Compilers and compiler generators: an introduction with C++*. London: International Thomson Computer Press, 1997. 579 pp. ISBN: 978-1-85032-298-6.
- [19] *The For-Each Loop*. URL: <https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html> (visited on 05/23/2024).
- [20] Simon Thompson. *Haskell: the craft of functional programming*. 3rd ed. Harlow, England ; New York: Addison Wesley, 2011. 585 pp. ISBN: 978-0-201-88295-7.
- [21] David Vandevoorde and Nicolai M. Josuttis. *C++ templates: the complete guide*. Boston, Mass.: Addison-Wesley, 2010. 528 pp. ISBN: 978-0-201-73484-2.

Global Type Inference for Featherweight Java with Wildcards

Andreas Stadelmeier, Martin Plümicke
Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
Department of Computer Science
Florianstraße 15, 72160 Horb
a.stadelmeier@hb.dhbw-stuttgart.de

Abstract

Type inference is a hallmark of functional programming. Its use in object-oriented programming is often restricted to type inference for local variables and the instantiation of generic type parameters. Global type inference for Featherweight Generic Java is a whole-program analysis that infers omitted method signatures. Compared to previous work, its results are more general as it infers types with wildcards. Global type inference is proved sound.

1 Introduction

Type inference is a hallmark of functional programming, where it improves readability and conciseness while enabling safe rapid prototyping by allowing the compiler to deduce types automatically without explicit type annotations. It allows developers to safely refactor their programs before fixing function signatures. In object-oriented programming (OOP), however, the use of type inference is more restricted. It is often employed for local variables and the instantiation of generic type parameters, offering similar readability and conciseness benefits as in functional programming. In languages like Java, the overall architecture of method signatures must be designed by the programmer up-front.

Java features a limited form of local type inference for local variables defined with the `var` keyword, lambda expressions, and method calls. Java infers the type of variables defined with `var`. The rationale here is that instantiations of generic types (e.g., maps of maps) are often unwieldy and they can be easily inferred from the initializing expression. The type of a lambda expression is inferred from the target type, i.e., the type of the method argument that accepts the lambda. The rationale is the intended use of lambdas as callback func-

tions for listeners etc. For calls to generic methods, Java infers the most specific instantiation of the generic parameters for each individual call.

The local type inference features of Java add some convenience, but programmers still have to provide method signatures beforehand. So there is scope for a global type inference algorithm that infers method signatures even if no type information (except class headers and types of field variables) is given. We present such a global type inference algorithm for Featherweight Generic Java with wildcards, a Java core calculus with generics and wildcards in the tradition of Featherweight Java [5] and its extensions [2, 21]. Our approach can also be used for regular Java programs, but we limit the formal presentation to this core calculus.

Our algorithm enables programmers to write Java code with only a minimum of type annotations (class headers and types of field variables), it can fill in missing type annotations in partially type-annotated programs, and it can suggest better (more general) types for ordinary, typed Java programs.

Listing 1 shows an example input of a method implementation without any type annotation. Our algorithm infers the type annotations in Listing 2: it

Listing 1: Missing return type

```
genList() {
  if( ... ) {
    return new List(1);
  } else {
    return new List("Str");
  }
}
```

Listing 2: Type inference solution

```
List<?> genList() {
  if( ... ) {
    return new List<Integer>(1);
  } else {
    return new List<String>("Str");
  }
}
```

Listing 3: Part of a valid Java program

```
<T> List<T> emptyList() { ... }
List<String> ls = emptyList();
```

adds the type arguments to `List` at object creation and it infers the most specific return type `List<?>`, which is a wildcard type. Our algorithm is the first to infer wildcard types that comes with a soundness proof. Previous work on global type inference for Java either does not consider wildcards or it simplifies the problem by not modeling key features of Java wildcards.

Global Type Inference for Featherweight Java [19] is a sound, but incomplete inference algorithm for Featherweight Generic Java. It does not support wildcards, which means that it infers the return type `Object` for the method `genList` in Listing 1. Like our algorithm, their algorithm restricts to monomorphic recursion, which leads to incompleteness.

A type unification algorithm for Java with wildcards [11] states the same capabilities, but it does not handle capture conversion correctly because it only supports types which are expressible in Java syntax (more details in section 4.2). Moreover, it appears that the subtype relation changes depending on whether a type is used as an argument to a method invocation. We resolve this problem by modeling Java wildcards as existential types [2, 21], which also serve as the basis of our soundness proof.

Standard Java provides type inference in a restricted form which only works for local environments where the surrounding context has known types. The example in Listing 3 exhibits the main differences to our global type inference al-

Listing 4: Limitations of Java's Type Inference

```
emptyList().add(new List<String>())
  .get(0)
  .get(0); //Typing Error
```

gorithm. The method call `emptyList` lacks its type parameters. Java relies on a matching algorithm [18] to instantiate `T` with `String` resulting in the correct expected type `List<String>`. This local type inference depends on the type annotation on the left side of the assignment. When calling the `emptyList` method without this type context its return value will be inferred as `List<Object>`. Therefore, Java rejects the code snippet in Listing 4: it infers the type `List<Object>` for `emptyList()` instead of the required `List<List<String>>`. Hence, the second call to `get` produces a type error.

The type inference algorithm presented in this paper correctly instantiates the type parameter `T` of the `emptyList` method with `List<List<String>>` and render this code snippet correct.

We summarize our contributions:

- We introduce the language TamedFJ (section 3), a Featherweight Java derivative including generics, wildcards, and type inference.
- Our algorithm handles existential types in a form which is not denotable by Java syntax [3]. Thus, we support capture conversion and Java style method calls.
- We present a novel approach to deal with existential types and capture conversion during constraint unification.
- We prove soundness and aim for a good compromise between completeness and time complexity.

2 Java Wildcards

As Java is an imperative language, subtyping for generic types is invariant. Even though `String` is subtype of `Object`, a `List<String>` is not a subtype of `List<Object>`: because someone might store an `Integer` in the list, which is compatible with `Object`, but not with `String` (see Listing 5).

Invariance is overly restrictive in read-only or write-only contexts. Hence, Java incooperates use-site variance by allowing wildcards (`?`) in types. For example, the type `List<?>` (short

Listing 5: Java Invariance Example

```
List<String> ls = ...;
List<Object> lo = ...;
lo = ls; // typing error!
lo.add(new Integer(1));
```

Listing 6: Use-Site Variance Example

```
List<String> ls = ...;
List<? extends Object> lo = ...;
lo = ls; // correct
lo.add(new Integer(1)); // error!
```

Listing 7: Wildcard Example with faulty call to a concat method

```
<X> List<X> concat(List<X> l1, List<X> l2)
{
    return l1.addAll(l2);
}

List<?> l1 = new List<String>("foo");
List<?> l2 = new List<Integer>(1); // List
    containing Integer

concat(l1, l2); // Error! Would concat two
    different lists
```

for `List<? extends Object>`, with `?` being a placeholder for any type) is a supertype of `List<String>` and `List<Object>`. Listing 6 shows a use of wildcards that renders the assignment `lo = ls` correct. The program still does not compile, because the addition of an `Integer` to `lo` is still incorrect.

Wildcard types like `List<?>` are virtual types, i.e., the run-time type of an object is always a fully instantiated type like `List<String>` or `List<Object>`. The issue is that the run-time type underlying a wildcard type can change at any time, for example when multiple threads share a reference to the same field. Hence, a wildcard `?` must be considered a different type everytime it is accessed. For that reason, the call to the method `concat` with two wildcard lists in Listing 7 is rejected.

To determine the correctness of method calls involving wildcard types Java's typechecker makes use of a concept called **capture conversion**.

One way to formalize this concept is by replacing wildcards with existential types and modeling capture conversion with suitably inserted let statements [2]. Our Featherweight Java derivative called **TamedFJ** is modeled after Bierhoff's calculus [2] (see section 3). To express the example in Listing 6 in our calculus we first translate the wildcard types: `List<? extends Object>` becomes $\exists A : [\perp..Object].List<A>$, where the existentially

Listing 8: TamedFJ representation of the concat call from listing 7

```
let l1' :  $\exists X.List<List<X>>$  = l1 in
let l2' :  $\exists Y.List<List<Y>>$  = l2 in
concat(l1', l2') // Error!
```

bound variable `A` has a lower bound \perp and an upper bound `Object`. Before we can call the add method on this type we perform capture conversion by inserting a let statement:

```
let v :  $\exists A : [\perp..Object].List<A>$  = lo in v.<A>.add
    (new Integer(1));
```

The variable `lo` (from Listing 6) is assigned to a new immutable variable `v` with type $\exists A : [\perp..Object].List<X>$, but inside the let statement the variable `v` will be treated as `List<A>`. Here `A` is a fresh variable or a captured wildcard. The only information we have about `A` is that it is a supertype of \perp and a subtype of `Object`. It is important to give the captured wildcard type `A` an unique name which is used nowhere else. This approach also clarifies why the method call to `concat` in listing 7 is rejected (see Listing 8).

3 TamedFJ

This section defines the calculus **TamedFJ**, which is used as input and output of our global type inference algorithm. Figure 2 contains the syntax with optional type annotations highlighted in yellow. The respective typing rules are defined in Figures 5 and 6. **TamedFJ** is a subset of the calculus defined by Bierhoff [2], but in addition we make the types for method arguments and return types optional. The point is that a correct and fully typed **TamedFJ** program is also a correct Featherweight Java program, which is vital for our soundness proof.

A method assumption consists of a method name, a list of type variables, a list of argument types, and a return type. The first type in the list of argument types is the type of the surrounding class also known as the `this` parameter. See Figure 1 for an example, where the `add` method is treated internally as a method with two arguments, because we add `this` to its argument list. The type variable `A` of the surrounding class is part of the method's list of type parameters.

```
class List<A extends Object> {
    List<A> add(A v){..,}
}

Π = {
    add : <A <Object> List<A>, A → List<A>
}
```

Figure 1: TamedFJ class and its corresponding method type environment

Additional Notes:

- We require type annotations for fields and generic class parameters, as we consider them as data declarations which are given by the programmer.
- We add the elvis operator (?:) to the syntax to easily showcase applications involving wildcard types. This operator is used to emulate Java's if-else expression but without the condition clause. Our operator just returns one of the two given statements at random. The point is that the return type of the elvis operator has to be a supertype of its two subexpressions.
- The new expression does not require generic parameters.
- Every method has an unique name. The calculus does not include method overriding for simplicity. Type inference with method overriding is described elsewhere [19] and can be added to our algorithm in a similar way.
- The T-PROGRAM typing rule ensures that there is one set of method assumptions used for all classes that are part of the program.
- Unlike previous work [19], the typing rules for expressions shown in Figure 5 allow for polymorphic recursion. Type inference for polymorphic recursion is undecidable [23] and when proving completeness the calculus needs to be restricted in that regard (cf. [19]). Our algorithm is not complete (see discussion in section 8), so we keep the **TamedFJ** calculus as simple as possible and close to Featherweight Java to simplify the soundness proof.

4 Global Type Inference Algorithm

This section gives an overview of the global type inference algorithm with examples and identifies the challenges that we had to overcome in the design of the algorithm.

Listings 9, 10, 11, and 12 showcase the global type inference algorithm step by step. Note that regular Java code snippets are displayed in a grey box, **TamedFJ** programs are yellow and constraints have a red background. In the Java code in Listing 9, the type of variable `l` is an existential type so that it has to undergo a capture conversion before being passed to a method call. To this end we convert the program to A-Normal form (Listing 10), which introduces a let statement defining a new variable `v`. The constraint generation step also assigns type placeholders to all variables missing a type annotation. In this case the algorithm puts the type placeholder `v` for the type of `v` before generating constraints (see Listing 11). These constraints mirror the typing rules of the **TamedFJ** calculus (section 3). The call to `add` generates the *capture constraint* $v \ll^c \text{List}\langle a? \rangle$. This constraint (\ll^c) is a kind of subtype constraint, which additionally expresses that the left side of the constraint is subject to a capture conversion. Next, the unification algorithm **Unify** (section 6) solves the constraints. Having the constraints in listing 11 as an input **Unify** changes the constraint $\exists X : [\perp..String].\text{List}\langle X \rangle \ll v$ to $v \doteq \exists X : [\perp..String].\text{List}\langle X \rangle$ thereby finding a type solution for `v`. Afterwards every occurrence of `v` in the constraint set is replaced by $\exists X : [\perp..String].\text{List}\langle X \rangle$ leaving us with the following constraints which are now subject to a capture conversion by the **Unify** algorithm:

$$\frac{\begin{array}{l} \exists X : [String..Object].\text{List}\langle X \rangle \ll^c \\ \text{List}\langle a? \rangle, String \ll^c a? \end{array}}{Y : [String..Object] \vdash \text{List}\langle Y \rangle \ll \text{List}\langle a? \rangle, String \ll a?} \text{Capture}$$

The constraint $\exists X : [String..Object].\text{List}\langle X \rangle \ll^c \text{List}\langle a? \rangle$ allows **Unify** to do a capture conversion to $\text{List}\langle X \rangle \ll \text{List}\langle a? \rangle$. The captured wildcard `X` gets a fresh name `Y`, which is stored in the wildcard environment of the **Unify** algorithm. Substituting `Y` for `a?` yields the constraints almost solved: $\text{List}\langle Y \rangle \ll \text{List}\langle Y \rangle, String \ll Y$. The first constraint is obviously satisfied and $String \ll Y$ is satisfied because `Y` has `String` as lower bound.

Parameterized classes	$N ::= C\langle\bar{T}\rangle$
Types	$S, T, U ::= X \mid \exists\Delta.N$
Lower bounds	$K, L ::= T \mid \perp$
Type variable contexts	$\Delta ::= \overline{X : [L..T]}$
Class declarations	$D ::= \text{class } C\langle\bar{X}\langle\bar{T}\rangle\rangle\langle N \{ \overline{T f; \bar{M}} \}$
Method declarations	$M ::= \langle\bar{X}\langle\bar{N}\rangle\rangle \bar{T} m(\overline{T x}) \{ \text{return } e; \}$
Terms	$e ::=$ x $\text{new } C\langle\bar{T}\rangle(\bar{e})$ $e.f$ $e.m\langle\bar{T}\rangle(\bar{e})$ $\text{let } x : \exists\Delta.N = e \text{ in } e$ $e ? : e$
Variable contexts	$\Gamma ::= \overline{x : T}$
Method type environment	$\Pi ::= \overline{m : \langle\bar{X}\langle\bar{N}\rangle\rangle \bar{T} \rightarrow T}$

Figure 2: Input Syntax with optional type annotations

$\text{S-REFL} \quad \frac{}{\Delta \vdash T <: T}$	$\text{S-TRANS} \quad \frac{\Delta \vdash S <: T \quad \Delta \vdash T <: U}{\Delta \vdash S <: U}$	$\text{S-UPPER} \quad \frac{X : [L..U] \in \Delta}{\Delta \vdash X <: U}$	$\text{S-LOWER} \quad \frac{X : [L..U] \in \Delta}{\Delta \vdash L <: X}$
$\text{S-EXTENDS} \quad \frac{\text{class } C\langle\bar{X}\langle\bar{U}\rangle\rangle\langle N \{ \dots \} \quad \bar{X} \cap \text{dom}(\Delta) = \emptyset}{\Delta \vdash \exists\Delta'. C\langle\bar{T}\rangle <: \exists\Delta'. [\bar{T}/\bar{X}]N}$	$\text{S-EXISTS} \quad \frac{\Delta', \Delta \vdash [\bar{T}/\bar{X}]\bar{L} <: \bar{T} \quad \Delta', \Delta \vdash \bar{T} <: [\bar{T}/\bar{X}]\bar{U} \quad \text{fv}(\bar{T}) \subseteq \text{dom}(\Delta, \Delta') \quad \text{dom}(\Delta') \cap \text{fv}(\exists X : [L..U].N) = \emptyset}{\Delta \vdash \exists\Delta'. [\bar{T}/\bar{X}]N <: \exists X : [L..U].N}$		

Figure 3: Subtyping

$\text{WF-BOT} \quad \Delta \vdash \perp \text{ ok}$	$\text{WF-TOP} \quad \frac{\Delta \vdash \bar{L}, \bar{U} \text{ ok}}{\Delta \vdash \bar{W} : [\bar{L}..U].\text{Object}}$	$\text{WF-VAR} \quad \frac{W : [L..U] \in \Delta \quad \Delta \vdash \bar{L}, \bar{U} \text{ ok}}{\Delta \vdash W : [L..U] \text{ ok}}$
$\text{WF-CLASS} \quad \frac{\Delta' = \bar{W} : [\bar{L}..U] \quad \Delta, \Delta' \vdash \bar{T}, \bar{L}, \bar{U} \text{ ok} \quad \Delta, \Delta' \vdash \bar{L} <: \bar{U} \quad \Delta, \Delta' \vdash \bar{T} <: [\bar{T}/\bar{X}]\bar{U} \quad \text{class } C\langle\bar{X}\langle\bar{U}'\rangle\rangle\langle N \{ \dots \}}{\Delta \vdash \exists\bar{W} : [\bar{L}..U].C\langle\bar{T}\rangle \text{ ok}}$		

Figure 4: Well-formedness

$$\begin{array}{c}
\text{T-VAR} \\
\frac{x : T \in \Gamma}{\Delta | \Gamma \vdash x : T} \\
\\
\text{T-NEW} \\
\frac{\Delta, \bar{\Delta} \vdash C \langle \bar{T} \rangle \text{ ok} \quad \Delta \vdash \bar{S} <: \bar{U} \quad \Delta | \Gamma \vdash \bar{v} : \bar{S} \quad \text{fields}(C \langle \bar{T} \rangle) = \bar{U} \bar{f}}{\Delta | \Gamma \vdash \text{new } C(\bar{v}) : C \langle \bar{T} \rangle} \\
\\
\text{T-FIELD} \\
\frac{\Delta | \Gamma \vdash v : T \quad \Delta \vdash T <: \exists N \quad \text{fields}(N) = \bar{U} \bar{f}}{\Delta | \Gamma \vdash v.f_i : U_i} \\
\\
\text{T-CALL} \\
\frac{\langle \bar{X} \triangleleft \bar{U}' \rangle \bar{U} \rightarrow U \in \Pi(m) \quad \Delta \vdash \bar{S} <: [\bar{S}/\bar{X}] \bar{U}' \quad \Delta \vdash \bar{S} \text{ ok} \quad \Delta | \Gamma \vdash v, \bar{v} : \bar{T} \quad \Delta \vdash \bar{T} <: [\bar{S}/\bar{X}] \bar{U}}{\Delta | \Gamma \vdash v.m(\bar{v}) : T} \\
\\
\text{T-LET} \\
\frac{\Delta, \Delta' | \Gamma, x : \exists N \vdash t_2 : T_2 \quad \Delta | \Gamma \vdash t_1 : T_1 \quad \Delta \vdash T_1 <: \exists \Delta' . N \quad \text{dom}(\Delta') \subseteq \text{fv}(N) \quad \Delta \vdash T, \exists \Delta' . N \text{ ok}}{\Delta | \Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T} \\
\\
\text{T-ELVIS} \\
\frac{\Delta | \Gamma \vdash t_1 : S \quad \Delta | \Gamma \vdash t_2 : T \quad \Delta \vdash S <: U \quad \Delta \vdash T <: U}{\Delta | \Gamma \vdash t_1 ? : t_2 : U}
\end{array}$$

Figure 5: Expression Typing

$$\begin{array}{c}
\text{T-METHOD} \\
\frac{\text{class } C \langle \bar{X} \triangleleft \bar{U} \rangle \triangleleft N \{ \dots \} \quad \langle \bar{Y} \triangleleft \bar{N} \rangle (C \langle \bar{X} \rangle, \bar{T}) \rightarrow T \in \Pi(m) \quad \Delta' = \bar{Y} : \perp..P \quad \Delta, \Delta' \vdash \bar{P}, T, \bar{T} \text{ ok} \quad \text{dom}(\Delta) = \bar{X} \quad \Pi | \Delta, \Delta' | \bar{x} : \bar{T}, \text{this} : C \langle \bar{X} \rangle \vdash e : S \quad \Delta \vdash S <: T}{\Pi | \Delta \vdash m(\bar{x}) \{ \text{return } e; \} \text{ ok in } C} \\
\\
\text{T-CLASS} \\
\frac{\Delta = \bar{X} : \perp..U \quad \Delta \vdash \bar{U}, \bar{T}, N \text{ ok} \quad \Pi | \Delta \vdash \bar{M} \text{ ok in } C}{\Pi \vdash \text{class } C \langle \bar{X} \triangleleft \bar{U} \rangle \triangleleft N \{ \bar{T} \bar{f}; \bar{M} \}} \\
\\
\text{T-PROGRAM} \\
\frac{\Pi = m : \langle \bar{X} \triangleleft \bar{N} \rangle \bar{T} \rightarrow T}{\Pi \vdash \bar{D}}
\end{array}$$

Figure 6: Class and Method Typing rules

$$\begin{array}{c}
\text{fields}(\text{Object} \langle \rangle) = \emptyset \\
\\
\text{F-CLASS} \\
\frac{\text{class } C \langle \bar{X} \triangleleft _ \rangle \triangleleft N \{ \bar{S} \bar{f}; \bar{M} \} \quad \text{fields}([\bar{T}/\bar{X}]N) = \bar{U} \bar{g}}{\text{fields}(C \langle \bar{T} \rangle) = \bar{U} \bar{g}, [\bar{T}/\bar{X}] \bar{S} \bar{f}}
\end{array}$$

Figure 7: Field access

Listing 9: Valid Java program

```
<A> List<A> add(List<A> l, A v)

List<? super String> l = ...;
add(l, "String");
```

Listing 10: TamedFJ representation

```
<A> List<A> add(List<A> l, A v)
List<? super String> l = ...;
let v:v = l
  in add(v, "String");
```

Listing 11: Constraints

```
 $\exists X : [\perp..String].List<X> \triangleleft v$ 
 $v \triangleleft_x^c List<a_?>$ 
 $String \triangleleft a_?$ 
```

Listing 12: Type solution

```
<A> List<A> add(List<A> l, A v)
List<? super String> l = ...;
let l2: $\exists X : [String..Object].List<X>$  = l
  in <X>add(l2, "String");
```

A correct Featherweight Java program including all type annotations and an explicit capture conversion via let statement is shown in Listing 12. This program can be deduced from the solution of the **Unify** algorithm. In the body of the let statement the type $\exists X : [String..Object].List<X>$ becomes $List<X>$ and the capture converted wildcard $X : [String..Object]$ can be used as a type parameter of the call $<X>add(v, "String")$.

4.1 Notes on Capture Constraints

Capture constraints must be stored in a multiset so that it is possible that two syntactically equal constraints remain in the same set. The equality relation on Capture constraints is not reflexive.

Capture Constraints are bound to a variable. For example a let statement like `let x = v in x.get()` will create the capture constraint $x \triangleleft_x^c List<a_?>$. This time we annotated the capture constraint with an x to show its relation to the variable x . Let's do the same with the constraints generated by the `concat` method invocation in listing 13, creating the constraints 14.

During the **Unify** process it could happen that two syntactically equal capture constraints evolve, but they are not the same because they are each linked to a different let introduced variable. In this example this happens when we substitute $\exists X.List<X>$ for x and y resulting in:

```
 $\exists X.List<X> \triangleleft_x^c List<a_?>, \exists X.List<X> \triangleleft_y^c List<a_?>$ 
```

Listing 13: Faulty Method Call

```
List<?> v = ...;

let x = v in
  let y = v in
    concat(x, y) // Error!
```

Listing 14: Annotated constraints

```
 $x \triangleleft_x^c List<a_?>, \exists X.List<X> \triangleleft x$ 
 $y \triangleleft_y^c List<a_?>, \exists X.List<X> \triangleleft y$ 
```

Thanks to the original annotations we can still see that those are different constraints. After **Unify** uses the (CAPTURE) rule on those constraints it gets obvious that this constraint set is unsolvable:

```
 $List<X> \triangleleft List<a_?>, List<Y> \triangleleft List<a_?>$ 
```

This paper will not annotate capture constraints with variable names. Instead we consider every capture constraint as distinct to other capture constraints even when syntactically the same, because we know that each of them originates from a different let statement. *Hint:* An implementation of this algorithm has to consider that seemingly equal capture constraints are actually not the same and has to allow doubles in the constraint set.

Note: In the special case `let x = v in concat(x, x)` the constraints would look like $\exists X.List<X> \triangleleft_x^c List<a_?>, \exists X.List<X> \triangleleft_x^c List<a_?>$ and we could actually delete one of them without losing information. But this case will never occur in our algorithm, because the let statements for our input programs are generated by transformation to ANF (see 5.1).

4.2 Challenges

Global type inference for Featherweight Generic Java without wildcards has been considered elsewhere [19]. Adding wildcards to the calculus created new problems, which have not been recognized by existing work on type unification for Java with wildcards [11]. Java's wildcard types are represented as existential types that have to be opened before they can be used. Opening can either be done implicitly ([3], [18]) or explicitly via a let statement ([2]). For all variations it is vital to know the argument types with which a method is called. In the absence of type annotations, we do not know where an existential type will emerge and where a capture conversion is

necessary. Rather, the type inference algorithm has to figure out the placement of existentials. We identified three main challenges related to Java wildcards and global type inference.

1. One challenge is to design the algorithm in a way that it finds a correct solution for programs like the one shown in Listing 9 and rejects programs like the one in Listing 7. The first one is a valid Java program, because the type `List<? super String>` is *capture converted* to a fresh type variable `X` which is used as the generic method parameter for the call to `add` as shown in Listing 10. Because we know that the type `String` is a subtype of the free variable `X`, it is safe to pass "String" for the first parameter of the function.

The second program shown in Listing 7 is incorrect. The method call to `concat` with two wildcard lists is unsound. The element types of the lists may be unrelated, therefore the call to `concat` cannot succeed. The problem gets apparent if we try to write the `concat` method call in the **TamedFJ** calculus (Listing 8): `l1'` and `l2'` are two different lists inside the body of the `let` statements, namely `List<X>` and `List<Y>`. For the method call `concat(x1, x2)` no replacement for the generic `A` exists to satisfy `List<A> <: List<X>, List<A> <: List<Y>`.

2. The program in Listing 15 exhibits a challenge involving wildcards and subtyping. The method call `shuffle(l)` is incorrect, because `l` has type `List<∃X.List<X>>` representing a list of unknown lists. However, `∃X.List2D<X>` is a subtype of `∃X.List<List<X>>` which represents a list of lists, all of the same type `X`, and can be passed safely to `shuffle`. This behavior can also be explained by looking at the types and their capture converted versions: 8

Listing 15: Intermediate Types Example

```
class List<X> extends Object {...}
class List2D<X> extends List<List<X>>
  {...}

<X> void shuffle(List<List<X>> list)
  {...}

List<List<?>> l = ...;
List2D<?> l2d = ...;

shuffle(l); // Error
shuffle(l2d); // Valid
```

Given such a program the type inference algorithm has to allow the call `shuffle(l2d)`

and decline the call to `shuffle(l)`.

3. Take the Java program in listing 16 for example. It uses `map` to apply a polymorphic method `id` to every element of a list `l : List<∃A.List<A>>`.

How do we get the **Unify** algorithm to determine the correct type for the variable `l2`? Although we do not specify constraint generation for language constructs like lambda expressions used in this example, we can imagine that the constraints have to look like in Listing 17.

Listing 16: List Map Example

```
<X> List<X> id(List<X> l){ ... }
List<List<?>> ls;
l2 = l.map(x -> id(x));
```

Listing 17: Constraints

```
∃A.List<A> <^c List<x?>
List<x?> < z,
List<z> < l2
```

The constraints `∃A.List<A> <^c List<x?>, List<x?> < z` stem from the body of the lambda expression `id(x)`. *For clarification:* This method call would be represented as the following expression in **TamedFJ**:

```
let x1 : ∃A.List<A> = x in id(x) : z
```

The T-Let rule prevents us from using free variables created by the method call to `id` to be used in the return type `z`. But this restriction has to be communicated to the **Unify** algorithm, which does not know about the origin and context of the constraints. If we naively substitute $\sigma(z) = \text{List}\langle A \rangle$ the return type of the `map` function would be the type `List<List<A>>`, which would be unsound.

5 Constraint generation

The constraint generation works on the **TamedFJ** language. This step is mostly the same as in [19] except for field access and method invocation. We will focus on those two parts where also the new capture constraints and wildcard type placeholders are introduced.

Before generating constraints the input is transformed by an ANF transformation (see section 5.1).

Java type	TamedFJ representation	Capture Conversion
List<List<?>>	List< $\exists X$.List<X>>	List< $\exists X$.List<X>>
List2D<?>	$\exists X$.List2D<X>	List2D<X>

Figure 8: Capture Converted Versions

Listing 18: TamedFJ example

```
m(l, v){
  return l.add(v);
}
```

Listing 19: A-Normal form

```
m(l, v) =
  let x1 = l in
  let x2 = v in x1.add(x2)
```

5.1 ANF transformation

Featherweight Java’s syntax involves no `let` statement and terms can be nested freely similar to Java’s syntax. Our calculus **TamedFJ** uses `let` statements to explicitly apply capture conversion to wildcard types, but we don’t know which expressions will spawn wildcard types because there are no type annotations yet. To emulate Java’s behaviour we have to preemptively add capture conversion in every suitable place. This is done by a *A-Normal Form* [17] transformation shown in figure 9. After this transformation every method invocation is preceded by `let` statements which perform capture conversion on every argument before passing them to the method. See the example in listings 18 and 19.

5.2 Constraint Generation Algorithm

The parameter types given to a generic method also affect their return type. During constraint generation the algorithm does not know the parameter types yet. We generate \llcorner^c constraints and let **Unify** do the capture conversion. \llcorner^c constraints are kept until they reach the form $G \llcorner^c G$ and a capture conversion is possible.

At points where a well-formed type is needed we use a normal type placeholder. Inside a method call expression sub expressions (receiver, parameter) wildcard placeholders are used. Here captured variables can flow freely. A normal type placeholder cannot hold types containing free variables. Normal type placeholders are assigned types which are also expressible with Java syntax. So no types like $\exists X$.Pair<X,X> or $\exists X$.List<List<X>>.

It is possible to feed the **Unify** algorithm a set of free variables with predefined bounds. This is used for class generics see figure 12. The **FJ-Type** function returns a set of constraints as well as an initial environment Δ containing the generics declared by this class. Those type variables count as regular types and can be held by normal type placeholders.

TYPEExpr($\Pi, \Gamma, e.f, a$) =

let r fresh
 $C_R = \mathbf{TYPEExpr}(\Pi, \Gamma, e, r)$
 $c = \mathbf{oc}\{\{r \llcorner^c C \llcorner \bar{a}_?, a \doteq$
 $\quad [\bar{a}_?/\bar{X}]T, \bar{a}_? \llcorner [\bar{a}_?/\bar{X}]\bar{N}\}$
 $\quad | T f \in \mathbf{class} C \llcorner \bar{X} \llcorner \bar{N} \rangle \{\bar{T} f; \dots\},$
 $\quad \bar{a}_? \text{ fresh}\}$
in $C_R \cup \{c\}$

TYPEExpr($\Pi, \Gamma, \mathbf{let} x = e_1 \text{ in } e_2, a$) =

let e_1, e_2, x fresh
 $C_1 = \mathbf{TYPEExpr}(\Pi, \Gamma, e_1, e_1)$
 $C_2 = \mathbf{TYPEExpr}(\Pi \cup \{x : x\}, e_2, e_2)$
 $c = \{e_1 \llcorner x, e_2 \llcorner a\}$
in $C_1 \cup C_2 \cup \{c\}$

TYPEExpr($\Pi, \Gamma, v.m(\bar{v}), a$) =

let r, \bar{r} fresh
 $c = [\bar{b}_?/\bar{Y}]\{\bar{S} \llcorner^c \bar{T}, T \llcorner a, \bar{Y} \llcorner \bar{N}\}$
in $(C_R \cup C \cup c, T)$
where $v, \bar{v} : \bar{S} \in \eta$
 $m : \llcorner \bar{Y} \llcorner \bar{N} \rangle \bar{T} \rightarrow T \in \Pi$

TYPEExpr($\Pi, \Gamma, e_1 ? : e_2, a$) =

let r_1, r_2 fresh
 $C_1 = \mathbf{TYPEExpr}(\Pi, \Gamma, e_1, r_2)$
 $C_2 = \mathbf{TYPEExpr}(\Pi, \Gamma, e_2, r_2)$
in $C_1 \cup C_2 \cup \{r_1 \llcorner a, r_2 \llcorner a\}$

TYPEExpr(Π, Γ, x, a) = $\Pi(x)$

$$\begin{aligned}
\tau(\mathbf{x}) &= \mathbf{x} \\
\tau(\text{new } \mathbf{C}(\bar{t})) &= \text{let } \bar{x} = \tau(\bar{t}) \text{ in new } \mathbf{C}(\bar{x}) \\
\tau(t.f) &= \text{let } x = \tau(t) \text{ in } x.f \\
\tau(t.m(\bar{t})) &= \text{let } x = \tau(t) \text{ in let } \bar{x} = \tau(\bar{t}) \text{ in } x.m(\bar{x}) \\
\tau(t_1 ? : t_2) &= \tau(t_1) ? : \tau(t_2) \\
\tau(\text{let } x = t_1 \text{ in } t_2) &= \text{let } x = \tau(t_1) \text{ in } \tau(t_2)
\end{aligned}$$

Figure 9: ANF Transformation τ

Terms $t ::=$

- | $\text{let } \bar{x}_c = \bar{t} \text{ in new } \mathbf{C}(\bar{x}_c)$
- | $\text{let } x_c = t \text{ in } x_c.f$
- | $\text{let } x_c = t \text{ in let } \bar{x}_c = \bar{t} \text{ in } x_c.m(\bar{x}_c)$
- | $t ? : t$

Figure 10: Syntax of a TamedFJ program in A-Normal Form

C	$::= \bar{c}$	Constraint set
c	$::= T < T \mid T <^c T \mid T \doteq T$	Constraint
T, U, L	$::= a \mid G$	Type placeholder or Type
a	$::= \underline{a} \mid a?$	Normal and wildcard type placeholder
G	$::= X \mid N$	Wildcard, or Class Type
N, S	$::= \exists \Delta. C < \bar{T} >$	Class Type
Δ	$::= \bar{W}$	Wildcard Environment
W	$::= X : [L..U]$	Wildcard

Figure 11: Syntax of types and constraints used by FJType and Unify

$$\begin{aligned}
&\mathbf{FJType}(\Pi, \text{class } C < \bar{X} < \bar{N} > \text{ extends } N \{ \bar{T} \bar{f}; \bar{M} \}) = \\
&\quad \mathbf{let} \quad \bar{\lambda} = \{ m : (C < \bar{X} >, \bar{a} \rightarrow a) \mid \{ m(\bar{x}) = e \} \in \bar{M}, a, \bar{a} \text{ fresh} \} \\
&\quad \quad \Delta = \{ \bar{X} : [L..N] \} \\
&\quad \quad C = \{ \mathbf{TYPEExpr}(\Pi \cup \bar{\lambda} \cup \{ \text{this} : C < \bar{X} >, \bar{x} : \bar{a} \}, e, a) \\
&\quad \quad \quad \mid \{ m(\bar{x}) = e \} \in \bar{M}, m : (C < \bar{X} >, \bar{a} \rightarrow a) \in \bar{\lambda} \} \\
&\quad \mathbf{in} \quad (\Delta, C)
\end{aligned}$$

Figure 12: Constraint generation for classes

```

TYPEExpr( $\Pi, \Gamma, \text{new } C(\bar{e}), a) =$ 
  let  $\bar{x}, \bar{a}$  fresh
       $\bar{C} = \text{TYPEExpr}(\Pi, \Gamma, \bar{e}, \bar{x})$ 
       $C = \{\bar{x} \leftarrow [\bar{a}/\bar{X}]\bar{T}, \bar{a} \leftarrow \bar{N}$ 
        | class  $C \langle \bar{X} \leftarrow \bar{N} \rangle \{ \bar{T} f; \dots \}$ 
      }
  in  $\bar{C} \cup \{a \doteq C \langle \bar{a} \rangle\}$ 

```

Example 1. Given the following input program missing type annotations for the example method:

```

class Class1 {
  <A> A head(List<X> l) { ... }
  List<? extends String> get() { ... }
}

class Class2 {
  example(c1) {
    return c1.head(c1.get());
  }
}

```

We assume the class *Class1* has already been processed by our type inference algorithm leading to the following type annotations:

$$\Pi = \left\{ \begin{array}{l} m : \langle A \leftarrow \text{Object} \rangle (\text{Class1}, \text{List}\langle A \rangle) \\ \quad \rightarrow X, \\ \text{get} : (\text{Class1}) \\ \quad \rightarrow \exists A : [\text{String}.\text{Object}].\text{List}\langle A \rangle \end{array} \right\}$$

At first we have to convert the example method to a syntactically correct **TamedFJ** program. Afterwards the **FJType** algorithm is able to generate constraints.

```

class Class2 {
  example(c1) = let x = c1 in
    let xp = x.get() in x.m(xp);
}

```

```

 $c1 \leftarrow x, x \leftarrow^c \text{Class1},$ 
 $c1 \leftarrow x, x \leftarrow^c \text{Class1},$ 
 $\exists A : [\perp..\text{String}].\text{List}\langle A \rangle \leftarrow xp,$ 
 $xp \leftarrow^c \text{List}\langle a? \rangle$ 

```

Following is a possible solution for the given constraint set:

```

class Class2 {
  example(c1) = let x : Class1 = c1 in
    let xp :  $\exists A : [\perp..\text{String}].\text{List}\langle A \rangle = x.get()$ 
      in x.m(xp);
}

```

```

 $\sigma(x) = \text{Class1},$ 
 $\sigma(xp) = \exists A : [\perp..\text{String}].\text{List}\langle A \rangle$ 

```

For $\exists A : [\perp..\text{String}].\text{List}\langle A \rangle$ to be a correct solution for *xp* the constraint $\exists A : [\perp..\text{String}].\text{List}\langle A \rangle \leftarrow^c \text{List}\langle a? \rangle$ must be satisfied. This is possible, because we deal with a capture constraint. The \leftarrow^c constraint allows the left side to undergo a capture conversion which leads to $\text{List}\langle A \rangle \leftarrow \text{List}\langle a? \rangle$. Now a substitution of the wildcard placeholder $a?$ with *A* leads to a satisfied constraint set.

The wildcard placeholders are not used as parameter or return types of methods. Or as types for variables introduced by let statements. They are only used for generic method parameters during a method invocation. Type placeholders which are not flagged as wildcard placeholders ($a?$) can never hold a free variable or a type containing free variables. This practice hinders free variables to leave their scope. The free variable *A* generated by the capture conversion on the type $\exists A : [\perp..\text{String}].\text{List}\langle A \rangle$ cannot be used anywhere else then inside the constraints generated by the method call *x.m(xp)*.

5.3 Constraint Generation

The constraint generation is defined on a subset of Java displayed in figure 13. The input language includes only a small set of expressions like method calls, field access and a elvis operator *?:* which acts as a replacement for if-else expressions.

We explain the process using the following example input. The classes *List* and *Id* are already fully typed and we want to create constraints for the class *CExample* now.

```

class Id {
  <X> X id(X x) { return x; }
}

class List<X> {
  <Y extends X> List<X> concat(List<Y> l) {
    ... }
}

class CExample {
  example(p1, p2) {
    return p1.id(p2).concat(p2);
  }
}

```

The first step of constraint generation is to assign type placeholders to every expression in the input program. Type placeholders also fill in for missing type annotations in method headers.

Parameterized classes	$N ::= C \langle \bar{T} \rangle$
Types	$S, T, U ::= X \mid \exists \Delta. N$
Lower bounds	$K, L ::= T \mid \perp$
Type variable contexts	$\Delta ::= \overline{X : [L..T]}$
Class declarations	$D ::= \text{class } C \langle \bar{X} \triangleleft \bar{T} \rangle \triangleleft N \{ \overline{T f; \bar{M}} \}$
Method declarations	$M ::= \langle \bar{X} \triangleleft \bar{N} \rangle \overline{T m(\bar{T} \bar{x}) \{ \text{return } e; \}}$
Terms	$e ::=$ <ul style="list-style-type: none"> x $\text{new } C \langle \bar{T} \rangle (\bar{e})$ $e.f$ $e.m \langle \bar{T} \rangle (\bar{e})$ $e ? : e$
Variable contexts	$\Gamma ::= \overline{x : T}$
Method type environment	$\Pi ::= m : \langle \bar{X} \triangleleft \bar{N} \rangle \overline{T} \rightarrow T$

Figure 13: Input Syntax with optional type annotations

```
class CExample{
  example(p1, p2) {
    return p1.id(p2).concat(p2);
  }
}

class CExample{
  a example(b p1, c p2) {
    return ((p1:b).id(p2:c):d).concat(p2:c)
           : e;
  }
}
```

The placeholders $a - e$ are freshly created in this example and added to every expression. The type of local variable expressions like $p1$ and $p2$ is already known and can be assigned directly. $p1 : b$ and $p2 : c$ in this case. The method call to `id` gets the fresh type placeholder d as type and the method call to `concat` is assigned the placeholder e .

Afterwards we create a method type environment Π containing all method declarations. Each entry has the form $m : \langle \bar{X} \triangleleft \bar{N} \rangle \overline{T} \rightarrow T$. The type arguments of the surrounding class and the type arguments of each method are merged together leading to the list $\langle X, Y \triangleleft X \rangle$ for the `concat` method (\triangleleft is short for `extends`). Also the type of the surrounding class is added to the parameter list of the method. This leads to the `id` method having two arguments. One is the `Id` class itself and the second one is the actual argument of the method.

$$\Pi = \left\{ \begin{array}{l} \text{id} : \langle X \rangle \text{Id}, X \rightarrow X, \\ \text{concat} : \langle X, Y \triangleleft X \rangle \text{List} \langle X \rangle, \text{List} \langle Y \rangle \\ \quad \rightarrow \text{List} \langle X \rangle, \\ \text{example} : \text{CExample}, b, c \rightarrow a, \end{array} \right\}$$

Note: type placeholders are used for the `example` method.

Our constraint generation rules have the form: $\Pi \vdash e : a \implies C$, that given a method environment Π and an expression e with type a generates the constraint set C .

According to the Method-Cons rule the constraints generated by the call to `id` are: $\Pi \vdash (p1 : b).id(p2 : c) : d \implies \{ b \triangleleft x_1, c \triangleleft x_2, x_1 \triangleleft^c \text{Id}, x_2 \triangleleft^c b?, b? \triangleleft d \}$.

5.4 Or-Constraints

In case the input program contains multiple method declarations holding the same name and same amount of parameters then so called Or-Constraints must be generated. Usually Java is able to determine which method to call based on the argument's types passed to the method. During the constraint generation step the argument types are unknown and we have to assume multiple methods as invocation target.

```
class String{
  bool equals(String s){ .. }
}
class Int{
  bool equals(Int i){ .. }
}
class OrConsExample{
  m(a, b){
    return a.equals(b);
  }
}
```

The method call to `equals` now has multiple possibilities. It could either be a call to the method in the class `Int` or in `String`. The method type environment therefore contains two versions of the `equals` method:

$$\begin{array}{c}
\text{METHOD-CONS} \\
\frac{\Pi \vdash e : e \Rightarrow C \quad \overline{\Pi \vdash e : e \Rightarrow C} \quad m : \langle \overline{Y} \langle \overline{N} \rangle \overline{T}_r, \overline{T} \rightarrow T \in \Pi}{\overline{b?}, x, \overline{x} \text{ fresh} \quad C_m = \{e \langle x, \overline{e} \langle x \rangle\} \cup [\overline{b?}/\overline{Y}]\{x \langle \overline{c} \overline{T}_r, x \langle \overline{c} \overline{T}, T \langle a, \overline{Y} \langle \overline{N} \rangle\}}}{\Pi \vdash e.m(\overline{e}) : a \Rightarrow C \cup \overline{C} \cup C_m} \\
\\
\text{ELVIS-CONS} \\
\frac{\Pi \vdash e_1 : e_1 \Rightarrow C_1 \quad \Pi \vdash e_2 : e_2 \Rightarrow C_2}{\Pi \vdash e_1 ? : e_2 : a \Rightarrow C_1 \cup C_2 \cup \{e_1 \langle a, e_2 \langle a\}} \\
\\
\text{NEW-CONS} \\
\frac{\overline{\Pi \vdash e : e \Rightarrow C} \quad \text{class } D \langle \overline{X} \langle \overline{N} \rangle \{ \overline{T} \overline{f}; \dots \}}{\Pi \vdash \text{new } D(\overline{e}) : a \Rightarrow \overline{C} \cup [\overline{b?}/\overline{X}]\{e \langle \overline{T}, D \langle \overline{X} \rangle \langle a, \overline{X} \langle \overline{N} \rangle\}}
\end{array}$$

$\Pi = \{\text{equals}_1 : \text{String}, \text{String} \rightarrow \text{bool}, \text{equals}_2 : \text{Int}, \text{Int} \rightarrow \text{bool}\}$

The Or-Cons rule considers multiple declarations of the same method separately and joins all of them into a Or-Constraint.

6 Unify

Input: An environment Δ' and a set of constraints $C = \{T \langle T, T \langle \overline{c} T, T \doteq T \dots\}$

Output: Set of unifiers $Uni = \{\sigma_1, \dots, \sigma_n\}$ and an environment Δ

Unify executes the following steps until a type solution is found:

Step 1: Apply the rules depicted in the figures 23, 25 and 14 exhaustively, starting with the (CIRCLE) rule.

Step 2: The second step is nondeterministic. For every $T \langle a$ constraint **Unify** has to pick exactly one transformation from figure 28. The same principle goes for constraints of the form $a \langle N, a \langle b$ and the two transformations in figure 29. If atleast one transformation was applied in this step revert to step 1. Otherwise proceed with step 3.

Hint: When implementing this step via backtracking the rules (GENERAL) and (SUPER) should be tried first. The (SETTLE) and (RAISE) rules should only be used when none of the rules in figure 28 can be applied.

Step 3: Apply the rules in figure 30 exhaustively. (GROUND) and (FLATTEN) deal with constraints containing free variables. If a type placeholder is solely used as lower bound (GROUND) can replace it with the bottom

type. Otherwise the (FLATTEN) rule has to remove the wildcards responsible for the free variable. Note: Only one of those rules has to be applied per constraint. If the constraint set has been changed by one of these rules the algorithm must return to step 1. But if only the (SUBELIM) rule is applied or the constraint set is not changed at all, the algorithm can proceed with step 4.

The cleanup step prepares the constraint set for the last step by applying the following concepts:

Bottom type The bottom type \perp is used to generate $? \text{ extends}$ wildcard definitions. This is the only possible solution when dealing with multiple upper bounds: $a \langle T, a \langle S$ is usually not a correct solution (given S and T are no subtypes of eachother). But if a is a lower bound of a wildcard it can be set to \perp . Those constraints only stay in the constraint set after the first step if S and T do not have a common subtype. The (GROUND) rule uses this concept to generate extends Wildcards .

Step 4: (Generating Result) Apply the rules in figure 31 until $\mathbb{W} = \emptyset$ and $C = \emptyset$. The resulting Δ, σ is a correct solution.

For this step to succeed there should only be four kinds of constraints left.

1. $a \doteq b$
2. $a \langle \exists \overline{W}.C \langle \overline{T} \rangle$, with $\text{fv}(\exists \overline{W}.C \langle \overline{T} \rangle) \subseteq \Delta_i n$
3. $a \doteq \exists \overline{W}.C \langle \overline{T} \rangle$, with $a \notin \overline{T}$
4. $a \doteq X$

$$\text{OR-CONS} \quad \frac{m_1 \dots m_n \in \text{dom}(\Pi) \quad \Pi \vdash e.m_1(\bar{e}) : a \implies C_1 \quad \dots \quad \Pi \vdash e.m_n(\bar{e}) : a \implies C_n}{\Pi \vdash e.m(\bar{e}) : a \implies \text{oc}(C_1, \dots, C_n)}$$

Unify fails if there is any $a \doteq T$ such that a occurs in T . For the cases 2, 3, and 4 the placeholder a cannot appear anywhere else in the constraint set. Otherwise the generation rules (GENSIGMA) and (GENDELTA) will not be able to process every constraint.

6.1 Transformation Rules (Step 1)

Our **Unify** process uses a similar concept as the standard unification by Martelli and Montanari [6], consisting of terms, relations and variables. Instead of terms we have types of the form $C \langle \bar{T} \rangle$ and the variables are called type placeholders. The input consist out of subtype relations. The goal is to find a solution (an unifier) which is a substitution for type placeholders which satisfies all input subtype constraints. Types are reduced until they reach a form like $a \doteq T$. Afterwards **Unify** substitutes type placeholder a with T . This is done until a substitution for all type placeholders and therefore a valid solution is reached. The reduction and substitutions are done in the first step of the algorithm. The algorithms state consists out of a wildcard environment and a constraint set ($\mathbb{W} \vdash C$). Initially they are set to the input environment Δ_{in} and the input constraints.

Each calculation step of the algorithm is expressed as a transformation rule consisting of three parts. The input is shown above the line, the output below, and additional premises are displayed on the right side. If the wildcard environment \mathbb{W} and the constraint set C match the pattern declared as the input the transformation will reduce them into the specified output. The (SUBST) rule (figure 19) for example takes a constraint set that has atleast one constraint of the form $\underline{a} \doteq T$ and replaces every occurrence of \underline{a} by T in the wildcard environment \mathbb{W} aswell as the remaining constraint set C .

The (UPPER) and (LOWER) conversions (figure 14) replace wildcards with their respective bounds when appearing in a subtype constraint. (LOWER) has to check if the wildcard is part of the input wildcards Δ_{in} . If that is the case the wildcard can be part of the type solution and stays in the constraint set. *Note:* The subtype constraints in

these rules are annotated with numbers $\langle \cdot \rangle_1$. All rule inputs containing subtype constraints ($\langle \cdot \rangle$) are always meant for both kinds, subtype ($\langle \cdot \rangle$) and capture constraints ($\langle \cdot \rangle^c$). If multiple constraints are stated in the input format they will be annotated with numbers which map them to the constraints used in the output of the rule. Constraints with the same number stay the same kind. So if the input to (UPPER) is $A \langle \cdot \rangle^c G$ the output will be something like $U \langle \cdot \rangle^c G$. If the input is a normal subtype constraint the output has to be a normal subtype constraint too. But the (ADOPT) rule for example takes multiple subtype constraints and also adds a new one. Here the numbered annotations are necessary. *Example:* Having the constraints $\underline{a} \langle \cdot \rangle^c b_?$, $\underline{a} \langle \cdot \rangle \text{String}$, $b_? \langle \cdot \rangle \text{Object}$ would lead to $b_? \langle \cdot \rangle \text{String}$, $\underline{a} \langle \cdot \rangle^c b_?$, $\underline{a} \langle \cdot \rangle \text{String}$, $b_? \langle \cdot \rangle^c \text{Object}$ after applying (ADOPT). Note that the new generated constraint $b_? \langle \cdot \rangle \text{String}$ is a normal subtype constraint.

A $\perp \langle \cdot \rangle T$ constraint is always satisfied and can be ignored. It will be removed by the (BOT) rule. For the type placeholder a in the constraint $a \langle \cdot \rangle \perp$ only the \perp type is a possible substitution, which is set by the (PIT) rule. The (ERASE) rule will remove redundant $T \doteq T$ constraints (EQUALS) ensures equality of two types by ensuring they are mutual subtypes. We define two types as equal if they are mutual subtypes of each other.

Definition 6.1. Type Equality: $\Delta \vdash S = T$ if $\Delta \vdash T \langle \cdot \rangle S$ and $\Delta \vdash T \langle \cdot \rangle S$

An existential type can never be a subtype of a regular type: $\exists X.C \langle X \rangle \not\prec C \langle X \rangle$. This is a problem for constraints of the form $\exists X.C \langle X \rangle \langle \cdot \rangle N$. Take the constraint $\exists X.C \langle X \rangle \langle \cdot \rangle C \langle a_? \rangle$ for example. After applying a reduction we get $X \doteq a_?$ which is not a valid solution. The problem is that we loose the information of the left side being an existential type with the reduction step. Therefore we have to assure beforehand that errors (like $X \doteq a_?$ in this example) cannot occur. But we also don't want to exclude reductions for these kind of constraints in general. The solution is to check if there are any free variables or wildcard placeholders on the right side of the constraint. If that is the case one of the rules (TRIM), (CLEAR), or (EXCLUDE) have to be applied. In our example this would be

$$\begin{array}{l}
 \text{(UPPER)} \quad \frac{\mathbb{W} \cup \{A : [L..U]\} \vdash C \cup \{A \leq_1 G\}}{\mathbb{W} \cup \{A : [L..U]\} \vdash C \cup \{U \leq_1 G\}} \\
 \text{(LOWER)} \quad \frac{\mathbb{W} \cup \{A : [L..U]\} \vdash C \cup \{G \leq_1 A\}}{\mathbb{W} \cup \{A : [L..U]\} \vdash C \cup \{G \leq_1 L\}} \\
 \text{(LOWER)} \quad \frac{\mathbb{W} \cup \{A : [L..U]\} \vdash C \cup \{\underline{a} \leq_1 A\}}{\mathbb{W} \cup \{A : [L..U]\} \vdash C \cup \{\underline{a} \leq_1 L\}} \quad A \notin \Delta_{in} \\
 \text{(BOT)} \quad \frac{\mathbb{W} \vdash C \cup \{\perp \leq T\}}{\mathbb{W} \vdash C} \quad \text{(PIT)} \quad \frac{\mathbb{W} \vdash C \cup \{a \leq \perp\}}{\mathbb{W} \vdash C \cup \{a \doteq \perp\}}
 \end{array}$$

Figure 14: Wildcard reduce rules

$$\begin{array}{l}
 \text{(PREPARE)} \quad \frac{\mathbb{W} \vdash C \cup \{\exists \Delta. C \langle \bar{S} \rangle \leq \exists \Delta'. C \langle \bar{T} \rangle\}}{\mathbb{W} \vdash C \cup \{\exists \Delta. C \langle \bar{S} \rangle \leq^c \exists \Delta'. C \langle \bar{T} \rangle\}} \quad \begin{array}{l} \text{fv}(N') \subseteq \Delta_{in} \\ \text{wfv}(N') = \emptyset \end{array} \\
 \text{(TRIM)} \quad \frac{\mathbb{W} \vdash C \cup \{\exists \Delta, \Delta'. C \langle \bar{S} \rangle \leq T\}}{\mathbb{W} \vdash C \cup \{\exists \Delta. C \langle \bar{S} \rangle \leq T\}} \quad \text{fv}(\bar{S}) \cap \Delta' = \emptyset \\
 \text{(CLEAR)} \quad \frac{\mathbb{W} \cup \{A : [L..U]\} \vdash C \cup \{\exists \Delta. C \langle \bar{S} \rangle \leq T\}}{[U/A] \mathbb{W} \vdash [U/A] C \cup [U/A] \{\exists \Delta. C \langle \bar{S} \rangle \leq T, U \doteq L\}} \quad \begin{array}{l} \Delta \neq \emptyset \\ A \in \text{fv}(T) \end{array} \\
 \text{(EXCLUDE)} \quad \frac{\mathbb{W} \vdash C \cup \{\exists \Delta. C \langle \bar{S} \rangle \leq T\}}{[a/a?] \mathbb{W} \vdash [a/a?] C \cup [a/a?] \{\exists \Delta. C \langle \bar{S} \rangle \leq T\}} \quad \begin{array}{l} \Delta \neq \emptyset \\ a? \in \text{fv}(T), a \text{ fresh} \end{array}
 \end{array}$$

Figure 15: Dealing with wildcard types on the left side of a subtype constraint

the (EXCLUDE) rule replacing the wildcard placeholder with $x?$ with a normal placeholder. Afterwards (PREPARE) can be used eventually leading to the erasure of the wildcard x by equalizing its upper and lower bounds: 16. Note that the (PREPARE) rule is always applied together with the (CAPTURE) and the (REDUCE) rule: (TRIM) removes unused wildcard declarations. (CLEAR) and (EXCLUDE) remove wildcard placeholders or wildcards to allow the constraint to be processed by (PREPARE).

Unify keeps $a \triangleleft T$ constraints as long as possible. The (MATCH) rule reduces two $a \triangleleft T$ constraints to one. There has to be a common subtype C of D and D' for this rule to work expressed as premises $C \ll D$ and $C \ll D'$. The \ll relation is the reflexive and transitive closure of the extends relations:

$$\frac{C \triangleleft \bar{X} \triangleleft \bar{N} \triangleright \triangleleft D \triangleleft \bar{N} \triangleright}{C \ll D} \quad \frac{C \ll D, D \ll E}{C \ll E}$$

For example the constraints $a \triangleleft \text{String}$ and $a \triangleleft \text{Integer}$ are unsolvable, because there exists no common subtype of `String` and `Integer`. Whereas the constraints $a \triangleleft^c \text{List} \triangleleft \text{Integer}$ and $a \triangleleft^c \text{List} \triangleleft b$ can be processed by (MATCH): 20

After (SUBST) and (SAME) the remaining constraints are $b \doteq \text{Integer}$ and $a \triangleleft \exists A : [\text{Integer}.. \text{Integer}]. \text{List} \triangleleft A$

The (REDUCE) rule represents the S-Exists type rule. This rule uses wildcard placeholders ($\bar{a}?$) to find a possible substitution for the wildcards on the right side. The constraint $N \triangleleft \exists X : [\bar{L}.. \bar{U}]. N'$ is satisfied if there is a substitution $[\bar{T}/\bar{X}]N = N'$ with \bar{T} inside the bounds \bar{U} and \bar{L} . For example the constraint $\text{List} \triangleleft a \triangleright \triangleleft \exists X : [\perp.. \text{Object}]. \text{List} \triangleleft X$ is converted to $\{a \doteq x?, x? \triangleleft \text{Object}, \perp \triangleleft x?\}$. After applying (SWAP) and (SUBST-WC) on $a \doteq x?$ we get $\{a \triangleleft \text{Object}, \perp \triangleleft a\}$ and can now apply the (BOT) rule. This leaves us with $\{a \triangleleft \text{Object}\}$.

The **Unify** algorithm applies a capture conversion by applying the (CAPTURE) transformation when needed. Capture conversion removes a types bounding environment Δ and adds the included wildcard definitions to the global environment \mathbb{W} . Only the (CAPTURE) transformation adds wildcards to the wildcard environment \mathbb{W} and every added wildcard gets a fresh unique name. This ensures the wildcard environment \mathbb{W} never gets the same wildcard twice.

Unify must not replace normal type placeholders with free variables except variables initially

passed by Δ_{in} . The (SUBST) rule checks if a type T contains any free variables or wildcard placeholders before replacing a normal type placeholder with it. This ensures that a normal type placeholder is never replaced by a type containing free variables. A type solution for a normal type placeholder will never contain free variables. This is needed to guarantee well-formed type solutions and also keep free variables inside their scope (see challenge 3). (SUBST-WC) does not need to do that and can freely replace wildcard placeholders with types despite their free variables. We do not keep replacements for wildcard placeholders and they will not show up in the final type solution. If the (SUBST) rule is not applicable then either the (NORMALIZE) or (CONTRACT) transformation has to be used to remove wildcard placeholders and wildcards.

6.2 Branching Step (Step 2)

Unify is described as a nondeterministic algorithm. Some constraints allow for multiple transformations from which the algorithm has to pick the right one. There are also cases where there is more than one correct transformation and therefore more than one correct solution to the given input constraints. For that case still only one correct solution is returned by this specification of the algorithm. Our implementation of the algorithm considers this and tries every possible transformation option and gathers all possible type solutions. We skip the definition of this practice, because it is already described in [19] and only needed for a proof of completeness.

6.3 Generate Result (Step 4)

The generation rules defined in figure 31 are similar to the other transformation rules but contain an additional part, the result output consisting of a wildcard environment and a set of unifier σ .

6.4 Adding Wildcards to the mix

Wildcard types are added preemptively and if necessary can be removed later down the line. The parts where existential types are created are the (MATCH) and (GENERAL) transformations. Everytime a constraint of the form $T \triangleleft a$ occurs, it could be possible that a wildcard type for a is needed.

$$\begin{array}{c}
 \frac{\exists X : [l..u].\text{List}\langle\text{List}\langle X \rangle\rangle \triangleleft \text{List}\langle\text{List}\langle \underline{x} \rangle\rangle}{\exists X : [l..u].\text{List}\langle\text{List}\langle X \rangle\rangle \triangleleft^c \text{List}\langle\text{List}\langle \underline{x} \rangle\rangle} \text{ (PREPARE)} \\
 \frac{\exists X : [l..u].\text{List}\langle\text{List}\langle X \rangle\rangle \triangleleft^c \text{List}\langle\text{List}\langle \underline{x} \rangle\rangle}{X : [l..u] \vdash \text{List}\langle\text{List}\langle X \rangle\rangle \triangleleft \text{List}\langle\text{List}\langle \underline{x} \rangle\rangle} \text{ (CAPTURE)} \\
 \frac{X : [l..u] \vdash \text{List}\langle\text{List}\langle X \rangle\rangle \triangleleft \text{List}\langle\text{List}\langle \underline{x} \rangle\rangle}{X : [l..u] \vdash \text{List}\langle X \rangle \doteq \text{List}\langle \underline{x} \rangle} \text{ (REDUCE)} \\
 \frac{X : [l..u] \vdash \text{List}\langle X \rangle \doteq \text{List}\langle \underline{x} \rangle}{X : [l..u] \vdash \text{List}\langle X \rangle \triangleleft \text{List}\langle \underline{x} \rangle, \text{List}\langle \underline{x} \rangle \triangleleft \text{List}\langle X \rangle} \text{ (EQUALS)} \\
 \frac{X : [l..u] \vdash \text{List}\langle X \rangle \triangleleft \text{List}\langle \underline{x} \rangle, \text{List}\langle \underline{x} \rangle \triangleleft \text{List}\langle X \rangle}{X : [l..u] \vdash \underline{x} \doteq X} \text{ (REDUCE)} \\
 \frac{X : [l..u] \vdash \underline{x} \doteq X}{\text{equalize upper and lower bound of } X : \underline{x} \doteq u, u \doteq l} \text{ (CONTRACT)}
 \end{array}$$

Figure 16: Applying the transformation rules

$$\begin{array}{c}
 \frac{a \triangleleft^c \text{List}\langle \text{Integer} \rangle, a \triangleleft^c \text{List}\langle b \rangle}{a \triangleleft \exists A : [l..u].\text{List}\langle A \rangle,} \text{ (MATCH)} \\
 \frac{a \triangleleft \exists A : [l..u].\text{List}\langle A \rangle, \exists A : [l..u].\text{List}\langle A \rangle \triangleleft^c \text{List}\langle \text{Integer} \rangle, \exists A : [l..u].\text{List}\langle A \rangle \triangleleft^c \text{List}\langle b \rangle}{A : [l..u] \vdash a \triangleleft \exists A : [l..u].\text{List}\langle A \rangle, \text{List}\langle A \rangle \triangleleft \text{List}\langle \text{Integer} \rangle, \text{List}\langle A \rangle \triangleleft \text{List}\langle b \rangle} \text{ (CAPTURE)} \\
 \frac{A : [l..u] \vdash a \triangleleft \exists A : [l..u].\text{List}\langle A \rangle, \text{List}\langle A \rangle \triangleleft \text{List}\langle \text{Integer} \rangle, \text{List}\langle A \rangle \triangleleft \text{List}\langle b \rangle}{A : [l..u] \vdash a \triangleleft \exists A : [l..u].\text{List}\langle A \rangle, A \doteq \text{Integer}, A \doteq b} \text{ (REDUCE)} \\
 \frac{A : [l..u] \vdash a \triangleleft \exists A : [l..u].\text{List}\langle A \rangle, A \doteq \text{Integer}, A \doteq b}{A : [l..u] \vdash a \triangleleft \exists A : [l..u].\text{List}\langle A \rangle, u \doteq \text{Integer}, l \doteq \text{Integer}, u \doteq b} \text{ (TAME)}
 \end{array}$$

Figure 17: Applying the transformation rules

(MATCH)	$ \frac{\mathbb{W} \vdash C \cup \{a \triangleleft_1 \exists \Delta.D \langle \bar{T} \rangle, a \triangleleft_2 \exists \Delta'.D' \langle \bar{T}' \rangle\}}{\mathbb{W} \vdash C \cup \left\{ \begin{array}{l} a \triangleleft \exists A : [l..u].C \langle \bar{A} \rangle, \bar{l} \triangleleft \bar{u}, \\ \exists A : [l..u].C \langle \bar{A} \rangle \triangleleft_1 \exists \Delta.D \langle \bar{T} \rangle, \\ \exists A : [l..u].C \langle \bar{A} \rangle \triangleleft_2 \exists \Delta'.D' \langle \bar{T}' \rangle \end{array} \right\}} $	$ \begin{array}{l} \text{fresh } \bar{A} : [l..u] \\ C \ll D \\ C \ll D' \end{array} $
(REDUCE)	$ \frac{\mathbb{W} \vdash C \cup \{C \langle \bar{S} \rangle \triangleleft \exists A : [L..U].C \langle \bar{T} \rangle\}}{\mathbb{W} \vdash C \cup \{\bar{S} \doteq [\bar{a}_?/\bar{A}]\bar{T}, \bar{a}_? \triangleleft [\bar{a}_?/\bar{A}]\bar{U}, [\bar{a}_?/\bar{A}]\bar{L} \triangleleft \bar{a}_?\}} $	$ \bar{a}_? \text{ fresh} $
(CAPTURE)	$ \frac{\mathbb{W} \vdash C \cup \{\exists B : [L..U].C \langle \bar{S} \rangle \triangleleft^c T\}}{\mathbb{W} \cup C : [[\bar{C}/\bar{B}]\bar{L}..[\bar{C}/\bar{B}]\bar{U}] \vdash C \cup \{[\bar{C}/\bar{B}]C \langle \bar{S} \rangle \triangleleft T\}} $	$ \bar{C} \text{ fresh} $
(ADAPT)	$ \frac{\mathbb{W} \vdash C \cup \{\exists \Delta.C \langle \bar{T} \rangle \triangleleft \exists \Delta'.D' \langle \bar{T}' \rangle\}}{\mathbb{W} \vdash C \cup \{\exists \Delta.D \langle \bar{T}/\bar{X} \rangle \bar{S} \triangleleft \exists \Delta'.D' \langle \bar{T}' \rangle\}} $	$ \begin{array}{l} C \ll D' \\ \text{class } C \langle \bar{X} \rangle \triangleleft \bar{N} \triangleleft D \langle \bar{S} \rangle \end{array} $

Figure 18: Constraint reduce rules

(SUBST)	$\frac{\mathbb{W} \vdash C \cup \{a \doteq T\}}{[T/a] \mathbb{W} \vdash [T/a] C \cup \{a \doteq T\}}$	$\frac{a \notin T}{\text{fv}(T) \subseteq \Delta', \text{wtv}(T) = \emptyset}$
(SUBST-WC)	$\frac{\mathbb{W} \vdash C \cup \{a? \doteq T\}}{[T/a?] \mathbb{W} \vdash [T/a?] C}$	$a? \notin T$
(NORMALIZE)	$\frac{\mathbb{W} \vdash C \cup \{a \doteq T\}}{[b/b?] \mathbb{W} \vdash [b/b?] C \cup \{a \doteq [b/b?] T\}}$	$\frac{b? \in \text{wtv}(T)}{b \text{ fresh}}$
(CONTRACT)	$\frac{\mathbb{W} \cup \{A : [L..U]\} \vdash C \cup \{a \doteq T\}}{[U/A] \mathbb{W} \vdash [U/A] C \cup [U/A] \{a \doteq T, L \doteq U\}}$	$A \in \text{fv}(T)$

Figure 19: Substitution rules

For example the constraint $\text{List}\langle \text{String} \rangle \ll a$ results in $a \doteq \exists X : [l..u]. \text{List}\langle X \rangle$. The upper and lower bounds of the freshly generated wildcard X are type placeholders. If a second constraint like $a \ll \text{List}\langle \text{String} \rangle$ exists the wildcard X has to be removed by setting both lower and upper bound to String : 20

Let's have a look at the constraints generated by the introduction example in listing 1:

```
List<x> < r, String < x,
List<y> < r, Integer < y
```

Unify executes the following steps to generate a solved constraint set: 21.

The constraint $\text{String} \ll x$ is solved by applying (SUPER) and afterwards substituting String for x , same for $\text{Integer} \ll y$, resulting in the two constraints $\text{List}\langle \text{Integer} \rangle \ll r$ and $\text{List}\langle \text{String} \rangle \ll r$. Now comes the part where **Unify** creates existential types by applying the (SAME) transformation. An existential type is created with fresh type placeholders as upper and lower bound. After a substitution all r are replaced with this new existential type. **Unify** can now determine type replacements for the freshly created bounds u and l : 22.

In this example a correct solution is $\sigma(u) = \text{Object}$ and $\sigma(l) = \perp$. Remember that the substitution for the type placeholder r is $\exists X : [l..u]. \text{List}\langle X \rangle$ leading to $\text{List}\langle ? \rangle$ as a return type for the `someList` method after applying σ : 24

```
List<? extends Object> someList(){
    return new List("String") :? new List
        (42);
}
```

Figure 28 are special transformations aimed at free variables defined in the input environment Δ_{in} . Usually (UPPER) takes care of $X \ll a$ constraints, but if X is an element of Δ_{in} we can also treat it as a regular type. This leaves us with the two possibilities (SUBST-X) and (GEN-X) which is the same as (UPPER).

7 Related Work

Igarashi et al [5] define Featherweight Java and its generic sibling, Featherweight Generic Java. Their language is a functional calculus reduced to the bare essentials, they develop the full metatheory, they support generics, and study the type erasing transformation used by the Java compiler. Stadelmeier et. al. extends this approach by global type inference [19].

7.1 Wildcards in formal Java models

Wildcards are first described in a research paper in [22]. In [21] the Featherweight Java-Calculus Wild FJ is introduced. It contains a formal description of wildcards. The Java Language Specification [4] refers to Wild FJ for the introduction of wildcards. In [3] a formal model based of explicite existential types is introduced and proven as sound. Additionally, for a subset of Java a translation to the formal model is given, such that this subset is proven as sound. In [2] another core calculus is introduced, which is proven as sound, too. In this paper it is shown that the unsoundness of Java which is discovered in [1] is avoidable, even in the absence of nullness-aware type system. In [20] finally a framework is presented which combines use-site variance (wildcards as

$$\begin{array}{c}
\frac{\text{List}\langle\text{Integer}\rangle \triangleleft r}{\text{List}\langle\text{Integer}\rangle \triangleleft r, r \doteq \exists X : [\underline{l}..u].\text{List}\langle X \rangle} \text{(SAME)} \\
\frac{\text{List}\langle\text{Integer}\rangle \triangleleft r, r \doteq \exists X : [\underline{l}..u].\text{List}\langle X \rangle}{\text{List}\langle\text{Integer}\rangle \triangleleft \exists X : [\underline{l}..u].\text{List}\langle X \rangle} \text{(SUBST)} \\
\frac{\text{List}\langle\text{Integer}\rangle \triangleleft \exists X : [\underline{l}..u].\text{List}\langle X \rangle}{\text{Integer} \triangleleft \underline{u}, \underline{l} \triangleleft \text{Integer}} \text{(REDUCE)}
\end{array}$$

Figure 20: Applying the transformation rules

$$\begin{array}{c}
\frac{\text{String} \triangleleft \underline{x}}{\text{String} \triangleleft \underline{x}, \underline{x} \doteq \text{String}} \text{(SAME)} \quad \frac{\text{Integer} \triangleleft \underline{y}}{\text{Integer} \triangleleft \underline{y}, \underline{y} \doteq \text{Integer}} \text{(SAME)} \\
\frac{\text{String} \triangleleft \underline{x}, \underline{x} \doteq \text{String} \quad \text{Integer} \triangleleft \underline{y}, \underline{y} \doteq \text{Integer}}{\text{String} \triangleleft \text{String}, \text{Integer} \triangleleft \text{Integer}, \underline{x} \doteq \text{String}, \underline{y} \doteq \text{Integer}} \text{(SUBST)} \\
\frac{\text{String} \triangleleft \text{String}, \text{Integer} \triangleleft \text{Integer}, \underline{x} \doteq \text{String}, \underline{y} \doteq \text{Integer}}{\underline{x} \doteq \text{String}, \underline{y} \doteq \text{Integer}} \text{(REDUCE)}
\end{array}$$

Figure 21: Applying the transformation rules

$$\begin{array}{c}
\frac{\text{List}\langle\text{Integer}\rangle \triangleleft r, \text{List}\langle\text{Integer}\rangle \triangleleft r}{\text{List}\langle\text{String}\rangle \triangleleft r, \text{List}\langle\text{Integer}\rangle \triangleleft r, r \doteq \exists X : [\underline{l}..u].\text{List}\langle X \rangle} \text{(SAME)} \\
\frac{\text{List}\langle\text{String}\rangle \triangleleft r, \text{List}\langle\text{Integer}\rangle \triangleleft r, r \doteq \exists X : [\underline{l}..u].\text{List}\langle X \rangle}{\text{List}\langle\text{String}\rangle \triangleleft \exists X : [\underline{l}..u].\text{List}\langle X \rangle, \text{List}\langle\text{Integer}\rangle \triangleleft \exists X : [\underline{l}..u].\text{List}\langle X \rangle} \text{(SUBST)} \\
\frac{\text{List}\langle\text{String}\rangle \triangleleft \exists X : [\underline{l}..u].\text{List}\langle X \rangle, \text{List}\langle\text{Integer}\rangle \triangleleft \exists X : [\underline{l}..u].\text{List}\langle X \rangle}{\text{String} \triangleleft \underline{u}, \underline{l} \triangleleft \text{String}, \text{Integer} \triangleleft \underline{u}, \underline{l} \triangleleft \text{Integer}} \text{(REDUCE)}
\end{array}$$

Figure 22: Applying the transformation rules

(TAME)	$\frac{\mathbb{W} \cup \{A : [\underline{L}..U]\} \vdash C \cup \{A \doteq T\}}{\mathbb{W} \cup \{A : [\underline{L}..U]\} \vdash C \cup \{L \doteq T, U \doteq T\}}$	T is no wildcard placeholder
(EQUALS)	$\frac{\mathbb{W} \vdash C \cup \{\exists \Delta.N \doteq \exists \Delta'.N'\}}{\mathbb{W} \vdash C \cup \{\exists \Delta.N \triangleleft \exists \Delta'.N', \exists \Delta'.N' \triangleleft \exists \Delta.N\}}$	
(ERASE)	$\frac{\mathbb{W} \vdash C \cup \{T \doteq T\}}{\mathbb{W} \vdash C} \quad \frac{\mathbb{W} \vdash C \cup \{T \triangleleft T\}}{\mathbb{W} \vdash C}$	
(SWAP)	$\frac{\mathbb{W} \vdash C \cup \{G \doteq a\}}{\mathbb{W} \vdash C \cup \{a \doteq G\}} \quad \frac{\mathbb{W} \vdash C \cup \{T \doteq a?\}}{\mathbb{W} \vdash C \cup \{a? \doteq T\}} \quad \frac{\mathbb{W} \vdash C \cup \{N \doteq A\}}{\mathbb{W} \vdash C \cup \{A \doteq N\}} \quad \frac{\mathbb{W} \vdash C \cup \{a \doteq A\}}{\mathbb{W} \vdash C \cup \{A \doteq a\}}$	

Figure 23: Constraint normalize rules

$$\begin{array}{c}
 \text{CAPTURE} \\
 \frac{\mathbb{W} \vdash C \cup \{\overline{\exists B : [L..U].C \langle \bar{S} \rangle} \langle^c T\}\}}{\mathbb{W} \cup C : [\overline{[C/B]L..[C/B]U}] \vdash C \cup \{\overline{[C/B]C \langle \bar{S} \rangle} \langle T\}\}} \quad \bar{c} \text{ fresh} \\
 \\
 \text{REDUCE} \\
 \frac{\mathbb{W} \vdash C \cup \{\exists \Delta.C \langle \bar{S} \rangle \langle \overline{\exists A : [L..U].C \langle \bar{T} \rangle}\}\}}{\mathbb{W} \vdash C \cup \{\bar{S} \doteq [\overline{a_?/A}] \bar{T}, \overline{a_?} \langle [\overline{a_?/A}] \bar{U}, [\overline{a_?/A}] \bar{L} \langle \overline{a_?} \rangle\}} \quad \text{wtv}(\overline{\exists A : [L..U].C \langle \bar{T} \rangle}) = \emptyset \\
 \\
 \text{REDUCE-EMPTY} \\
 \frac{\mathbb{W} \vdash C \cup \{C \langle \bar{S} \rangle \langle \overline{\exists A : [L..U].C \langle \bar{T} \rangle}\}\}}{\mathbb{W} \vdash C \cup \{\bar{S} \doteq [\overline{a_?/A}] \bar{T}, \overline{a_?} \langle [\overline{a_?/A}] \bar{U}, [\overline{a_?/A}] \bar{L} \langle \overline{a_?} \rangle\}} \\
 \\
 \text{EXCLUDE} \\
 \frac{\mathbb{W} \vdash C \cup \{\exists \Delta.C \langle \bar{S} \rangle \langle T\}\}}{[a/a_?] \mathbb{W} \vdash [a/a_?] C \cup [a/a_?] \{\exists \Delta.C \langle \bar{S} \rangle \langle T\}\}} \quad \Delta \neq \emptyset, a_? \in \text{fv}(T), a \text{ fresh}
 \end{array}$$

Figure 24: Applying the transformation rules

$$\text{(CIRCLE)} \frac{\mathbb{W} \vdash C \cup \{a_1 \leq a_2, a_2 \leq a_3, \dots, a_n \leq a_1\}}{\mathbb{W} \vdash C \cup \{a_1 \doteq a_2, a_2 \doteq a_3, \dots, a_n \doteq a_1\}} \quad n > 0$$

Figure 25: Rules for normal placeholders

If we find an illicit constraint assigning a type containing free variables to a type placeholder not flagged as a wildcard placeholder the algorithm fails.

$\{a \doteq N\} \in C$ with $\text{fv}(N) \cap \Delta_{in} \neq \emptyset \implies \text{fail!}$

Figure 26: Fail conditions

$$\begin{array}{c}
 \text{(SAME)} \quad \frac{\mathbb{W} \vdash C \cup G \leq a}{\mathbb{W} \vdash C \cup \{a \doteq G\}} \\
 \text{-----} \\
 \text{(GENERAL)} \quad \frac{\mathbb{W} \vdash C \cup \exists \Delta.C \langle \bar{T} \rangle \leq a}{\mathbb{W} \vdash C \cup \{\exists \Delta.C \langle \bar{T} \rangle \leq \underline{a}, \underline{a} \doteq \exists X : [l..u].C \langle \bar{X} \rangle, \bar{u} \leq \bar{S}\}} \quad \begin{array}{l} \text{class } C \langle \bar{X} \rangle \langle D \langle \bar{N} \rangle \\ \text{fresh } X : [l..u] \end{array} \\
 \text{-----} \\
 \text{(SUPER)} \quad \frac{\mathbb{W} \vdash C \cup \exists \Delta.C \langle \bar{T} \rangle \leq a}{\mathbb{W} \vdash C \cup \{\exists \Delta'.D \langle [\bar{T}/\bar{X}] \bar{N} \rangle \leq a\}} \quad \begin{array}{l} \text{class } C \langle \bar{X} \rangle \langle D \langle \bar{N} \rangle \\ \bar{X} \notin \mathbb{W} \cup \Delta, \Delta' = \Delta \cap \text{fv}([\bar{T}/\bar{X}] \bar{N}) \end{array}
 \end{array}$$

Figure 27: Step 2 branching: Multiple rules can be applied to the same constraint

$$\begin{array}{c}
 \text{(SUBST-X)} \quad \frac{\mathbb{W} \cup \{X : [L..U]\} \vdash C \cup X \leq a}{\mathbb{W} \cup \{X : [L..U]\} \vdash C \cup \{a \doteq X\}} \quad X \in \Delta_{in} \\
 \text{-----} \\
 \text{(GEN-X)} \quad \frac{\mathbb{W} \cup \{X : [L..U]\} \vdash C \cup X \leq a}{\mathbb{W} \cup \{X : [L..U]\} \vdash C \cup \{U \leq a\}}
 \end{array}$$

Figure 28: Step 2 branching: Multiple rules can be applied to the same constraint

$$\begin{array}{c}
 \text{(SETTLE)} \frac{\mathbb{W} \vdash C \cup \{a \triangleleft N, a \triangleleft_1 b\}}{\mathbb{W} \vdash C \cup \{a \triangleleft_1 b, b \triangleleft N\}} \\
 \text{-----} \\
 \text{(RAISE)} \frac{\mathbb{W} \vdash C \cup \{a \triangleleft_1 N, a \triangleleft b\}}{\mathbb{W} \vdash C \cup \{a \triangleleft_1 N, N \triangleleft b\}}
 \end{array}$$

Figure 29: Step 2 branching: Multiple rules can be applied to the same constraint

$$\begin{array}{c}
 \text{(SUBELIM)} \frac{\mathbb{W} \vdash C \cup \{a \triangleleft b\}}{[a/b]\mathbb{W} \vdash [a/b]C \cup \{b \doteq a\}} \\
 \text{(GROUND)} \frac{\mathbb{W} \cup \{\mathbf{X} : [a..U]\} \vdash C \cup \{a \triangleleft T\}}{\mathbb{W} \cup \{\mathbf{X} : [a..U]\} \vdash C \cup \{a \doteq \perp\}} \quad a \notin C, a \notin \overline{T}, a \notin \mathbb{W} \\
 \text{(FLATTEN)} \frac{\mathbb{W} \cup \{\mathbf{X} : [L..U]\} \vdash C \cup \{a \triangleleft T\}}{[U/X]\mathbb{W} \vdash [U/X]C \cup [U/X]\{a \triangleleft T, U \doteq L\}} \quad X \in \text{fv}(T)
 \end{array}$$

Figure 30: Cleanup rules

$$\begin{array}{c}
 \text{(GENDELTA)} \frac{\mathbb{W} \vdash C \cup \{\underline{b} \triangleleft T\} \implies \Delta, \sigma}{\mathbb{W} \vdash [B/\underline{b}]C \implies \Delta \cup \{B : [\perp..T]\}, \sigma \cup \{\underline{b} \rightarrow B\}} \quad \begin{array}{l} \text{tph}(T) = \emptyset, \text{fv}(T) \subseteq \Delta \cup \Delta_{in} \\ B \text{ fresh}, \underline{b} \notin \text{dom}(\sigma), \Delta, \Delta_{in} \vdash T \text{ ok} \end{array} \\
 \text{(GENDELTA')} \frac{\mathbb{W} \vdash C \cup \{b_? \triangleleft T\} \implies \Delta, \sigma}{\mathbb{W} \vdash [B/\underline{b}]C \implies \Delta \cup \{B : [\perp..T]\}, \sigma} \quad \begin{array}{l} \text{tph}(T) = \emptyset, \text{fv}(T) \subseteq \Delta \cup \Delta_{in} \\ B \text{ fresh}, \Delta, \Delta_{in} \vdash T \text{ ok} \end{array} \\
 \text{(GENSIGMA)} \frac{\mathbb{W} \vdash C \cup \{\underline{a} \doteq T\} \implies \Delta, \sigma}{\mathbb{W} \vdash C \implies \Delta, \sigma \cup \{\underline{a} \rightarrow T\}} \quad \begin{array}{l} \text{tph}(T) = \emptyset \\ \underline{a} \notin \text{dom}(\sigma), \Delta, \Delta_{in} \vdash T \text{ ok} \end{array}
 \end{array}$$

Figure 31: Generate result

in Java) and definition-site variance (as in Scala). For instance, it can be used to add use-site variance to Scala and extend the Java type system to infer the definition-site variance.

Our calculus is mixture ...

7.2 Type inference

Some object-oriented languages like Scala, C#, and Java perform *local* type inference [8, 9]. Local type inference means that missing type annotations are recovered using only information from adjacent nodes in the syntax tree without long distance constraints. For instance, the type of a variable initialized with a non-functional expression or the return type of a method can be inferred. However, method argument types, in particular for recursive methods, cannot be inferred by local type inference.

Milner's algorithm \mathcal{W} [7] is the gold standard for global type inference for languages with parametric polymorphism, which is used by ML-style languages. The fundamental idea of the algorithm is to enforce type equality by many-sorted type unification [6, 16]. This approach is effective and results in so-called principal types because many-sorted unification is unitary, which means that there is at most one most general result.

Plümicke [13] presents a first attempt to adopt Milner's approach to Java. However, the presence of subtyping means that type unification is no longer unitary, but still finitary. Thus, there is no longer a single most general type, but any type is an instance of a finite set of maximal types (for more details see Section ??). Further work by the same author [12, 15], refines this approach by moving to a constraint-based algorithm and by considering lambda expressions and Scale-like function types.

Pluemicke has a different approach to introduce wildcards in [11]. He allows wildcards as any substitution for type variables and disclaim the capture conversion. Instead he extended the subtyping ordering such that for $\theta <: \theta' <: \theta''$ holds indeed the transitive closure of $? \text{ extends } \theta <: \theta'$ and $\theta' <: ? \text{ super } \theta''$ but not the reflexive closure. He gave a type unification algorithm for this type system, which he proved as sound and complete.

The problem of his type system is in the losing reflexivity as shown in example 1. First approaches to solve this problem he gave in [10], where he fixes that no pairwise different nodes in

the abstract syntax gets the same type variable and that no pairwise different type variables are equalized. In [14] he showed how his type inference algorithm suffices these properties.

8 Discussion

We couldn't verify it with an implementation yet, but we assume at least the same functionality as the global type inference algorithm for Featherweight Java without Wildcards [19].

When it comes to wildcards there is a limitation we couldn't find a good workaround:

Example 2. *This example does not work:*

```
class Example{
  <A> Pair<A,A> make(List<A> l){...}
  <A> bool compare(Pair<A,A> p){...}

  bool test(List<?> l){
    return compare(make(l));
  }
}
```

```
 $\exists A : [\perp..Object].List<A> < List<x?> ,$ 
 $Pair<x?, x?> < \tau ,$ 
 $\tau < Pair<z?, z?>$ 
```

We will not infer intermediate types like $\exists X.Pair<X,X>$ for a normal type placeholder. τ will get the type $\exists X, Y.Pair<X, Y>$ which renders the constraint set unsolvable.

x will get the type $\exists X.List<X>$ and from the constraint $\exists X.List<X> < List<a?>$ **Unify** deduces $a? \doteq X$ leading to $Pair<X, X> < m$.

Finding a supertype to $Pair<X, X>$ is the crucial part. The correct substitution for τ would be $\exists X.Pair<X, X>$. But this leads to additional branching inside the **Unify** algorithm and increases runtime. This also just solves this specific issue and not the problem in general.

9 Conclusion and Further Work

The problems introduced in the opening 4.2 can be solved via our **Unify** algorithm. Additionally we could prove that type solutions generated by our type inference algorithm are correct respective to **TamedFJ's** type rules. The important parts are lemma A.1 and A.8. They prove that the **Unify** algorithm solves a constraint set according to our

subtyping rules and that the generated constraints match **TamedFJ**'s type system.

The crucial parts introduced in this paper are capture constraints, wildcard placeholders and existential types. Those data structures allow us to solve the problems introduced in chapter 4.2.

Our algorithm is designed for extensibility with the final goal of full support for Java. **Unify** is the core of the algorithm and can be used for any calculus sharing the same subtype relations as depicted in 3. Additional language constructs can be added by implementing the respective constraint generation functions in the same fashion as described in chapter 5.

A Soundness

The differentiation of wildcard placeholders and normal type placeholders is vital for the soundness proof. During a let statement the environment Δ is extended by capture converted wildcards, but only for the scope of the body of the let statement. The capture converted wildcards must not be used outside of the let statement. This is ensured by two things: The first is lemma A.10 which ensures that free variables only travel one hop at the time through a constraint set. And the second one is the fact that normal type placeholders never contain free variables.

Unify calculates solutions for all normal type placeholders. Those are used for all untyped method's argument and return type. A correct typing for method calls can be deduced from those type informations.

Lemma A.1. *Soundness: **Unify**'s type solutions for a constraint set generated by **TYPEExpr** are correct.*

*if **TYPEExpr**(Π, e, a) = (Δ', C)*

then $\Delta|\Gamma \vdash e : \sigma(a)$ where $\Delta = \Delta_u \cup \Delta'$

Proof: By structural induction over the expression e .

let $x = t_1$ in t_2 $\text{dom}(\Delta') \subseteq \text{fv}(N)$ by lemma A.6. $\Delta \vdash T, \exists \Delta'. N$ ok by lemma A.5. $\Delta|\Gamma \vdash t_1 : \sigma(e_1)$ by assumption. $\Delta \vdash \sigma(e_1) <: \exists \Delta'. N$ by constraint $e_1 \leq x$ and lemma A.8. The body of the let statement will only use the free variables provided by Δ and Δ' . We can prove this by lemma A.7 and the fact

that the wildcard placeholders used generated the body of the let statement are not used outside of it. Therefore we can say $\Delta, \Delta'|\Gamma, x : N \vdash t_2 : \sigma(e_2)$ by assumption and $\Delta, \Delta' \vdash \sigma(e_2) <: \sigma(a)$ by lemma A.8 given the constraint $e_2 \leq a$.

let $x = t_1$ in $x.f$ The let statement in the input is untyped and we have to create a let statement $\text{let } x : \exists \Delta'. N = t_1 \text{ in } x.f$ that suffices the T-Let and T-Field type rules. The case where no capture conversion is needed, because $\Delta' = \emptyset$, is trivial. Here the Let statement can be skipped entirely. We investigate the case $\sigma(x) = \exists \Delta'. N$. Let $T_1 = \exists \Delta'. N = \sigma(x)$, $\sigma(t_1) = T_1$, $\sigma(a) = T_2$ then

- $\Delta|\Gamma \vdash t_1 : T_1$ by assumption
- $\Delta \vdash T_1 <: \exists \Delta'. N$ by constraint $t_1 \leq x$ and lemma A.8
- $\Delta, \Delta'|\Gamma, x : N \vdash x.f_1 : T_2$ First we can say $\Delta|\Gamma \vdash x : N$ by T-Var. By lemma A.5 and WF-Var we can deduct $\text{fv}(\exists \Delta'. N) \subseteq \Delta$ and by constraint $x \leq^c C \langle \bar{a}_? \rangle$ and lemmas A.8 and A.10 we can finally say $\Delta, \Delta' \vdash N <: \sigma(C \langle \bar{a}_? \rangle)$. With the constraint $a \doteq [\bar{a}_?/\bar{X}]T$ and $\sigma([\bar{a}_?/\bar{X}]T) \in \text{fields}(\sigma(C \langle \bar{a}_? \rangle))$ by F-Class we proof $\Delta, \Delta'|\Gamma, x : N \vdash x.f_1 : T_2$.
- $\Delta, \Delta' \vdash T_2 <: T$ by constraint

We are allowed to use capture conversion for v here. $\Delta \vdash v : \sigma(a)$ by assumption. $\Delta \vdash \sigma(a) <: \sigma(C \langle \bar{a}_? \rangle)$ and $\Delta \vdash U_i <: \sigma(a)$, because of the constraints $[\bar{a}_?/\bar{X}]T \leq a$, $r \leq^c C \langle \bar{a}_? \rangle$ and lemma A.8. $\text{fields}(\sigma(C \langle \bar{a}_? \rangle)) = \sigma([\bar{a}_?/\bar{X}]T)$.

let $x = e$ in let $\bar{x} = \bar{e}$ in $x.m(\bar{x})$ generates constraints $e \leq x$, $\bar{e} \leq \bar{x}$, $r \leq a$, $\bar{x} \leq^c \bar{T}$, $T \leq r$, $\bar{b}_? \leq \bar{N}$. We omit the case where a capture conversion is not needed and assume $\sigma(x) = \exists \Delta'. N$, $\sigma(\bar{x}) = \exists \Delta'. \bar{N}$. We have to show T-Let and T-Call which leaves us with:

- $\Delta|\Gamma \vdash e : \sigma(e)$ and $\Delta|\Gamma \vdash \bar{e} : \sigma(\bar{e})$ by assumption
- $\Delta \vdash T_1 <: \exists \Delta'. N$ and $\Delta \vdash \bar{T}_1 <: \exists \Delta'. \bar{N}$ by constraints $e \leq x$, $\bar{e} \leq \bar{x}$ and lemma A.8
- $\Delta, \Delta', \bar{\Delta}'|\Gamma, x : N, \bar{x} : \bar{N} \vdash x.m(\bar{x}) : \sigma(r)$ by T-Call and lemma A.3, because the following promises are satisfied. Note: The lemma A.3 and the fact that the wildcard placeholders $\bar{b}_?$ are solely used here guarantees that no other than the

variables declared in $\Delta, \Delta', \overline{\Delta}$ appear in $\sigma'(b?)$.

- $\langle \overline{x} \triangleleft \overline{u}' \rangle \overline{u} \rightarrow U \in \Pi(m)$ is given, otherwise the program fails at the constraint generation step.
- $\Delta, \Delta', \overline{\Delta} \vdash \overline{s} \triangleleft : [\overline{s}/\overline{x}]\overline{u}'$ by constraints $\overline{b?} \triangleleft [\overline{b?}/\overline{x}]\overline{N}$ and lemma A.8 there must be some \overline{s} satisfying this premise.
-

$v.m(\overline{v})$ Proof is analog to field access, except the $\Delta \vdash \overline{s} \text{ ok}$ premise. We know that $\text{Unify}(\Delta, [\overline{b?}/\overline{y}]\{\overline{e} \triangleleft^c \overline{T}, \overline{T} \triangleleft a, \overline{Y} \triangleleft \overline{N}\} \cup C) = (\Delta', \sigma)$ and if we assume $\sigma(\overline{e}) = \exists \Delta.N'$ then the call to $\text{Unify}(\Delta \cup \overline{\Delta}, [\overline{b?}/\overline{y}]\{\overline{N}' \triangleleft \overline{T}, \overline{T} \triangleleft a, \overline{Y} \triangleleft \overline{N}\} \cup C)$ succeeds with \overline{b} being normal placeholders this time, which proves $\Delta \vdash \overline{s} \text{ ok}$ via lemma A.5.

During the unification process free variables can only move one hop at a time. Free variables originate from capture constraints. By applying the capture rule to a constraint of the form $\exists \Delta.N \triangleleft^c T$ the variables declared in Δ are now free in the constraint $N \triangleleft T$. The other way of free variables traveling into a constraint is by substitution. This is only possible if a constraint contains wildcard placeholders. And only free variables which reside in the same constraint as a wildcard placeholders have the possibility of being set in.

This effect is transitive. Free variables travel from constraint to constraint. If there is no connection of constraints containing wildcard type placeholders between a free variable and a constraint, then free variables cannot travel there.

Lemma A.2. *Free variables travel only*

Lemma A.3. *Free variables never leave their scope. If a set of constraints does not share a single wildcard type placeholder with the rest of the constraint set, then it will never contain free variables used outside of it. $(\Delta, \sigma) = \text{Unify}(\Delta_{in}, C \cup C')$*

and $\text{wtv}(C') \cap \text{wtv}(C) = \emptyset$ and $\overline{\sigma(x)} = \exists \Delta.N$

then $\Delta, \Delta', \overline{\Delta}$ are all the variables used in C' .

Proof: This is due to free variables only travel in-between constraints and themselves via substitution.

If $S \triangleleft^c T$ spawns free variables, they will be contained within the reach of the wildcard type placeholders used for the method call. No other free variables can move into that scope. With this lemma we can further proof that during a method call only the generated free variables are used.

Lemma A.4. *If $(\Delta, \sigma) = \text{Unify}(\Delta', C)$*

Then $\sigma(a) = \exists \Delta''.N \implies \text{dom}(\Delta'') \subseteq \text{fv}(N)$

Proof: Easy - All types given in the input to the **Unify** algorithm already comply with this requirement and none of the rules change. Note: the (SUPER) rule removes unnecessary wildcards.

Lemma A.5. *Unify generates well-formed types as long as well-formed types are supplied.*

If $(\Delta, \sigma) = \text{Unify}(\Delta', C)$

and $\exists \Delta.N \in C \implies \Delta' \vdash \exists \Delta.N \text{ ok}$

then $\Delta \vdash \sigma(a) \text{ ok}$

Proof: by induction over every step of the **Unify** algorithm. Hint: a type placeholder a will never be replaced by a free variable or a type containing free variables. This fact together with the presumption that every supplied type is well-formed we can easily show that this lemma is true.

Lemma A.6. *Unify generates well-formed existential types*

If $(\Delta, \sigma) = \text{Unify}(\Delta', C)$

and $\forall \exists \Delta''.N \in C : \text{dom}(\Delta'') \subseteq \text{fv}(N)$

and $\sigma(a) = \exists \Delta'.N$

then $\text{dom}(\Delta') \subseteq \text{fv}(N)$

Proof is straightforward induction over the transformation rules of **Unify** under the assumption that all supplied existential types are satisfying the premise. All parts of **Unify** that generate wildcard types always spawn types $\exists \Delta'.N$ with $\text{dom}(\Delta') \subseteq \text{fv}(N)$. Only the (ADAPT) rule is able to produce types neglecting the premise, but is immediately corrected by the (TRIM) rule.

Lemma A.7. *Unify does not add free variables to types not containing free variables.*

If $(\sigma, \Delta) = \text{Unify}(\Delta', C)$

and a being a type placeholders used in C

then $\text{fv}(\sigma(a)) \subseteq \Delta, \Delta'$

Proof: Trivial. **Unify** fails when a constraint $a \doteq x$ arises.

Lemma A.8. Unify Soundness: *Unify's type solutions are correct respective to the subtyping rules defined in figure 3.*

If $(\sigma, \Delta) = \text{Unify}(\Delta', \overline{S} \ll \overline{T} \cup \overline{S'} \ll^c \overline{T'})$

Then there exists a substitution σ' with:

- $\sigma \subseteq \sigma'$
- $\Delta, \Delta' \vdash \overline{\sigma'(S)} \ll: \sigma'(T)$
- $\Delta, \Delta' \vdash \overline{\sigma'(S')} \ll: \exists \Delta.N$
- $\Delta, \Delta', \overline{\Delta} \vdash \overline{N} \ll: \sigma'(T')$

Lemma A.9. Unify Soundness: *Unify's type solutions are correct respective to the subtyping rules defined in figure 3.*

If $(\sigma, \Delta) = \text{Unify}(\Delta', \overline{S} \ll \overline{T} \cup \overline{S'} \ll^c \overline{T'})$ with $\text{wfv}(\overline{S}) = \emptyset$ and $\text{wfv}(\overline{T}) = \emptyset$

Then $\Delta, \Delta' \vdash \overline{\sigma(S)} \ll: \sigma(T)$

and either $\Delta, \Delta' \vdash \overline{\sigma(S')} \ll: \sigma(T')$ or $\Delta, \Delta', \overline{\Delta} \vdash \overline{\sigma'(S')} \ll: \sigma(T')$ with the capture converted substitution $\sigma' = \{a \mapsto N \mid a \mapsto \exists \Delta.N \in \sigma\}$ and $\overline{\Delta} = \{\Delta \mid a \mapsto \exists \Delta.N \in \sigma\}$

Proof:

For every step in the **Unify** algorithm: Assuming the unifier σ is correct for a constraint set C' , the unifier is also correct for the constraint set C before the transformation.

Unify terminates with $C = \emptyset$ for which the preposition holds: $\Delta \vdash \sigma(\emptyset)$

We now show that for every transformation of a constraint set C to a constraint set C' the preposition holds for C using the assumption that it holds for $C' : \Delta \vdash \sigma(C') \implies \Delta \vdash \sigma(C)$

AddDelta C is not changed

GenDelta by definition, S-Var-Left, and S-Trans

GenDelta' same as GenDelta by setting $\sigma'(b?) = \text{B}$

GenSigma by definition and S-Refl.

Ground Assumption and S-Bot

Sub-Elim Assumption and S-Refl

Force by assumption and $X = U$

Raise Assumption, S-Trans

Settle Assumption, S-Trans

Super S-Extends ($\vdash \exists \Delta.C \ll \overline{T} \ll: \exists \Delta.D \ll \overline{[T/X]N}$), S-Trans

General by Assumption, because $C \subset C'$

Same by S-Exists

SameW

Adapt Assumption, S-Extends, S-Trans

Adopt Assumption, because $C \subset C'$

Prepare Given a $\exists \Delta_c.C \ll \overline{S} \ll: \exists \Delta''.N$ we get $\Delta, \Delta', \overline{\Delta} \vdash C \ll \overline{S} \ll: \exists \Delta''.N$ with $\Delta_c \subseteq \overline{\Delta}$.

$\text{fv}(\sigma'(\exists \Delta_c.C \ll \overline{S})) = \emptyset$ implies $\text{fv}(\overline{S}) \subseteq \text{dom}(\Delta, \Delta_c)$ and $\text{fv}(\sigma'(\exists \Delta''.N)) = \emptyset$ implies $\text{dom}(\Delta_c) \cap \text{fv}(\sigma'(\exists \Delta''.N)) = \emptyset$.

No free variables on both sides also mean we do not need $\overline{\Delta}$. Therefore we can say $\Delta, \Delta' \vdash \exists \Delta_c.C \ll \overline{S} \ll: \exists \Delta''.N$.

Capture Everytime the (CAPTURE) rule is invoked we add the freshly generated free variables to the global environment \mathbb{W} . We get a σ' with $\Delta, \Delta' \vdash \sigma'(\overline{[C/B]C \ll \overline{S}}) \ll: \sigma'(\exists \Delta.C \ll \overline{T})$ where $\sigma'(\overline{C} : \overline{[L..U]}) \subseteq \Delta'$ by assumption. **Unify** performs a capture conversion only on \ll^c constraints. Therefore we can say that $\Delta, \Delta', \overline{\Delta} \vdash \sigma'(C \ll \overline{S}) \ll: \sigma'(\exists \Delta.C \ll \overline{T})$ with $\overline{\Delta}$ being all the fresh wildcards generated by (CAPTURE).

Reduce To proof $\exists \Delta.C \ll \overline{S} \ll: \exists \overline{X} : \overline{[L..U]}.C \ll \overline{T}$ we have to show S-Exists for some Δ'' and $\overline{T} = \sigma(\overline{a?})$:

- $\Delta'' \vdash \overline{[T/X]L} \ll: \overline{T}$ by assumption and $\overline{[a?/X]L} \ll \overline{a?}$
- $\Delta'' \vdash \overline{T} \ll: \overline{[T/X]U}$ by assumption and $\overline{a?} \ll \overline{[a?/X]L}$
- $\text{fv}(\overline{T}) \subseteq \text{dom}(\Delta'', \Delta')$ by setting Δ'' accordingly

If $\text{fv}(C \ll \overline{S}) = \text{fv}(\exists A : \overline{[L..U]}.C \ll \overline{T}) = \emptyset$ the preposition holds by Assumption and S-Exists. Otherwise $\Delta' \vdash \text{CC}(\sigma(C \ll \overline{S})) \ll: \sigma(\exists A : \overline{[L..U]}.C \ll \overline{T})$ holds with any Δ' so that $(\text{fv}(C \ll \overline{S}) \cup \text{fv}(\exists A : \overline{[L..U]}.C \ll \overline{T})) \subseteq \text{dom}(\Delta')$.

Match Assumption, S-Trans

Trim Assumption and S-Exists

Remove C is not changed

Circle S-Refl

Swap by definition

Erase S-Refl

Equals by definition 6.1

Normalize Assumption and lemma 5 *substitution preserves subtyping*. The GenSigma and GenDelta steps remove Wildcards which have the same upper and lower bound. $A, B \notin \sigma(C)$

```
<A> A m(List<? extends List<A>> l, A)
m(List<List<? super String>> l, "hi")
```

Tame same reasoning as Normalize

Bot S-Bot

Pit S-Bot

Upper S-Trans and S-VarLeft

Lower S-Trans and S-VerRight

Subst-WC by S-Refl

Subst $\sigma(C \cup \{a \doteq T\}) = \sigma([T/a]C \cup \{a \doteq T\})$ and
 $\sigma(W) = \sigma([T/a]W)$

Subst-WC Same as Subst

Lemma A.10. *A constraint $a \leq^c T$ or $a \leq T$ implies that $fv(\sigma(T)) \subseteq fv(\sigma(a))$. Only free variables, which are part of the left side are used on the right side.*

B Additional Functions

tph returns all type placeholders inside a given type.

Example: $\text{tph}(\exists X : [\perp..a].\text{Pair}\langle b?, X \rangle) = \{a, b?\}$

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: 32 ($[\bar{C}/\bar{B}]$ in the (REDUCE) rule)

Free Variables: The fv function assumes every wildcard type variable to be a free variable as well.

$$\begin{aligned}fv(A) &= \{A\} \\fv(a) &= \emptyset \\fv(\exists \Delta.C \langle \bar{T} \rangle) &= \{fv(T) \mid T \in \bar{T}\} / \text{dom}(\Delta)\end{aligned}$$

Fresh Wildcards: $\text{fresh } \bar{A} : [\bar{l}..u]$ generates fresh wildcards. The new names \bar{A} are fresh, as well as the type variables \bar{u} and \bar{l} , which are used for the upper and lower bounds.

$$\begin{aligned}[\bar{A}/\bar{B}]\bar{B} &= \bar{A} \\[\bar{A}/\bar{B}]\bar{C} &= \bar{C} && \text{if } \bar{B} \neq \bar{C} \\[\bar{A}/\bar{B}]\exists X : [\bar{L}..U].N &= \exists X : [[\bar{A}/\bar{B}]\bar{L}..[\bar{A}/\bar{B}]\bar{U}].[\bar{A}/\bar{B}]N && \text{if } \bar{B} \notin \bar{X} \\[\bar{A}/\bar{B}]\exists X : [\bar{L}..U].N &= \exists X : [\bar{L}..U].N && \text{if } \bar{B} \in \bar{X}\end{aligned}$$

Figure 32: Alpha conversion

References

- [1] Nada Amin and Ross Tate. “Java and scala’s type systems are unsound: the existential crisis of null pointers”. In: Oct. 2016, pp. 838–848.
- [2] Kevin Bierhoff. “Wildcards Need Witness Protection”. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). URL: <https://doi.org/10.1145/3563301>.
- [3] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. “A Model for Java with Wildcards”. In: *ECOOP 2008 – Object-Oriented Programming*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 2–26. ISBN: 978-3-540-70592-5.
- [4] James Gosling et al. *The Java® Language Specification*. Java SE 21. 2023. URL: <https://docs.oracle.com/javase/specs/jls/se21/jls21.pdf>.
- [5] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. “Featherweight Java: a minimal core calculus for Java and GJ”. In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450. ISSN: 0164-0925. URL: <https://doi.org/10.1145/503502.503505>.
- [6] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: *ACM Trans. Program. Lang. Syst.* 4.2 (Apr. 1982), pp. 258–282. ISSN: 0164-0925. URL: <https://doi.org/10.1145/357162.357169>.
- [7] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and Systems Sciences* 17.3 (1978), pp. 348–375. URL: [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4).

- [8] Martin Odersky, Matthias Zenger, and Christoph Zenger. “Colored local type inference”. In: *Proc. 28th ACM Symposium on Principles of Programming Languages* 36.3 (2001), pp. 41–53.
- [9] Benjamin C. Pierce and David N. Turner. “Local type inference”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’98. San Diego, California, United States, 1998, pp. 252–265.
- [10] Martin Plümicke. “Avoiding the Capture Conversion in Java–TX”. In: *Proceedings of the 38th and 39th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts*. INSIGHTS – Schriftenreihe der Fakultät Technik 01/2024. 2023, pp. 109–115. URL: <https://www.dhbw-stuttgart.de/forschung-transfer/technik/schriftenreihe-insights>.
- [11] Martin Plümicke. “Java type unification with wildcards”. In: *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*. Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf. Vol. 5437. Lecture Notes in Artificial Intelligence. Springer-Verlag Heidelberg, 2009, pp. 223–240.
- [12] Martin Plümicke. “More Type Inference in Java 8”. In: *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. Ed. by Andrei Voronkov and Irina Virbitskaite. Vol. 8974. Lecture Notes in Computer Science. Springer, 2015, pp. 248–256.
- [13] Martin Plümicke. “Typeless Programming in Java 5.0 with Wildcards”. In: *5th International Conference on Principles and Practices of Programming in Java*. Ed. by Vasco Amaral et al. Vol. 272. ACM International Conference Proceeding Series. Sept. 2007, pp. 73–82.
- [14] Martin Plümicke and Daniel Holle. “Principal generics in Java–TX”. In: *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. Ed. by Thomas Noll and Ira Justus Fesefeldt. Aachener Informatik-Berichte (AIB) AIB-2023-03. RWTH Aachen. Aachen, Sept. 2023, pp. 122–143. URL: <https://publications.rwth-aachen.de/record/972197/files/972197.pdf#%5B%7B%22num%22%3A281%2C%22gen%22%3A0%7D%2C%7B%22name%22%3A%22XYZ%22%7D%2C89.292%2C740.862%2Cnu11%5D>.
- [15] Martin Plümicke and Andreas Stadelmeier. “Introducing Scala-like Function Types into Java-TX”. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: ACM, 2017, pp. 23–34. ISBN: 978-1-4503-5340-3.
- [16] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *Journal of ACM* 12(1) (Jan. 1965), pp. 23–41.
- [17] Amr Sabry and Matthias Felleisen. “Reasoning about programs in continuation-passing style.” In: *ACM SIGPLAN Lisp Pointers* 1 (1992), pp. 288–298.
- [18] Daniel Smith and Robert Cartwright. “Java type inference is broken: can we fix it?” In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA ’08. Nashville, TN, USA: Association for Computing Machinery, 2008, pp. 505–524. ISBN: 9781605582153. URL: <https://doi.org/10.1145/1449764.1449804>.
- [19] Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. “Global Type Inference for Featherweight Generic Java”. In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 28:1–28:27. ISBN: 978-3-95977-225-9. URL: <https://drops.dagstuhl.de/opus/volltexte/2022/16256>.
- [20] Ross Tate, Alan Leung, and Sorin Lerner. “Taming wildcards in Java’s type system”. In: *SIGPLAN Not.* 46.6 (June 2011), pp. 614–627. ISSN: 0362-1340. URL: <https://doi.org/10.1145/1993316.1993570>.
- [21] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. “Wild FJ”. In: *Proceedings of FOOL 12*. Ed. by Philip Wadler. ACM. Long Beach, California, USA: School of Informatics, University of Edinburgh, Jan. 2005. URL: <http://homepages.inf.ed.ac.uk/wadler/fool/>.

- [22] Mads Torgersen et al. “Adding wildcards to the Java programming language”. In: *Proceedings of the 2004 ACM symposium on Applied computing*. 2004, pp. 1289–1296.
- [23] Joe B Wells. “Typability and type checking in System F are equivalent and undecidable”. In: *Annals of Pure and Applied Logic* 98.1-3 (1999), pp. 111–156.

Towards Tagless Interpretation of Stratified System F

Peter Thiemann
University of Freiburg
thiemann@acm.org

Marius Weidner
University of Freiburg
weidner@cs.uni-freiburg.de

Abstract

We explore the definition of an intrinsically typed interpreter for stratified System F in Agda.

1 Introduction

Defining semantics is one of the key activities of a programming languages researcher. We learn that there are different styles of dynamics (small-step, big-step, denotational, just to name the most frequently used one), each with different trade-offs. When it comes to implementing or mechanizing semantics, there are further options to choose from, in particular if we are also interested in statics.

One important choice is whether we want to express the statics extrinsically or intrinsically, that is, do we want to start with untyped syntax and then define the statics as an afterthought, or do we integrate types with the syntax.

If we opt for intrinsically typed syntax, some properties are already paid for by construction. For instance, a small-step semantics for intrinsically typed syntax satisfies type preservation by construction. For another instance, consider specifying a denotational semantics by a compositional mapping from syntax to some semantic domain. With untyped syntax, the semantic domain has to lump the interpretations of different types together and distinguish them using type tags. But with intrinsically typed syntax the semantics can map into type-indexed semantic domains and thus elide type tags. This observation directly translates to tagless interpreters on intrinsically typed syntax, which elide tag checks at run time.

For concreteness, we show the well-known example of a tagless interpreter for the simply-typed

lambda calculus implemented in Agda in Figure 1. We define the syntax as an inductive data type along with a compositional mapping to the semantic domain, spanned by Agda's natural number type and the function space. We define intrinsically typed syntax of expressions as an inductive datatype parameterized over a typing environment and indexed on the return type. For variables, we use de Bruijn indices into the typing environment.

The semantics of a typing environment is a runtime environment in the form of a heterogeneous list of suitably typed values. With all that, we can define the semantics of an expression $\mathcal{E}[_]$ as a function from the semantics of a typing environment $\mathcal{G}[_]$ to the semantics of the type $\mathcal{T}[_]$. Clearly this definition also serves as a tagless interpreter for the simply-typed lambda calculus, which means that type preservation is also built into its definition. Moreover, as Agda accepts this definition as terminating, we know that evaluation of every simply-typed lambda term terminates; a non-trivial semantic property of the simply-typed lambda calculus.

Agda-encodings of intrinsically-typed interpreters have been explored quite a lot, but rarely in the context of polymorphic source languages. One possible reason is that the archetypical polymorphic lambda calculus, System F, cannot be embedded in Agda because of its impredicativity. This begs the question if we can develop a tagless interpreter for a predicative version of System F in Agda.

We answer this question affirmatively for Leivant's stratified version of the polymorphic lambda cal-

```

module STLC where

open import Data.Nat using (ℕ; zero; suc)
open import Data.List using (List; []; ::; _)

data Type : Set where
  nat : Type
  _⇒_ : Type → Type → Type

 $\mathcal{T}[\_]$  : Type → Set
 $\mathcal{T}[\text{nat}] = \mathbb{N}$ 
 $\mathcal{T}[S \Rightarrow T] = \mathcal{T}[S] \rightarrow \mathcal{T}[T]$ 

Env = List Type

data _∈_ : Type → Env → Set where
  here :  $\forall \{T \Gamma\} \rightarrow T \in (T :: \Gamma)$ 
  there :  $\forall \{S T \Gamma\} \rightarrow S \in \Gamma \rightarrow S \in (T :: \Gamma)$ 

data Expr (Γ : Env) : Type → Set where
  con :  $\mathbb{N} \rightarrow \text{Expr } \Gamma \text{ nat}$ 
  var :  $\forall \{T\} \rightarrow T \in \Gamma \rightarrow \text{Expr } \Gamma T$ 
  lam :  $\forall \{S T\} \rightarrow \text{Expr } (S :: \Gamma) T \rightarrow \text{Expr } \Gamma (S \Rightarrow T)$ 
  app :  $\forall \{S T\} \rightarrow \text{Expr } \Gamma (S \Rightarrow T) \rightarrow \text{Expr } \Gamma S \rightarrow \text{Expr } \Gamma T$ 

data  $\mathcal{G}[\_]$  : Env → Set where
  [] :  $\mathcal{G}[\_]$ 
  _::_ :  $\forall \{T \Gamma\} \rightarrow \mathcal{T}[T] \rightarrow \mathcal{G}[\Gamma] \rightarrow \mathcal{G}[T :: \Gamma]$ 

lookup :  $\forall \{T \Gamma\} \rightarrow T \in \Gamma \rightarrow \mathcal{G}[\Gamma] \rightarrow \mathcal{T}[T]$ 
lookup here (x :: _) = x
lookup (there x) (_ :: γ) = lookup x γ

 $\mathcal{E}[\_]$  :  $\forall \{\Gamma T\} \rightarrow \text{Expr } \Gamma T \rightarrow \mathcal{G}[\Gamma] \rightarrow \mathcal{T}[T]$ 
 $\mathcal{E}[\text{con } n] \gamma = n$ 
 $\mathcal{E}[\text{var } x] \gamma = \text{lookup } x \gamma$ 
 $\mathcal{E}[\text{lam } e] \gamma = \lambda v \rightarrow \mathcal{E}[e] (v :: \gamma)$ 
 $\mathcal{E}[\text{app } e_1 e_2] \gamma = \mathcal{E}[e_1] \gamma (\mathcal{E}[e_2] \gamma)$ 

```

Figure 1: Simply typed lambda calculus, denotationaly

culus [10]. The key idea of his calculus is to stratify the set of polymorphic types in levels such that universal quantification only ranges over strictly smaller levels. This restriction literally embodies predicativity and, as we will discover, the stratification corresponds directly to Agda’s universe stratification.

2 Types

The definition of the type language for stratified System F is taken literally from Leivant’s paper. It is defined as an inductive type parameterized over a level environment (that assigns levels to free type variables) and indexed over the level of the type.

```

data Type (Δ : List Level) : Level → Set where
  nat : Type Δ zero
  _⇒_ : Type Δ ℓ → Type Δ ℓ' → Type Δ (ℓ ⊔ ℓ')
  ∀_ : ℓ ∈ Δ → Type Δ ℓ
  ∀*_ :  $\forall \ell \rightarrow \text{Type } (\ell :: \Delta) \ell' \rightarrow \text{Type } \Delta (\text{suc } \ell \sqcup \ell')$ 

```

The unit type lives at level 0. Type variables live at their declared level. The level of a function type $S \Rightarrow T$ is the maximum of the levels of S and T . The level of a universal quantification at level l is the maximum of $l + 1$ and the level of the body.

As for the simply-typed lambda calculus, we can define a compositional mapping from type syntax to Agda types.

```

 $\mathcal{T}[\_]$  : Type Δ ℓ → DEnv Δ → Set ℓ
 $\mathcal{T}[\text{nat}] \eta = \mathbb{N}$ 
 $\mathcal{T}[T_1 \Rightarrow T_2] \eta = \mathcal{T}[T_1] \eta \rightarrow \mathcal{T}[T_2] \eta$ 
 $\mathcal{T}[\forall \alpha] \eta = \text{apply-env } \eta \alpha$ 
 $\mathcal{T}[\forall \ell T] \eta = (D : \text{Set } \ell) \rightarrow \mathcal{T}[T] (\text{ext-env } D \eta)$ 

```

Given a type at level l , this function returns an Agda type in $\text{Set } l$. To do so it needs a domain environment to interpret type variables. This environment gets extended in the last clause that maps universal quantification to a dependent function that takes an element of $\text{Set } l$ and pushes it on the environment.

The type of the domain environment is interesting because its range type is unusual.

```

data DEnv : List Level → Set $\omega$  where
  [] : DEnv []
  _::_ : Set ℓ → DEnv Δ → DEnv (ℓ :: Δ)

```

As a value in the environment (the interpretation of a type, colloquially speaking) can live in a $\text{Set } l$, for any finite level l , we cannot assign the type any finite level. Hence, the type DEnv lives in the limit type $\text{Set}\omega$, which we use in this definition.

3 Expressions

Inspired by the encoding of System $F\omega$ by Chapman and coworkers [7], we define a unified environment for type variables and term variables. Type environments grow to the left.

```

data TEnv : List Level → Set where
  [] : TEnv []
  _◁_ : Type Δ ℓ → TEnv Δ → TEnv Δ - type binding
  _◁*_ :  $\forall \ell \rightarrow \text{TEnv } \Delta \rightarrow \text{TEnv } (\ell :: \Delta)$  - level binding

```

Membership of a term variable in a type environment is defined by the inn relation.

```

data inn : Type Δ ℓ → TEnv Δ → Set where
  here :  $\forall \{T : \text{Type } \Delta \ell\} \{\Gamma\} \rightarrow \text{inn } T (T \triangleleft \Gamma)$ 
  there :  $\forall \{T : \text{Type } \Delta \ell\} \{T' : \text{Type } \Delta \ell'\} \{\Gamma\} \rightarrow \text{inn } T \Gamma \rightarrow \text{inn } T (T' \triangleleft \Gamma)$ 
  tskip :  $\forall \{T : \text{Type } \Delta \ell\} \{\Gamma\} \rightarrow \text{inn } T \Gamma \rightarrow \text{inn } (\text{Tweaken } T) (\ell' \triangleleft^* \Gamma)$ 

```

In the last alternative, we skip over a type binding. Hence, the type T we find under the binding must be weakened to account for the extra type

variables. Weakening is a special case of renaming, which is implemented as advocated by Benton and coworkers [4].

The type of expressions is now given as follows.

```
data Expr {Δ : List Level} (Γ : TEnv Δ) : Type Δ ℓ → Set where
#_ : ∀ (n : ℕ) → Expr Γ nat
'_ : ∀ {T : Type Δ ℓ}
  → inn T Γ → Expr Γ T
λ_ : ∀ {T : Type Δ ℓ} {T' : Type Δ ℓ'}
  → Expr (T ◁ Γ) T' → Expr Γ (T ⇒ T')
'_ : ∀ {T : Type Δ ℓ} {T' : Type Δ ℓ'}
  → Expr Γ (T ⇒ T') → Expr Γ T → Expr Γ T'
Λ : ∀ (ℓ : Level) → {T : Type (ℓ :: Δ) ℓ'}
  → Expr (ℓ ◁* Γ) T → Expr Γ (∀ ℓ T)
'_ : ∀ {T : Type (ℓ :: Δ) ℓ'}
  → Expr Γ (∀ ℓ T) → (T' : Type Δ ℓ)
  → Expr Γ (T [ T' ] T)
```

Variables, lambda abstractions, and application are encoded just like for the simply-typed lambda calculus. Type abstraction takes a level l and a body where the new type variable is bound to l . Type application takes an expression with universal quantification at level l and a type T' of level l . It constructs an expression where type T' has been substituted in the body T of the quantified type. Substitution is defined as in PLFA [4, 9].

4 Semantics

It remains to define a compositional function from the expression syntax to the semantic domain that we already prepared in Section 2. We start with value environments.

```
Env : ∀ {Δ : List Level} → TEnv Δ → DEnv Δ → Setω
Env {Δ} Γ η = ∀ {ℓ} {T : Type Δ ℓ} → inn T Γ →  $\mathcal{T} \llbracket T \rrbracket \eta$ 
```

Value environments are represented as functions—we could have done that in the simply-typed interpreter, too. They are indexed by a domain environment to be able to calculate the correct return type.

The definition of the interpretation function follows.

```
 $\mathcal{E} \llbracket \_ \rrbracket : \forall \{T : Type \Delta \ell\} \{\Gamma : TEnv \Delta\}$ 
  → Expr Γ T → (η : DEnv Δ) → Env Γ η →  $\mathcal{T} \llbracket T \rrbracket \eta$ 
 $\mathcal{E} \llbracket \# n \rrbracket \eta \gamma = n$ 
 $\mathcal{E} \llbracket ' x \rrbracket \eta \gamma = \gamma x$ 
 $\mathcal{E} \llbracket \lambda \_ e \rrbracket \eta \gamma = \lambda v \rightarrow \mathcal{E} \llbracket e \rrbracket \eta (\text{extend } \gamma v)$ 
 $\mathcal{E} \llbracket e_1 \cdot e_2 \rrbracket \eta \gamma = \mathcal{E} \llbracket e_1 \rrbracket \eta \gamma (\mathcal{E} \llbracket e_2 \rrbracket \eta \gamma)$ 
 $\mathcal{E} \llbracket \Lambda \ell e \rrbracket \eta \gamma = \lambda D \rightarrow \mathcal{E} \llbracket e \rrbracket (\text{ext-env } D \eta) (\text{extend-skip } \gamma)$ 
 $\mathcal{E} \llbracket \_ \bullet \_ \{T = T'\} e T \rrbracket \eta \gamma =$ 
  subst id (sym (single-subst-preserves T T'))
  (  $\mathcal{E} \llbracket e \rrbracket \eta \gamma (\mathcal{T} \llbracket T' \rrbracket \eta)$  )
```

The cases for term variables, lambda abstraction, and application are similar to the simply-typed lambda calculus.

The first issue arises in the case for type abstraction. We interpret a type abstraction at level l as a function with argument type $\text{Set } l$. This argument has to be pushed onto the domain environment η and we have to account at the value level for the additional type variable in the type environment. The following function adapts the types.

```
extend-skip : ∀ {Δ : LEnv} {Γ : TEnv Δ} {η : DEnv Δ} {D : Set ℓ}
  → Env Γ η → Env (ℓ ◁* Γ) (D :: η)
extend-skip {η = η} {D = D} γ (tskip {T = T} x) =
  subst id (sym (ren*-preserves-semantics {ρ = wkr} {η} {D :: η}
    (wkr ∈ TRen* η D) T))
  (γ x)
```

The lemma we need in the rewrite clause proves that interpreting a weakened type in an extended domain environment gives the same result as interpreting the type in the original domain environment. The statement of this lemma is more general as it applies to arbitrary renamings:

```
ren*-preserves-semantics :
  ∀ {ρ : TRen Δ1 Δ2}} {η1 : DEnv Δ1}} {η2 : DEnv Δ2}}
  → (ren* : TRen* ρ η1 η2) → (T : Type Δ1 ℓ)
  →  $\mathcal{T} \llbracket Tren \rho T \rrbracket \eta_2 \equiv \mathcal{T} \llbracket T \rrbracket \eta_1$ 
```

The argument ren^* roughly states that η_1 and η_2 are domain environments related by renaming ρ by precomposition $\eta_1 \equiv \eta_2 \circ \rho$. The proof of the lemma is by induction on T with an interesting subgoal in the case for universal quantification:

```
(T : Type (ℓ :: Δ1) ℓ') →
((D : Set ℓ) →  $\mathcal{T} \llbracket Tren (\text{ext}_r \rho) T \rrbracket (D :: \eta_2)$ ) ≡
((D : Set ℓ) →  $\mathcal{T} \llbracket T \rrbracket (D :: \eta_1)$ )
```

We can show that the ranges of the function are equal with the inductive hypothesis. But the usual extensionality principle does not let us expose this equation. However, it can be used to prove a dependent extensionality principle (from the standard library), which enables us to complete the proof.

```
∀-extensionality :
  ∀ {a b} {A : Set a} {FG : (α : A) → Set b}
  → (∀ (α : A) → F α ≡ G α)
  → ((α : A) → F α) ≡ ((α : A) → G α)
```

The final case for type application opens two different cans of worms. First, the type of the right hand side does not match the expected type. Essentially, we have to prove that the composition of the meaning function for types commutes with substitution. Here $T[T']$ substitutes T' for the outermost variable of T .

```
single-subst-preserves :
  ∀ {η : DEnv Δ} {T' : Type Δ ℓ} (T : Type (ℓ :: Δ) ℓ')
  →  $\mathcal{T} \llbracket T [ T' ] T \rrbracket \eta \equiv \mathcal{T} \llbracket T \rrbracket (\mathcal{T} \llbracket T' \rrbracket \eta :: \eta)$ 
```

Second, some steps in the proof involve equalities over entities of $\text{Set } \omega$. These cannot be handled with the standard definition of propositional equality which works parametrically for entities of

Set l , for any l , but not for $\text{Set}\omega$. While it is easy to define these equalities, it is somewhat tedious to re-establish standard lemmas for transforming equality proofs like `cong`, `subst`, and `trans` to deal with $\text{Set}\omega$.

5 Related Work

Arjen Rouvoet's thesis [14] gives an excellent overview of the state of the art in intrinsically typed techniques for modeling language semantics. He pushed the limits of this technology in a range of papers with various coauthors [11, 15, 16, 18].

Giving semantics in an interpretive style is a defining feature of denotational semantics [17], but it can be traced back to Reynolds's idea of definitional interpreters [12].

The particular encoding of de Bruijn style variable representations used in this work dates back to work on nested datatypes [2, 5, 6], which subsequently lead to GADTs [8].

A tagless interpreter for a simply typed calculus was developed in Cayenne [3], but for extrinsically typed syntax (i.e., syntax and separate typing predicate).

Intrinsically typed encodings are further studied by Allais and others [1], who define a range of tagless functions including denotational semantics on intrinsically typed terms.

We draw some inspiration from the program implemented by Benton and coworkers [4]. Based on intrinsically typed syntax for a simply-typed lambda applied calculus, they define a big-step semantics and a set-theoretic denotational semantics. They prove soundness of the former semantics with respect to the latter as well as adequacy (using a logical relation). They also develop an expression encoding for System F, but the paper stops short of discussing its semantics.

6 Future Work

We are currently formalizing the small-step semantics of the language with the goal of proving an adequacy theorem for reduction with respect to the denotational semantics.

It would also be interesting to extend the stratified calculus by level quantification as in Agda's universe polymorphism.

References

- [1] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, Paris, France, 195–207. <https://doi.org/10.1145/3018610.3018613>
- [2] Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic, CSL '99, 8th Annual Conference of the EACSL (Lecture Notes in Computer Science, Vol. 1683)*, Jörg Flum and Mario Rodríguez-Artalejo (Eds.). Springer, Madrid, Spain, 453–468. https://doi.org/10.1007/3-540-48168-0_32
- [3] Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. (May 1999). <https://www.researchgate.net/publication/2610129> unpublished manuscript.
- [4] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *J. Autom. Reason.* 49, 2 (2012), 141–159. <https://doi.org/10.1007/s10817-011-9219-0>
- [5] Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction, MPC'98 (Lecture Notes in Computer Science, Vol. 1422)*, Johan Jeuring (Ed.). Springer, Marstrand, Sweden, 52–67. <https://doi.org/10.1007/BFb0054285>
- [6] Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77–91. <https://doi.org/10.1017/s0956796899003366>
- [7] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction - 13th International Conference, MPC 2019 (Lecture Notes in Computer Science, Vol. 11825)*, Graham Hutton (Ed.). Springer, Porto, Portugal, 255–297. https://doi.org/10.1007/978-3-030-33636-3_10

- [8] James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types*. Technical Report TR2003-1901. Cornell University. <https://ecommons.cornell.edu/handle/1813/5614>
- [9] Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Sci. Comput. Program.* 194 (2020), 102440. <https://doi.org/10.1016/j.scico.2020.102440>
- [10] Daniel Leivant. 1991. Finitely Stratified Polymorphism. *Inf. Comput.* 93, 1 (1991), 93–113. [https://doi.org/10.1016/0890-5401\(91\)90053-5](https://doi.org/10.1016/0890-5401(91)90053-5)
- [11] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.* 2, POPL (2018), 16:1–16:34. <https://doi.org/10.1145/3158104>
- [12] John C. Reynolds. 1975. User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In *New Directions in Algorithmic Languages*, Stephen A. Schumann (Ed.). INRIA, St. Pierre-de-Chartreuse, 309–317. Reprinted in [13].
- [13] John C. Reynolds. 1994. User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, Carl A. Gunter and John C. Mitchell (Eds.). MIT Press, Cambridge, MA, USA, 13–23. Originally published in [12].
- [14] Arjen Rouvoet. 2021. *Correct by Construction Language Implementations*. Ph.D. Dissertation. Delft University of Technology, Netherlands. <https://doi.org/10.4233/uuid:f0312839-3444-41ee-9313-b07b21b59c11>
- [15] Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2021. Intrinsically typed compilation with nameless labels. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. <https://doi.org/10.1145/3434303>
- [16] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, New Orleans, LA, USA, 284–298. <https://doi.org/10.1145/3372885.3373818>
- [17] David A. Schmidt. 1986. *Denotational Semantics, A Methodology for Software Development*. Allyn and Bacon, Inc, Massachusetts. <https://people.cs.ksu.edu/~schmidt/text/ds0122.pdf>
- [18] Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter D. Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1903–1932. <https://doi.org/10.1145/3563355>

Pattern Matching in Java-TX

Daniel Holle
Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
Department of Computer Science
Florianstraße 15, 72160 Horb
daniel.holle@dhbw.de

Zusammenfassung

In den letzten Jahren hat sich die Programmiersprache Java stetig weiterentwickelt. Sie folgt dabei einem breiteren Trend, der darauf abzielt, Elemente aus den funktionalen Programmiersprachen zu integrieren. Eines dieser Features ist das sogenannte Pattern-Matching, welches z.B. aus Haskell bekannt ist. Während in Java das neu eingeführte Pattern-Matching nur innerhalb von Switch und instanceof-Expressions möglich ist, geht Java-TX einen Schritt weiter und erlaubt Patterns auch in Funktionsköpfen. Da Funktionen nur nach Objekttypen und nicht deren Struktur überladen werden können, müssen weitere Ansätze verfolgt werden um diese Funktionalität im Bytecode zu erhalten.

Was ist Pattern-Matching?

Pattern-Matching ist im Prinzip eine Fallunterscheidung, mit der nach der Struktur der Elemente unterschieden wird. Konzeptionell ist es ähnlich wie ein If-Statement mit mehreren Branches. Bei einem If-Statement werden jedoch nur boolsche Ausdrücke ausgewertet und nicht nach der Struktur unterschieden. Die Struktur können der Vergleich von konkreten Werten sein, welche in gekürzter Form dem If-Statement entsprechen. Gerade funktionale Programmiersprachen wie Haskell bieten jedoch deutlich mehr Pattern an. So kann beispielsweise eine Liste, welche mit einem bestimmten Wert anfängt, gematcht werden. Konzeptionell erinnert das Pattern-Matching an ein Switch-Statement, welches bereits in C eingeführt worden ist. Tatsächlich baut das in Java 16 eingeführte Pattern-Matching auf einer Erweiterung des Switch-Statements auf. Der große Unterschied liegt allerdings in der Implementierung. Während ein Switch auf einem primitiven Integer direkt in einen Tableswitch umgewandelt werden kann, muss für das Pattern-Matching die Struktur jedes Branches einzeln geprüft werden. Bei einem Tableswitch wird direkt anhand des Wertes in

einen bestimmten Branch gesprungen. Deshalb ist dieser bei einer naiven Implementierung oftmals schneller als ein If-Statement mit mehreren Branches, da die einzelnen Bedingungen nicht geprüft werden müssen. Der große Vorteil beim Pattern-Matching ist allerdings eine verbesserte Lesbarkeit gegenüber dem If-Statement. Die Performanz ist generell eher Nebensache.

Pattern-Matching in Java

Wie bereits erwähnt, wird in Java das Switch-Statement angepasst, um das Pattern-Matching zu implementieren. Das klassische Switch-Statement weist allerdings, da es identisch zu C übernommen wurde, mehrere Probleme auf. Beim Pattern-Matching ist es üblich, dass immer nur einer der Branches ausgeführt werden kann. Das Switch-Statement aus C ist allerdings konzeptionell ein gelabeltes goto-Statement. Das bedeutet, dass an die richtige Stelle gesprungen wird, und dann der Code solange ausgeführt wird, bis ein break-Statement erreicht wird. Das ist bei einfachen Bedingungen kein Problem, aber sollte

ein Pattern nur in einem Branch zutreffen, würde, falls das `break`-Statement weggelassen wird, auch ein Branch ausgeführt werden, in dem das Pattern nicht zutrifft.

```
int x = 3;
switch (x) {
case 1:
case 2:
    print("small");
    break;
case 3:
    print("three");
    break;
default:
    print("?");
}
```

Abbildung 1: Switch-Statement in Java

Um diesen Fehler zu beheben, wurde mit Java-14 eine neue Variante von Switch-Statements eingeführt, die sogenannten Switch-Expressions:

```
int x = 3;
var res = switch(x) {
    case 1, 2 -> "small";
    case 3 -> "three";
    default -> "?";
};
print(res);
```

Abbildung 2: Switch-Expression in Java

Wie hier zu sehen ist, fällt das `break`-Statement komplett weg und jeder Branch besitzt nur ein Pattern. Für einfache Patterns können auch mehrere Werte gleichzeitig geprüft werden, damit wird das Fallthrough überflüssig gemacht. Weiterhin ist hier der `default`-Case verpflichtend. Das liegt daran, dass die Switch-Expression logischerweise eine Expression ist, die einen Wert zurückgeben kann. Deshalb müssen alle Möglichkeiten abgedeckt werden. Interessanterweise sind die neuen Patterns auch im alten Switch-Statement valide. Hier wird allerdings überprüft, ob der Branch ein finales `break` enthält.

Arten von Patterns in Java

In Java wurden die folgenden drei Arten von Patterns eingeführt:

1. Value pattern:

```
case 10 -> ...;
case "my string" -> ...;
```

2. Type pattern:

```
case Integer i -> ...;
case String s -> ...;
```

3. Record pattern:

```
case Point(
    Integer x, Integer y) -> ...;
case Some(Object v) -> ...;
```

Das Value-Pattern entspricht dem, was mit dem alten Switch-Statement möglich war.

Die Type-Patterns sind dementsprechend gleichzusetzen mit einem `instanceof`. Hier wird geprüft, ob ein Wert von einem bestimmten Typ ist, und gleichzeitig wird eine lokale Variable mit diesem Typ im Branch erstellt. Die Type-Patterns sind besonders hilfreich, um das Visitor-Pattern zu ersetzen. Dieses Feature arbeitet besonders gut mit den in Java 17 eingeführten Sealed Classes. Da hier bereits von vornherein bestimmt wird, welche Klassen von einem Interface erben können, kann für die Fallunterscheidung bestimmt werden, ob alle Möglichkeiten abgedeckt sind. Ein `default` ist in diesem Fall also nicht nötig.

Die Record-Patterns können nur für die in Java 14 eingeführten Records verwendet werden. In funktionalen Sprachen wie Haskell kann hier jeder Data constructor verwendet werden. Das liegt daran, dass diese Typen einen kanonischen Konstruktor aufweisen. Records in Java besitzen ebenso einen solchen kanonischen Konstruktor. Falls hier jede Klasse verwendet werden könnte, stellen sich Fragen, wie ob die Reihenfolge der Felder eingehalten werden oder wie `private` oder `protected` Felder gematcht werden müssten. Durch die Records werden diese Einschränkungen festgelegt, deshalb können sie hier verwendet werden. Ein weiterer interessanter Punkt ist, dass diese Patterns genestet werden können. Ebenso können verschiedene Typen, also Subklassen der Felder des Records, angegeben werden (Hier beispielsweise `Float x` statt `Integer x`). Es wird immer der genestete Typ überprüft und nur in den Case gesprungen, der die Bedingungen erfüllt.

Pattern Matching in Java-TX

JavaTX ist eine Programmiersprache die auf Java aufbaut. Das wichtigste Feature ist eine globale Typinferenz, die es erlaubt, an den meisten Stellen die Typen wegzulassen und vom Compiler zu inferieren. Auf die Besonderheiten der Sprache soll hier nicht näher eingegangen, sondern lediglich die Unterschiede zu Java in Bezug auf Pattern-Matchings aufgezeigt werden.

Die naheliegendste Möglichkeit ist es sicherlich, die Typen bei den Record patterns wegzulassen. Das sieht folgendermaßen aus:

```
case Point(Integer x, Integer y)
→ case Point(x, y)
```

Man könnte hier naiv einfach die Typen einsetzen, welche im Record definiert wurden. Tatsächlich bilden diese allerdings nur eine untere Schranke. Interessant wird es, wenn die Typen weiter eingeschränkt werden.

```
m(Integer i) { ... }
m(Double d) { ... }

record R(Object o) {}

main(r) {
  switch(r) {
    case R(o) -> m(o);
    default -> null;
  };
}
```

Abbildung 3: Überladen von Case-Labels

In diesem Beispiel würde die Variable `o` durch den Methodenaufruf `m(o)` auf die beiden Typen `Integer` und `Double` eingeschränkt. Da die Methode `m` allerdings überladen ist, wird nun der case Überladen. In diesem Fall sind beide Typen angegeben, es könnte aber auch genauso von der Typinferenz berechnet werden. Wenn das Ganze nach Java übersetzt werden würde, würde es so aussehen:

```
switch(r) {
  case R(Integer o) -> m(o);
  case R(Double d) -> m(d);
  default -> null;
};
```

JavaTX bietet bereits eine automatische Überladung von Methoden an, falls mehrere korrekte

Typeeinsetzungen generiert werden. Das Switch-Statement ist also eine logische Fortführung dieser Idee.

Eine weitere Möglichkeit wäre die Erweiterung des Value-Matchings, bei dem es auch möglich ist, geschachtelte Records als Patterns zu verwenden. Das entspricht im Ansatz dem, was auch mit Haskell möglich ist.

```
case Point(10, 20) -> ...
```

Als letzte Erweiterung wird die Implementierung von Pattern-Matching im Funktionskopf vorgestellt.

```
f(Integer x) { ... }

f(Point(Integer x, Integer y)) { ... }
```

Die Typen können natürlich auch weggelassen werden. Die zweite Funktion kann also nur mit Points aufgerufen werden, welche als `x` und `y` `Integer` zugewiesen bekommen. Die Semantik entspricht dem regulären Pattern-Matching. In diesem Fall gibt es auch eine Überladung, was nach Java kein Problem ist, da die erste Funktion einen `Integer` und die zweite Funktion einen `Point` übergeben bekommt.

Das nächste Beispiel lässt sich allerdings nicht so einfach mit Überladungen lösen:

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

f(Point(Float x, Float y)) { ... }

f(Point(Integer x, Integer y)) { ... }
```

Hier haben beide Funktionen die selbe Signatur, es kann also nicht zur Compilezeit entschieden werden, welche Funktion aufgerufen werden kann. Als Lösung müssen zwei Funktionen mit unterschiedlichen Namen generiert werden. Diese werden von einer weiteren Funktion aufgerufen, welche zur Laufzeit je nach den Record-Parametern entscheidet, welche Funktion aufgerufen wird.

```

record Point(Number x, Number y) {}

private void f$1(Point p) { ... }
private void f$2(Point p) { ... }

void f(Point point) {
    switch(point) {
        case Point(Float x, Float y)
            -> f$1(record)
        case Point(Integer x, Integer y)
            -> f$2(record)
    }
}

```

Dies entspricht in etwa dem generierten Java-Code.

Implementierung der Switch-Expression in Java-TX

Für die Implementierung der klassischen Switch-Expression, das heißt unter der Verwendung von Integer-Werten, gibt es zwei JVM Instruktionen: `tableswitch` und `lookupswitch`.

Der `tableswitch` funktioniert ähnlich wie in klassischen C-Programmen. Hier wird für jeden Wert direkt über eine Sprungtabelle in den richtigen Branch gesprungen. Dies ist gerade für viele Werte sehr viel schneller als ein `if`-Statement. Ein einziges Problem mit diesem Ansatz besteht darin, wenn die Werte weit voneinander entfernt sind, und der Switch viele Löcher aufweist.

Für diese Fälle wird ein `lookupswitch` verwendet. Hier müssen nicht alle Werte nahe beieinander liegen. Die JVM wendet hier üblicherweise eine Binärsuche an, es lohnt sich also weiterhin mehr als primitive `if`-Statements.

Diese beiden Varianten funktionieren natürlich nur für Integer-Werte. Bei einem Switch mit Typ-Patterns muss die Klasse des Eingabewertes überprüft werden. Ein naiver Ansatz wäre beispielsweise den `instanceof` Befehl der JVM zu verwenden. Der Aufbau entspricht dann weitestgehend einem `if`-Statement mit mehreren Branches. Wenn man allerdings den Bytecode der kompilierten Switch-Expression betrachtet, werden einige Unterschiede deutlich.

```

int f(Object o) {
    return switch(o) {
        case Integer i -> 1;
        case Float f -> 2;
        case String s -> 3;
        default -> 4;
    };
}

// Hier werden Parameter übergeben
// InvokeDynamic
// #0: typeSwitch:
//   (Ljava/lang/Object;I)I
11: invokedynamic #13, 0
16: tableswitch { // 0 to 2
           0: 44
           1: 54
           2: 64
           default: 74
        }
44:
// class java/lang/Integer
45: checkcast    #17
48: astore       4
50: iconst_1
51: goto         75
// Cases für Float & String
// ausgeklammert
// default:
74: iconst_4
75: ireturn

// Bootstrap Methode:
0: #33 REF_invokeStatic
...SwitchBootstraps.typeSwitch:...
   #17 java/lang/Integer
   #19 java/lang/Float
   #21 java/lang/String

```

Wie hier zu sehen ist, wird für die Implementierung von Type-Switches der Befehl `invokedynamic` verwendet zusammen mit der Methode `SwitchBootstraps.typeSwitch`, an die als Parameter die Klassen `Integer`, `Float` & `String` übergeben werden. `Invokedynamic` ist ein relativ neues Sprachfeature von Java. Mit `invokedynamic` wird praktisch beim ersten Aufruf der Bytecode für eine Methode generiert, wobei die Bootstrap Methode die konstanten Parameter aus der Klasse übergeben bekommt.

Die Methode, die hier generiert wird, bekommt zwei Parameter übergeben. Der erste Parameter ist das Objekt, welches inspiziert werden soll. Als zweiter Parameter wird der Offset ab dem geprüft werden soll, übergeben. Das ist wichtig, weil die Methode bei komplexeren Beispielen mehrmals in einer Schleife aufgerufen wird. Das kann passieren, wenn durch eine weitere Überprüfung fest-

gestellt wird, dass der momentane Branch doch nicht der richtige war. Das ist bei Record-Patterns wichtig, weil es hier mehrere Möglichkeiten für den selben Typ geben kann.

Die Methode gibt als Rückgabewert den Index des Branches zurück, beziehungsweise -1 für den Default-Case. Auf diesem Wert wird dann ein regulärer `tableswitch` ausgeführt, um in den richtigen Branch zu springen.

Zusammenfassung

Pattern-Matching ist ein sehr beliebtes Feature aus funktionalen Sprachen. In diesem Paper wurde evaluiert, wie Pattern-Matching in Java umgesetzt wurde und, darauf aufbauend, Java-TX erweitert wurde, um einen Mehrwert gegenüber Java zu schaffen. Es wurden Ideen für die Implementierung ergänzt und dabei explizit auf `invokedynamic` verwiesen. Zum jetzigen Stand befinden sich die Features noch in der Erprobungsphase und sind noch nicht final umgesetzt.

Literatur

- [1] Gavin Bierman. *JEP 361: Switch Expressions*. 11. März 2022. URL: <https://openjdk.org/jeps/361>.
- [2] Gavin Bierman. *JEP 395: Records*. 3. Feb. 2024. URL: <https://openjdk.org/jeps/395>.
- [3] Gavin Bierman. *JEP 409: Sealed Classes*. 3. Jan. 2024. URL: <https://openjdk.org/jeps/409>.
- [4] Gavin Bierman. *JEP 441: Pattern Matching for switch*. 19. Sep. 2023. URL: <https://openjdk.org/jeps/441>.
- [5] Martin Plümicke und Etienne Zink. *Java-TX: The language*. INSIGHTS – Schriftenreihe der Fakultät Technik 01/2022. DHBW Stuttgart, 2022. URL: https://www.dhbw-stuttgart.de/fileadmin/dateien/Forschung/Forschungsschwerpunkte_Technik/DHBW_Stuttgart_INSIGHTS_1_2022_Java-TX_The_language.pdf.
- [6] John R Rose. „Bytecodes meet combinators: invokedynamic on the JVM“. In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*. 2009, S. 1–11.
- [7] Sukyoung Ryu, Changhee Park und Guy L Steele Jr. „Adding pattern matching to existing object-oriented languages“. In: *ACM SIGPLAN Foundations of Object-Oriented Languages Workshop*. Bd. 5. Citeseer. 2010.

Featherweight-Java-TX: A Minimal Core Calculus for Java-TX (FJ-TX)

Martin Plümicke
 Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
 Department of Computer Science
 Florianstraße 15, 72160 Horb
 pl@dhbw.de

Abstract

In this paper we present a calculus **FJ-TX** for Java-Type eXtended (Java-TX).

One of the predominant new features in Java-TX is global type inference. For this there is an algorithm which use indeed wildcards but have no capture conversion.

Instead of the wildcards are consider a real types and the subtyping ordering is extended to wildcard types.

The results of the type inference algorithm of Java-TX consists of substitutions and sets of remaining type variable constraints. The remaining set of constraints are distributed to the type variables (generics) of the type inferred class and its methods, respectively. Furthermore the constraints have to be adapted as not any binary relation of type variables are correct generics in Java.

1 Introduction

Java's subtype relation is invariant by default, meaning the type `List<Integer>` is not a subtype of `List<Object>`. The reason is shown in figure 1: where Java's type check prevents a value of type `Object` to be added to a list of `Integers`. But invariant subtyping can also be overly restrictive. Therefore Java uses so called Wildcards to allow use-side variance.

```
List<Integer> intList = new List<
    Integer>();

//Illegal assignment
List<Object> objList = intList;

// This would add an Object to a List
// of Integers:
objList.add(new Object());
```

Figure 1: Erronous Java program

The problem of missing covariance is solved by introduction of so called *wildcards*: `"? extends ty"`. This is a form of existential types and means that there is a type parameter which is a subtype of `ty`. In other words for the `?` all subtypes of `ty` could be instantiated. This means the type must be correct for all possible instances. The above example for a list conversion looks like:

```
List<Integer> intList =
    new List<Integer>();
List<? extends Object> objList=intList
;
objList.add(new Object()); //incorrect
```

Now the last line is not correct, as `Object` is not a subtype of all subtypes of `Object`.

Following the idea `? extends Object` could not be a subtype of itself, which means the Java subtyping relation with wildcard types is not reflexive.

This leads to a problem in methods where a generic type variable is used multiple, e.g.

```
<T> void append2List(List<T> x,
                    List<T> y) {
    x.append(y);
}
```

The list's element type of y must be a subtype of the list's element type of x . Therefore, the following call is forbidden:

```
List<?> l1 = ...
List<?> l2 = ...
append2List(l1, l2);
```

This is realized by the so-called *capture conversion*.

Definition 1.1 (Capture Conversion). Let $G\langle A_1, \dots, A_n \rangle$ be a generic type with corresponding bounds U_1, \dots, U_n .

$G\langle S_1, \dots, S_n \rangle$ is a capture conversion of $G\langle T_1, \dots, T_n \rangle$

if

- $T_i = ? \text{ extends } ty \Rightarrow S_i$ is a fresh type variable with upper bound

$glb(ty, U_i[A_1 \mapsto S_1, \dots, A_n \mapsto S_n])$.

- $T_i = ? \text{ super } ty \Rightarrow S_i$ is a fresh type variable with upper bound $U_i[A_1 \mapsto S_1, \dots, A_n \mapsto S_n]$ and lower bound ty .
- otherwise: $S_i = T_i$.

Summarized, this means each wildcard is substituted by a fresh capture type variable, respectively. Therefore in the above example `append(v1, v2)` is wrong as T must be instantiated by the same type at each appearance.

This approach, which is realized in Java for about 20 years, is on the one hand sufficient but not necessary and on the other hand very hard to understand, especially error-messages.

The next example presents a program which is not type-correct but sound.

In Fig. 2 the variable `l2` is assigned to `l1`. This is correct although both types contain wildcards. But the call of `assign` becomes incorrect as the capture conversion generates two fresh type variables although the method assigns `l2` to `l1`.

The Java-Compiler gives the following complicated error-message:

```
class Assign {
    <X> void assign(List<X> l1,
                 List<X> l2) {
        l1 = l2;
    }

    void main() {
        List<?> l1 = new ArrayList<>()
            ;
        List<?> l2 = new ArrayList<>()
            ;
        l1 = l2;

        //not type correct but sound

        assign(l1, l2);
    }
}
```

Figure 2: The assign-example

```
assign(l1, l2);
~
required: List<X>,List<X>
found: List<CAP#1>,List<CAP#2>
reason: inference variable X has
incompatible
equality constraints CAP#2,CAP#1
where X is a type-variable:
X extends Object declared in
method <X>assign(List<X>,List<X>)
where CAP#1,CAP#2 are fresh type-
variables:
CAP#1 extends Object from capture of ?
CAP#2 extends Object from capture of ?
```

For these two reasons we give another approach which avoids capture conversion.

There are key-points of our approach

- Wildcards should be treated as all other types, especially it should be allowed to instantiate wildcards into type variables,

- Following the semantics of the wildcard-types

$? \text{ extends } \theta$: There is a subtype of θ .

$? \text{ super } \theta'$: There is a supertype of θ' .

we continue the subtyping relation $<$: on wildcard-types.

For two given Java-types θ, θ' with $\theta <: \theta'$ holds

– $\theta <: ? \text{ super } \theta'$,

– $? \text{ extends } \theta <: \theta'$, and

– $? \text{ extends } \theta <: ? \text{ super } \theta'$.

Remark: Especially, $<:$ is not reflexive on wildcards.

- The consequence of this remark is that type variables are not reflexive, too.

The above examples would have the following typing:

```
<R, T extends R>
void append2List(List<R> x,
                 List<T> y) {
    x.append(y);
}
```

Then

```
List<? extends Object> l1 =
    new List<Integer>();
List<? extends Object> l2 =
    new List<String>();
append(l1, l2);
```

leads to

$$[R \mapsto ? \text{ extends Object}, T \mapsto ? \text{ extends Object}]$$

and this results in

$$? \text{ extends Object} <: ? \text{ extends Object}$$

which leads to a type error as $<:$ is not reflexive on wildcard-types.

In contrast the assign-example (Fig. 2) leads to

```
List<? extends Object> <
    List<? extends Object>
```

which is correct.

2 Featherweight Java-TX

2.1 Syntax

In Fig. 3 the syntax of **FJ-TX** is defined. It is an extension of Featherweight Generic Java (**FGJ**) in [6] where in the arguments of the parameterized types wildcards are allowed. As not at all type positions wildcards are allowed we have define different rules for types:

T includes alle **FJ-TX** types.

U are all types without type variables. They are needed for type-casts.

E are all **FJ-TX** types added by wildcard types.

N are the **FGJ** types without type variables. They are needed for class extensions, as there are no type variables allowed.

NX are all **FGJ** types including type variables.

The rule **w** defines the wildcard types. Wildcard types are real type in **FJ-TX**. Indeed, they are not allowed in declarations but, as will see later, they are considered as a normal type in the subtyping relation.

$?_{LB}^{UB}$ means that the wildcard has an upper-bound UB and a lower bound LB . For example $? \text{ extends Integer}$ is denoted as $?_{Null}^{Integer}$, where $Null$ stands for the lower bound of all types. Analogously, e.g. $?_{Number}^{Object}$ stands for $? \text{ super Number}$.

2.2 Type rules

An environment Γ is a finite mapping from variables to types, written $\bar{x} : \bar{T}$; a type environment Δ is a finite relation from type variables to non variable types and the identity relationship, written $\bar{X} <: \bar{T}$, which takes each type variable to its bound. Though each type variable has only one bound. Additionally each type variable could mapped to itself ($X <: X$). This is necessary as type variables are not reflexive by default. Therefore, $X <: X$ is explicitly declared as reflexive.

In Fig. 4 the subtyping relation and wellformed types are defined. In Fig. 5 the expression, the method typing, and the class typing rules are defined. The rules are oriented at the rules in [6].

2.2.1 Subtyping

The following rule defines the subtype relation in **FJ-TX**.

S-REFL

The **S-REFL** rule is essentially changed. While in **FGJ** types variables are reflexive in **FJ-TX** type variables are not reflexive. This is a necessary change as wildcards are not reflexive and wildcards should be instantiated into type variables.

$$\begin{aligned}
 T &::= U \mid X \\
 U &::= C\langle \bar{E} \rangle \\
 E &::= T \mid W \\
 W &::= ?_T^T \\
 N &::= C\langle \bar{N}\bar{X} \rangle \\
 NX &::= X \mid N \\
 L &::= \text{class } C\langle \bar{X} \triangleleft \bar{T} \rangle \triangleleft N \{ \bar{T} \bar{f}; \bar{M} \} \\
 M &::= \langle \bar{X} \triangleleft \bar{T} \rangle T m(\bar{T} \bar{x}) \{ \text{return } e; \} \\
 e &::= x \mid e.f \mid e.m(\bar{e}) \mid \text{new } C(\bar{e}) \mid (U) e
 \end{aligned}$$

Figure 3: Syntax of FJ-TX

Subtyping:

$ \begin{array}{c} \text{S-REFL} \\ \Delta \vdash U <: U \end{array} $	$ \begin{array}{c} \text{S-TRANS} \\ \frac{\Delta \vdash E <: E' \quad \Delta \vdash E' <: E''}{\Delta \vdash E <: E''} \end{array} $	$ \begin{array}{c} \text{S-VAR} \\ \frac{(X <: T) \in \Delta}{\Delta \vdash X <: T} \end{array} $	$ \begin{array}{c} \text{S-CLASS} \\ \frac{\text{class } C\langle \bar{X} \triangleleft \bar{T} \rangle \triangleleft N \{ \dots \}}{C\langle \bar{N}\bar{X} \rangle <: [\bar{N}\bar{X}/\bar{X}]\bar{N}} \end{array} $
$ \begin{array}{c} \text{S-WC-CLASS} \\ \frac{\text{class } C\langle \bar{X} \triangleleft \bar{T}, \bar{X}' \triangleleft \bar{T}' \rangle \triangleleft C' \langle \bar{X} \rangle \{ \dots \} \quad \bar{E} <_? \bar{E}'}{C\langle \bar{E} \rangle <: C' \langle \bar{E}' \rangle} \end{array} $	$ \begin{array}{c} \text{S-SUPER-OBJECT} \\ \Delta \vdash E <: \text{Object} \end{array} $	$ \begin{array}{c} \text{S-WC-SUPER} \\ \Delta \vdash T <: ?_T^T \end{array} $	
$ \begin{array}{c} \text{S-WC-EXTENDS} \\ \Delta \vdash ?_T^T <: T' \end{array} $			

Use-side-variance Subtyping:

$ \begin{array}{c} \text{USV-EXTENDS} \\ \frac{\Delta \vdash E <: T' \quad \Delta \vdash T <: E}{E <_? ?_T^T} \end{array} $	$ \begin{array}{c} \text{USV-EQUALS} \\ \frac{E = E'}{E <_? E'} \end{array} $
--	--

Well-formed types:

$ \begin{array}{c} \text{WF-OBJECT} \\ \Delta \vdash \text{Object OK} \end{array} $	$ \begin{array}{c} \text{WF-VAR} \\ \frac{X \in \text{dom}(\Delta)}{\Delta \vdash X \text{ OK}} \end{array} $	$ \begin{array}{c} \text{WF-CLASS} \\ \frac{\text{class } C\langle \bar{X} \triangleleft \bar{N} \rangle \triangleleft N \{ \dots \} \quad \Delta \vdash \bar{E} \text{ OK} \quad \Delta \vdash \bar{E} <: [\bar{E}/\bar{X}]\bar{N}}{\Delta \vdash C\langle \bar{E} \rangle \text{ OK}} \end{array} $
$ \begin{array}{c} \text{WF-WILDCARD} \\ \frac{\Delta \vdash T <: T'}{\Delta \vdash ?_T^T \text{ OK}} \end{array} $		

Figure 4: Subtyping

S-TRANS

The subtype ordering is transitive not only for type variables and type terms, but for wildcard types, too.

The rules **S-VAR**, **S-CLASS** are unchanged. Additionally, there are four new rules.

S-WC-CLASS

This rule introduces the use-side variance. Therefore, another subtyping relation, the use-side variance subtyping $\leq_?$, is introduced. This relation defines if two type in argument position are in subtype relation.

E.g. as `List<Integer>` is a subtype of `List<? extends Object>` it holds

$$\text{Integer} \leq_? \text{?}_{Null}^{\text{Object}}$$

It is important that the extended type $C'<\bar{x}>$ in the declared class is not a nested type as in nested supertypes it is not allowed to instantiate wildcards.

The three rules **S-Super-Object**, **S-WC-Extends**, and **S-WC-Super** extends the subtyping relation $<:$ to wildcard types.

S-Super-Object

As `Object` is the maximum in Java type hierarchy all types are subtypes of `Object`, especially $\text{?}_{T'}^{T'}$ for any T, T'

S-WC-Super

All supertype of a given type T' are also supertypes of any subtype T of the given type T' . E.g. $\text{Integer} <: \text{?}_{Number}^{\text{Object}}$

S-WC-Extends

All subtypes of a given type T are also subtypes of any supertype T' of the given type T . E.g. $\text{?}_{Null}^{\text{Integer}} <: \text{Number}$.

With the **S-TRANS** rule for a subtype relationship $T <: T'$ holds especially $\text{?}_{T''}^T <: \text{?}_{T'}^{T''}$.

USV-Extends

The **USV-Extends** rule defines the use-side variance of **FJ-TX**. Covariance is defined by $E \leq_? \text{?}_T^{T'}$ as $E <: T'$ which means $C(\dots E \dots) <: C'(\dots \text{? extends } T' \dots)$ and contravariance as $T <: E$ which means $C(\dots E \dots) <: C'(\dots \text{? super } T \dots)$

USV-Equals

The **USV-Equals** rule guarantees invariant subtyping.

The rules for well-formed types are nearly unchanged. Only the rule **WF-Wildcard** is added which guarantees that the lower bound is a subtype of the upper bound. allowed in **FJ-TX**.

2.2.2 Class, method, and exptype typing rule

The rules in Fig. 5 are structurally unchanged. Only the different types are adopted.

There is one difference in the presentation. While in [6] in each class a constructor is given, explicitly, we do not present the constructors, explicitly. But we assume that in each class a constructor is given, implicitly.

In the following we give two examples to show the idea of the calculus.

Example 2.1. Let the class `List` be given.

```
class List<T> {
    T elem;
    List<T> rest;

    List<T> addElement(T newElem) {
        return new List<>(newElem, this);
    }

    List<T> append(List<? extends T> l) {
        return new List(elem, rest.append(l));}
}

class Empty<T> extends List<T> {
    List<T> append(List<? extends T> l)
    { return l; } ⚡
}
```

Bound:	
$bound_{\Delta}(X) = U$, where $(X, U) \in \Delta, X \neq U$	
$bound_{\Delta}(E) = E$	
Field lookup:	
$fields(Object) = \emptyset$	$\frac{class\ C \langle \bar{X} \triangleleft \bar{T}' \rangle \triangleleft T \{ \bar{T} \bar{f}; \dots \}}{fields(C \langle \bar{E} \rangle) = \bar{U} \bar{g}, [\bar{E}/\bar{X}] \bar{T} \bar{f}}$
Method lookup:	
$\frac{class\ C \langle \bar{X} \triangleleft \bar{T}' \rangle \triangleleft T \{ \bar{T} \bar{f}; \bar{M} \}}{\langle \bar{Y} \triangleleft \bar{T}' \rangle V m(\bar{V} \bar{x}) \{ return\ e_0; \} \in \bar{M}}$	$\frac{class\ C \langle \bar{X} \triangleleft \bar{T}' \rangle \triangleleft T \{ \bar{T} \bar{f}; \bar{M} \}}{mtype(m, C \langle \bar{E} \rangle) = mtype(m, [\bar{T}/\bar{X}] T)}$
Expression typing:	
$\Delta; \Gamma \vdash x : \Gamma(x)$	(GT-VAR)
$\frac{\Delta; \Gamma \vdash e_0 : E_0 \quad fields(bound_{\Delta}(E_0)) = \bar{T} \bar{f}}{\Delta; \Gamma \vdash e_0.f_i : T_i}$	(GT-FIELD)
$\frac{\Delta; \Gamma \vdash e_0 : E_0 \quad mtype(m, bound_{\Delta}(E_0)) = \langle \bar{Y} \triangleleft \bar{T}' \rangle \bar{T}'' \rightarrow T''}{\Delta \vdash \bar{F} OK \quad \Delta \vdash \bar{F} \langle : [\bar{F}/\bar{Y}] \bar{T} \quad \Delta; \Gamma \vdash \bar{e} : \bar{G} \quad \Delta \vdash \bar{G} \langle : [\bar{F}/\bar{Y}] \bar{T}''}}{\Delta; \Gamma \vdash e_0.m(\bar{e}) : [\bar{F}/\bar{Y}] T''}$	(GT-INVK)
$\frac{\Delta \vdash N OK \quad N = C \langle \bar{U} \rangle \quad fields(N) = \bar{T} \bar{f} \quad \Delta; \Gamma \vdash \bar{e} : \bar{E} \quad \Delta \vdash \bar{E} \langle : \bar{T}}{\Delta; \Gamma \vdash new\ C(\bar{e}) : N}$	(GT-NEW)
$\frac{\Delta \vdash N OK \quad \Delta; \Gamma \vdash e_0 : E_0}{\Delta; \Gamma \vdash (N)e_0 : N}$	(GT-CAST)
where E_0, F , and G are any FJ-TX types including wildcard types.	
Method typing:	
$\frac{\Delta = \bar{X} \langle : \bar{T}', \bar{Y} \langle : \bar{T}'' \quad \Delta \vdash \bar{T}', \bar{T}'', \bar{T}, \bar{T}, E_0 OK}{\Delta; \bar{x} : \bar{T}, this : C \langle \bar{X} \rangle \vdash e_0 : E_0 \quad \Delta \vdash E_0 \langle : T}}{class\ C \langle \bar{X} \triangleleft \bar{T}' \rangle \triangleleft N \{ \dots \} \quad override(m, N, \langle \bar{Y} \triangleleft \bar{T}'' \rangle \bar{T} \rightarrow T)}{\langle \bar{Y} \triangleleft \bar{T}' \rangle T m(\bar{T} \bar{x}) \{ return\ e_0; \} OK\ in\ C \langle \bar{X} \triangleleft \bar{T}' \rangle}$	(GT-METHOD)
Class typing:	
$\frac{\bar{X} \langle : \bar{T}' \vdash \bar{T}', \bar{T} OK \quad \bar{M} OK\ in\ C \langle \bar{X} \triangleleft \bar{T}' \rangle}{class\ C \langle \bar{X} \triangleleft \bar{T}' \rangle \triangleleft N \{ \bar{T} \bar{f}; \bar{M} \} OK}$	(GT-CLASS)

Figure 5: Typing rules

As `List<? extends T>` is no subtype of `List<T>` the statement `return l`; is not type correct although in this case no problem would appear. The reason is that all elements of `l` are introduced into the return-list. Therefore, the solution of this problem is that we add elementwise the elements of `l` to the return-list.

```
List<T> append(List<? extends T> l) {
    return
        new List<T>(l.elem,
                    addElem(l.rest));
}
```

```
List<T> addElem(List<? extends T> l) {
    return
        new List<T>(l.elem,
                    addElem(l.rest));
}
```

- $N = \text{List}\langle T \rangle$
- $\text{fields}(\text{List}\langle T \rangle) = (T \text{ elem}, \text{List}\langle T \rangle \text{ rest})$
- $l.\text{elem} : ?_{Null}^T$
- $\text{addElem}(l.\text{rest}) : \text{List}\langle T \rangle$
- $?_{Null}^T <: T$
- $\text{List}\langle T \rangle <: \text{List}\langle T \rangle$

$\Rightarrow \text{new List}\langle T \rangle(l.\text{elem}, \text{addElem}(l.\text{rest})) : \text{List}\langle T \rangle$

The next example show the influence of the changed S-REFL rule (instead of T, types including type variables, in this calculus U, types without type variables).

Example 2.2. Let `List<X>` given as before with an additional method `get` which determines the nth element of the list.

```
<X extends Object> List<List<X>>
    shuffle(List<List<X>> l){
        l.addElement(addElement(l.get
            (0).get(0)));
    }
```

The method is not type correct as `l.get(0).get(0)` has the type `X` and `addElement` waits for a `X`. But with the S-REFL rule $X \not<: X$. If we change the bound of `X` from `Object` to `X`

```
<X extends X> List<List<X>>
    shuffle(List<List<X>> l){
        l.addElement(addElement(l.get
            (0).get(0)));
    }
```

with the **S-VAR** rule $X <: X$, such that the method is type correct.

Another method

```
void error(List<List<?>> anyL){
    shuffle<?>(anyL);
}
```

is not type correct, as in the GT-INVK rule the precondition $\Delta \vdash ? <: [?/X]X$ must hold which is not given.

Despite this any non wildcard instantiation of `X` is type correct.

3 Operational semantics and soundness

In Fig. 6 the operational semantics of **FJ-TX** is defined by reduction rules. The rules are unchanged in comparison to **FGJ**.

Definition 3.1 (FJ-TX value). A **FJ-TX** *value* is defined as

$$v ::= \text{new } C(\bar{v})$$

Definition 3.2 (Normal form). A well-types **FJ-TX** expression `e` is in *normal form* if it is a value.

Theorem 3.3. If

$$\emptyset; \emptyset \vdash e : T \text{ and } e \longrightarrow^* e' \text{ (in normal form),}$$

then `e'` is either

(1) a **FJ-TX** value `w` with

$$\emptyset; \emptyset \vdash e' : S$$

and

$$\emptyset \vdash S <: T$$

or

(2) an expression containing `(P)new N(\bar{e})` where $\emptyset \vdash N \not<: P$.

Computation:	
$\frac{fields(N) = \bar{T} \bar{f}}{(new\ N(\bar{e})) . f_i \longrightarrow e_i}$	(GR-FIELD)
$\frac{mbody(m\langle\bar{V}\rangle, N) = \bar{x} . e_0}{(new\ N(\bar{e})) . m\langle\bar{V}\rangle(\bar{d}) \longrightarrow [\bar{d}/\bar{x}, new\ N(\bar{e})/this]e_0}$	(GR-INVK)
$\frac{\emptyset \vdash N <: P}{(P)\ (new\ N(\bar{e})) \longrightarrow new\ N(\bar{e})}$	(GR-CAST)
Congruence:	
$\frac{e_0 \longrightarrow e'_0}{e_0 . f \longrightarrow e'_0 . f}$	(GRC-FIELD)
$\frac{e_0 \longrightarrow e'_0}{e_0 . m\langle\bar{T}\rangle(\bar{e}) \longrightarrow e'_0 . m\langle\bar{T}\rangle(\bar{e})}$	(GRC-INV-RECV)
$\frac{e_i \longrightarrow e'_i}{e_0 . m\langle\bar{T}\rangle(\dots, e_i, \dots) \longrightarrow e_0 . m\langle\bar{T}\rangle(\dots, e'_i, \dots)}$	(GRC-INV-ARG)
$\frac{e_i \longrightarrow e'_i}{new\ N(\dots, e_i, \dots) \longrightarrow new\ N(\dots, e'_i, \dots)}$	(GRC-NEW-ARG)
$\frac{e_0 \longrightarrow e'_0}{(N)e_0 \longrightarrow (N)e'_0}$	(GRC-CAST)

Figure 6: FJ-TX: Reduction Rules

4 Related Work

Igarashi et al [6] define Featherweight Java and its generic sibling, Featherweight Generic Java. Their language is a functional calculus reduced to the bare essentials, they develop the full metatheory, they support generics, and study the type erasing transformation used by the Java compiler. Stadelmeier et. al. extends this approach by global type inference [8].

Plümicke presented in [7] a type unification algorithm for Java type terms. He defined the subtyping relation very similar to our approach. He proved the type inference algorithm as sound and complete.

Wildcards are first described in a research paper in [10]. In [9] the Featherweight Java-Calculus Wild FJ is introduced. It contains a formal description of wildcards. The Java Language Specification [5] refers to Wild FJ for the introduction of wildcards. In [4] a formal model based of explicit existential types is introduced and proven as sound. Additionally, for a subset of Java a translation to the formal model is given, such that this subset is proven as sound. In [3] another core calculus is introduced, which is proven as sound,

too. In this paper it is shown that the unsoundness of Java which is discovered in [2] is avoidable, even in the absence of nullness-aware type system. In [1] finally a framework is presented which combines use-site variance (wildcards as in Java) and definition-site variance (as in Scala). For instance, it can be used to add use-site variance to Scala and extend the Java type system to infer the definition-site variance.

All these approaches uses the so-called capture conversion (cp. Def. 1.1) in method-calls with wildcards. In contrast in our calculus wildcards are instantiated, directly.

5 Summary and Outlook

In this paper introduced a calculus **FJ-TX**. The calculus corresponds to a core of Java-TX. The calculus is a new approach to deal with wildcards. Wildcards are introduced into Java in version 5. They are introduced to realize use-site variance. For the soundness of the language so-called capture conversions are introduced which converts each wildcard into an own type variable.

Our calculus avoids the capture conversion. Therefore, we extend the subtyping ordering by wildcards as types. This extension is implemented in Java-TX.

With this approach some sound programs become type correct that are not type correct by using capture conversion.

We formulate a theorem that capture conversion can be avoided if no type variables that are in sub-type relationship are equalized.

On the other hand some programs which are type correct in the capture conversion approach are not type correct in our approach.

In the future, we have to prove the theorem. Furthermore we have compare the wildcard approaches with and without capture conversion.

References

- [1] John Altidor, Shan Shan Huang, and Yannis Smaragdakis. “Taming the Wildcards: Combining Definition- and Use-Site Variance”. In: vol. 46. June 2011, pp. 602–613.
- [2] Nada Amin and Ross Tate. “Java and scala’s type systems are unsound: the existential crisis of null pointers”. In: Oct. 2016, pp. 838–848.
- [3] Kevin Bierhoff. “Wildcards Need Witness Protection”. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022).
- [4] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. “A Model for Java Wildcards”. In: *ECOOP 08*. Vol. 5142. Lecture Notes in Computer Science. June 2008. URL: <http://pubs.doc.ic.ac.uk/wildcards-ecoop-08/>.
- [5] James Gosling et al. *The Java[®] Language Specification*. Java SE 21. 2023. URL: <https://docs.oracle.com/javase/specs/jls/se21/jls21.pdf>.
- [6] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. “Featherweight Java: a minimal core calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3 (2001), pp. 396–450.
- [7] Martin Plümicke. “Java Type Unification with Wildcards”. In: *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4–6, 2007, Revised Selected Papers*. Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf. Vol. 5437. Lecture Notes in Computer Science. Springer, 2007, pp. 223–240.
- [8] Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. “Global Type Inference for Featherweight Generic Java”. In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Leibniz International Proceedings in Informatics (LIPIcs) 222 (2022). Ed. by Karim Ali and Jan Vitek, 28:1–28:27. ISSN: 1868-8969.
- [9] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. “Wild FJ”. In: *Proceedings of FOOL 12*. Ed. by Philip Wadler. ACM. Long Beach, California, USA: School of Informatics, University of Edinburgh, Jan. 2005. URL: <http://homepages.inf.ed.ac.uk/wadler/fool/>.
- [10] Mads Torgersen et al. “Adding wildcards to the Java programming language”. In: *Journal of Object Technology* 3.11 (Dec. 2004), pp. 97–116.

IMPRESSUM

Schriftenreihe INSIGHTS
Themenreihe Engineering INSIGHTS

Herausgeber:

Fakultät Technik der
Dualen Hochschule Baden-Württemberg Stuttgart
Postfach 10 05 63, 70004 Stuttgart

Prof. Dr.-Ing. Harald Mandel

Prorektor für Forschung, Transfer und Nachhaltigkeit & Dekan Fakultät Technik
Lerchenstraße 1, 70174 Stuttgart

E-Mail: harald.mandel@dhbw-stuttgart.de

Tel.: +49 (0)711 1849-605

www.dhbw-stuttgart.de/technik/insights

Umschlaggestaltung: Kerstin Faißt

Bildnachweis: Gerd Altmann auf Pixabay

ISSN 2193-9098

© Daniel Holle, Prof. Dr. Jens Knoop, Prof. Dr. habil. Martin Plümicke, Prof. Dr. Peter Thiemann,
Prof. Dr. habil. Baltasar Trancón y Widemann (Hrsg.), 2024

Alle Rechte vorbehalten. Der Inhalt dieser Publikation unterliegt dem deutschen Urheberrecht.
Die Vervielfältigung, Bearbeitung, Verbreitung und jede Art der Verwertung außerhalb der Grenzen
des Urheberrechtes bedürfen der schriftlichen Zustimmung der Autorinnen und Autoren und der
Herausgeberin.

Der Inhalt der Publikation wurde mit größter Sorgfalt erstellt. Für die Richtigkeit, Vollständigkeit und
Aktualität des Inhalts übernimmt der Herausgeber keine Gewähr.

ISSN 2193-9098

www.dhbw-stuttgart.de/technik/insights