DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart

Engineering

INSIGHTS

Schriftenreihe der Fakultät Technik: 1/2024

Tagungsband der Jahrestreffen 2021–2023 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

Vorwort

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

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

Dieser Tagungsband ist in erster Linie dem Treffen des Jahres 2023 gewidmet, dem 39. Workshop der Reihe, es wurden aber auch Beiträge der Vorjahre 2021 und 2022 aufgenommen. Der für 2020 geplante 37. Workshop der Reihe musste leider aufgrund von Reise- und Versammlungsrestriktionen zur Eindämmung der COVID-19 Pandemie abgesagt werden. In 2021 wurde ein Workshop in einem virtuellen Format abgehalten. Seit dem 38. Workshop 2022 finden die Treffen wieder in Präsenz am gewohnten Ort statt.

Beim 39. Workshop 2023 gab es Sitzungen zu allen Kernthemen des Gebietes, nämlich Syntax, Semantik und Pragmatik von Programmiersprachen, sowie die Schwerpunktthemen Typsysteme und deklaratives Programmieren. Daneben wurden auch zahlreiche Prototypen von Werkzeugen demonstriert.

Allen Teilnehmenden gilt der Dank, dass sie durch ihre Vorträge, Papiere und Diskussion den jährlichen Workshop regelmäßig zu einem interessanten und anregenden Ereignis machen. Ein besonderer Dank gebührt den Mitarbeiterinnen und Mitarbeitern des Physikzentrums Bad Honnef, die auch in schwierigen post-pandemischen Zeiten durch ihre umfassende Betreuung für eine angenehme und anregende Atmosphäre gesorgt haben.

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

Inhaltsverzeichnis

Typdeduktion in MOSTflexiPL

Christian Heinlein
Studienbereich Informatik
Hochschule Aalen – Technik und Wirtschaft
christian.heinlein@hs-aalen.de

MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language, flexipl.info) bietet neben seiner einzigartigen syntaktischen Flexibilität auch ein interessantes Typsystem mit Ähnlichkeiten zu „dependent types“. Typdeduktion, d. h. die automatische Ermittlung von Typen durch den Compiler, ist hier an unterschiedlichen Stellen notwendig oder zumindest hilfreich.

In der einfachsten Form, kann bei der Deklaration einer Konstanten wie z. B. `N : int = 10` der Typ weggelassen werden (`N := 10`), weil er sich normalerweise (siehe unten) unmittelbar aus dem Typ des Initialisierungsausdrucks ergibt.

Die zweite Stelle, wo Typdeduktion wichtig ist, sind generische Operatoren, die auf Operanden mit unterschiedlichen Typen angewandt werden können. Der vordefinierte Klammeroperator könnte z. B. wie folgt in der Sprache selbst definiert werden, obwohl er natürlich vordefiniert ist:

```
[(T:type)] "(" (x:T) ")" -> (T = x)
```

Die Signatur eines Operators links vom Pfeil kann allgemein folgendes enthalten: Namen des Operators, d. h. entweder Folgen von Buchstaben und Ziffern, die mit einem Buchstaben beginnen, oder beliebige Zeichenfolgen in Anführungszeichen.

Parameterdeklarationen der Art `(Name:Typ)`, wobei der Typ auch fehlen kann (siehe unten).

Eckige Klammern, die wie in EBNF optionale Teile kennzeichnen.

Geschweifte Klammern zur Kennzeichnung von Teilen, die beliebig oft wiederholt werden dürfen.

Runde Klammern mit einer oder mehreren Alternativen, von denen genau eine gewählt werden muss. (Tatsächlich können auch eckige und geschweifte Klammern mehrere Alternativen enthalten, von denen dann maximal eine bzw. beliebig viele nacheinander gewählt werden können.) Rechts vom Pfeil steht in Klammern der Resultattyp des Operators (der auch fehlen kann, siehe unten) ein Gleichheitszeichen und ein Implementierungsausdruck, der den Resultatwert einer Operatoranwendung zur Laufzeit berechnet, typischerweise unter Verwendung der Parameterwerte.

Bei einer Anwendung des Klammeroperators wie z. B. `(a + b) * c` mit `a`, `b` und `c` vom Typ `int` besitzt der zum Parameter `x` gehörende Operand `a + b` ebenfalls den Typ `int`, sodass der optionale Typparameter `T` automatisch mit `int` belegt wird. Ähnliche Möglichkeiten gibt es in vielen anderen Programmiersprachen, wenn sie in irgendeiner Form generische Definitionen erlauben, z. B. mit „templates“ in C++ oder mit „generics“ in Java.

Die meisten Sprachen verlangen allerdings, dass sich solche Typparameter aus den Typen der anderen Parameter und damit aus den Typen der Operanden ermitteln lassen, sodass der Compiler die Typen von Ausdrücken immer „von unten nach oben“ ermitteln kann. MOSTflexiPL erlaubt jedoch auch eine Deduktion aus dem Resultattyp und damit aus dem Verwendungskontext eines Operators. Ein typisches Beispiel ist der vordefinierte Operator `nil`, der den nil-Wert eines beliebigen Typs liefert, vergleichbar mit `nullptr` in C++ oder `null` in Java. Dieser Operator könnte wie folgt in der Sprache selbst definiert werden:M

```
[(T:type)] nil -> (T = x:T?; ?x)
```

Hier kann der Typ `T` bei einer Verwendung des Operators, die nur aus dem Wort `nil` besteht, nur aus dem Verwendungskontext ermittelt werden. Beispielsweise ergibt sich bei einem Ausdruck wie

`nil + 1` der Typ `int` des Teilausdrucks `nil` erst, wenn dieser als Operand in den Additionsoperator eingesetzt wird, weil dieser Operanden des Typs `int` verlangt.[1]

Beim Ausdruck `(nil) + 1` wird der Teilausdruck `nil` mit zunächst unbekanntem Typ in den generischen Klammeroperator eingesetzt. Weil dieser Operanden mit beliebigem Typ akzeptiert, kann der Typ von `nil` hier noch nicht ermittelt werden, sodass auch der Resultattyp des Klammerausdrucks vorläufig unbekannt ist. Erst beim Einsetzen des Klammerausdrucks in den Additionsoperator ergeben sich beide unbekannten Typen als `int`.

Es ist wichtig zu beachten, dass jede Verwendung von `nil` einen anderen Typ besitzen kann, weil der Typparameter `T` – wie jeder andere Parameter – jedesmal einen anderen Wert besitzen kann. Im Beispiel `nil | nil < 1` ergibt sich, dass die erste Verwendung von `nil` Typ `bool` besitzen muss, weil der Disjunktionsoperator Operanden dieses Typs verlangt, während die zweite Verwendung von `nil` Typ `int` besitzen muss, weil der Kleiner-Operator Operanden dieses Typs verlangt.

Wenn eine Konstante ohne expliziten Typ mit dem generischen Wert `nil` initialisiert wird, z. B. `x := nil`, kann ihr Typ nicht wie sonst aus der Initialisierung ermittelt werden. In diesem Fall bleibt dieser Typ so lange unbekannt, bis er sich aus einer Verwendung der Konstanten wie z. B. `x + 1` oder `(x) + 1` eindeutig ergibt. Anders als der generische Operator `nil`, der bei jeder Verwendung einen anderen Typ besitzen kann, besitzt eine Konstante wie `x` aber einen einzigen festen Typ, d. h. aus allen nachfolgenden Verwendungen von `x` muss sich derselbe Typ ergeben.

Obwohl die bis jetzt gezeigten Beispiele alle künstlich sind, demonstrieren sie trotzdem wichtige Aspekte, die bei der Typdeduktion beachtet werden müssen.

Schließlich kann Typdeduktion auch verwendet werden, um Parameter- und Resultattypen von Operatoren vom Compiler automatisch bestimmen zu lassen, zum Beispiel:

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

[1] Die Deklaration `x:T?` vereinbart eine Variable `x` mit Inhaltstyp `T`, deren Wert von dem Ausdruck `?x` geliefert wird. Da der Variablen noch kein Wert zugewiesen wurde, enthält sie gerade den Wert nil. Semikolon beschreibt die Nacheinanderausführung von Teilausdrücken, die als Wert den Wert des rechten Teilausdrucks liefert.

Bei dieser Definition des Fakultätsoperators ergibt sich der Typ `int` des Parameters `n` bereits beim Einsetzen von `n` in den Vergleichsoperator, was mit den weiteren Verwendungen als Operand der Subtraktion und der Multiplikation konsistent ist. Der zunächst ebenfalls unbekannte Resultattyp des Operators ergibt sich aus der rekursiven Verwendung des Operators im linken Operanden der Multiplikation ebenfalls als `int`, was dann mit dem Typ des gesamten Implementierungsausdrucks `if ... end` übereinstimmt. (Der Typ einer solchen Fallunterscheidung stimmt mit den Typen des zweiten und dritten Operanden überein.)

Zur Implementierung der beschriebenen Typdeduktionen erzeugt der Compiler für jeden zunächst unbekannten Typ einen eindeutigen Platzhalter. Wenn ein solcher Platzhalter bei einem nachfolgenden Typvergleich, der z. B. beim Einsetzen eines Operanden in einen übergeordneten Ausdruck ausgeführt wird, mit einem bereits bekannten Typ verglichen wird, kann der Platzhalter mit diesem Typ belegt werden. Wenn zwei noch unbelegte Platzhalter bei einem solchen Vergleich aufeinandertreffen, ergibt sich zumindest, dass beide Platzhalter denselben unbekannten Typ repräsentieren müssen, sodass willkürlich einer von beiden mit dem anderen belegt werden kann. Damit ein MOSTflexiPL-Programm abschließend typkorrekt ist, müssen am Ende alle Platzhalter direkt oder indirekt mit einem bekannten Typ belegt sein.

Modellierung von kollaborativen Spreadsheets mit CRDTs

Elena Yanakieva, Philipp Bird, Annette Bieniusa
RPTU Kaiserslautern-Landau
Fachbereich Informatik
{yanakieva, p_bird14, bieniusa}@cs.uni-kl.de

Abstract

Das kollaborative Bearbeiten mit Hilfe digitaler Tools setzt sich - nicht zuletzt auf Grund der Corona-Pandemie - in unserem Alltag immer weiter durch. Neben Textdokumenten ist die gemeinsame Bearbeitung von Tabellen (*spreadsheets*) ein häufiger Anwendungsfall. Dabei ist das nebenläufige Modifizieren mehrerer Nutzer in der Regel problematisch: Wie sollen Editierkonflikte aufgelöst werden? Oft gibt es mehrere Möglichkeiten, die aber unter Umständen nicht die Absicht der Bearbeitenden reflektieren. So haben beispielsweise in existierenden Softwareprodukten Löschoperationen Vorrang vor Editieroperationen.
In unserem Vortrag präsentieren wir eine systematische Diskussion der möglichen nebenläufigen Aktionen, die man in Spreadsheets vornehmen kann und deren nebenläufige Semantik. Weiterhin schlagen wir solche dezentralisierte Designs vor, die die Benutzerabsicht bewahren und gleichzeitig die offline Arbeit unterstützen. Unsere Datenmodelle basieren auf zusammengestellte Conflict-free Replicated Data Types (CRDTs).

1 Motivation

Kollaborative Anwendungen ermöglichen das Teilen und gemeinsame Bearbeitung von Dokumenten durch verschiedene Benutzer in Echtzeit. Sie sind mittlerweile ein wichtiges Tool in unserem privaten Umfeld und Arbeitsalltag. Je nach Art der geteilten Daten eignen sich unterschiedliche Arten der Repräsentierungen, insbesondere text-, bild- und tabellenbasierte Darstellungen.

Um solche Anwendungen zu implementieren, eignen sich Conflict-free Replicated Data Types (CRDTs) [6]. CRDTs garantieren eventuelle Konsistenz der Daten bei optimistischer nebenläufiger Änderung, wie sie sowohl bei dezentralen Anwendungen als auch bei (temporären) Verbindungsunterbrechungen wie in Offline-first Software auftritt. Jedoch müssen die Entwickler die Semantik der Anwedung detailliert analysieren, um bei Updates Konflikten im Sinne des Nutzers mit Hilfe der CRDT-spezifischen Strategie aufzulösen.

Kollaborative text-basierte Applikationen werden dazu seit Jahren beforscht. Zahlreiche Ansätze sind dabei entwickelt worden, darunter Treedoc [5], Logoot [7], LSEQ [3], RGA [1], und YATA [4]. Diese Arbeiten betrachten einen Text als Sequenz von Zeichen und benutzen unterschiedliche Techniken, um das Editieren an der gleichen Position so aufzulösen, dass die Änderungen eines Nutzers möglichst zusammenhängend in der Datenstruktur verbleiben und beispielsweise in Markdown-Dokumenten Invarianten wie das korrekte Verschachteln von Textauszeichnungen zu gewährleisten [2].

Ähnlich zum Text muss auch bei Spreadsheets sichergestellt werden, dass bei einer nebenläufigen Bearbeitung bzw. dem Zusammenführen bearbeiteter Tabllen deren Struktur nicht verletzt wird. Jedoch existieren (unseres Wissens) bislang keine CRDT-Ansätze zu Spreadsheets. Kommerzielle Produkte, wie Google

Sheets[1] und Notion[2] zeigen, dass eine systematische Erforschung kollaborativer Spreadsheets ein Desiderat darstellt. So wird beispielsweise in Google Sheets scheinbar arbiträr der Funktionsumfang der Applikation eingeschränkt, wenn man offline ist. Notion wiederum verursacht kritische Datenverluste: Bearbeitet eine Person eine Tabelle offline, während zeitgleich eine weitere Person online Änderungen vornimmt, werden die Änderungen der online agierenden Person ohne Warnung überschrieben und sind somit verloren.

Im folgenden skizzieren wir eine systematische Diskussion der möglichen Semantiken eines kollaborativen Spreadsheets. Wir schlagen außerdem ein Datenmodell vor, das aus zusammengesetzen CRDTs besteht und stellen eine prototypische Implementierung bereit, um die praktische Umsetzung unseres Konzepts zu validieren. Für eine detaillierte Präsentation verweisen wir auf [8].

2 Kollaboratives Spreadsheet-Design mittels CRDTs

Ein Spreadsheet ist ein elektronisches Dokument, um Daten in tabellarischer Form zu organisieren. Es besteht aus einem Zellengitter, angeordnet in Spalten und Zeilen. Jede Zelle kann einen Dateneintrag beinhalten (z.B. eine Nummer, Text usw.)[3].

Jede Spalte und Zeile ist eindeutig identifizierbar. Wir beschränken uns dabei auf endlich viele Spalten und Zeilen.

Die möglichen Nutzeraktionen sind wie folgt definiert:

- `add_row()` fügt eine Zeile am Ende hinzu,
- `add_column()` fügt eine Spalte am Ende hinzu,
- `remove_row(r)` entfernt Zeile `r`,
- `remove_column(c)` entfernt Spalte `c`,
- `insert_column(c)` fügt eine Spalte vor Spalte `c` hinzu,
- `insert_row(r)` fügt eine Zeile vor Zeile `r` hinzu, und
- `edit_cell(c,r,new_content)` ändert der Zelleneintrag der Zeile `r` und Spalte `c` auf `new_content`.

Bei nebenläufiger Ausführung dieser Basisoperationen haben wir die folgenden diskussionsbedürftigen Szenarien identifiziert und schlagen folgende Semantiken vor:

- Gleichzeitiges Löschen derselben Spalte (bzw. Zeile) soll die Spalte (bzw. Zeile) genau einmal entfernen.
- Gleichzeitiges Hinzufügen von Spalten (bzw. Zeilen) an derselben Stelle soll die Spalten (bzw. Zeilen) so anordnen, dass die Reihenfolge bei allen Nutzern – nach Synchronisierung der Operation – gleich ist. Dafür können etablierte Ansätze aus der Literatur für text-basierte Applikationen verwendet werden [5, 7].
- Um den Konflikt bei gleichzeitigen Änderungen am selben Zelleneintrag zu lösen, kann man z.B. die Einträge verschmelzen oder den zuletzt geschriebenen Eintrag übernehmen.
- In dem Fall, dass eine Zelle modifiziert wird und gleichzeitig die dazugehörige Spalte (bzw. Zeile) entfernt wird, haben wir mehrere Möglichkeiten identifiziert: 1. die Spalte/Zeile wird entfernt und somit auch die Zelle (**Remove-wins** Semantik); 2. eine neue Spalte/Zeile wird erstellt, die nur den neuen Zelleneintrag enthält (**Remove-reset** Semantik); oder 3. das Löschen wird übergangen (**Update-wins** Semantik).

Die **Update-wins** Semantik ist dabei interessant, da sie Datenverlusten aufgrund von Präzedenz der Löschoperationen vermeidet. Um die *Update-wins* Semantik zu implementieren, schlagen wir eine Datenmodellierung mit Hilfe zweier Array CRDTs vor, um die Reihenfolge der Zeilen und Spalten zu verwalten. Die Zelleneinträge können in einer Map-CRDT gesammelt werden. Zwei weitere Map-CRDTs (**Keep-Maps**), enthalten die notwendigen Metadaten, um nebenläufiges Löschen zu identifizieren, so dass die Spalte/Reihe nicht entfernt wird, wenn eine gleichzeitige Zellenänderung passiert ist.

Die Keep-Maps ordnen bei jedem Spalten- und Zeile-Identifier eine Menge von Operationenzeitstempeln zu. Bei jeder Operation, z.B. Änderung des Zelleninhalts oder Entfernen einer Spalte, werden diese Tags entfernt. In dem Fall der Änderung des Zelleninhalts werden diese noch dazu durch einen neuen Tag ersetzt. Eine Spalte/Zeile

[1] `https://www.google.com/sheets/about/`
[2] `https://www.notion.so/`
[3] Wir schränken uns hier auf ein einzelnes Spreadsheet ein. Die Unterstützung von Formeln und Diagrammen ist orthogonal zu unserer Arbeit hier und wird nicht näher diskutiert.

wird genau dann angezeigt, wenn mindestens ein Wert in dem entsprechenden Array vorliegt, d.h. die letzte Modifikation keine Löschoperation war.

Eine prototypische Implementierung des Ansatzes mittels der Yjs Frameworks [9] ist unter `https://github.com/Roffelchen/spreadsheet-crdt` als Open-Source Projekt verfügbar.

Literatur

[1] Loïck Briot, Pascal Urso, and Marc Shapiro. "High Responsiveness for Group Editing CRDTs". In: *Proceedings of the 19th International Conference on Supporting Group Work, Sanibel Island, FL, USA, November 13 - 16, 2016*. ACM, 2016, pp. 51–60. URL: https://doi.org/10.1145/2957276.2957300.

[2] Geoffrey Litt et al. "Peritext: A CRDT for Collaborative Rich Text Editing". In: *Proc. ACM Hum. Comput. Interact.* 6.CSCW2 (2022), pp. 1–36. URL: https://doi.org/10.1145/3555644.

[3] Brice Nédelec et al. "LSEQ: an adaptive structure for sequences in distributed collaborative editing". In: *ACM Symposium on Document Engineering 2013, DocEng '13, Florence, Italy, September 10-13, 2013*. ACM, 2013, pp. 37–46. URL: https://doi.org/10.1145/2494266.2494278.

[4] Petru Nicolaescu et al. "Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types". In: *Proceedings of the 19th International Conference on Supporting Group Work, Sanibel Island, FL, USA, November 13 - 16, 2016*. ACM, 2016, pp. 39–49. URL: https://doi.org/10.1145/2957276.2957310.

[5] Nuno M. Preguiça et al. "A Commutative Replicated Data Type for Cooperative Editing". In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 2009, pp. 395–403. URL: https://doi.org/10.1109/ICDCS.2009.20.

[6] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Vol. 6976. Lecture Notes in Computer Science. Springer, 2011, pp. 386–400. URL: https://doi.org/10.1007/978-3-642-24550-3%5C_29.

[7] Stéphane Weiss, Pascal Urso, and Pascal Molli. "Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks". In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 2009, pp. 404–412. URL: https://doi.org/10.1109/ICDCS.2009.75.

[8] Elena Yanakieva, Philipp Bird, and Annette Bieniusa. "A Study of Semantics for CRDT-based Collaborative Spreadsheets". In: *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2023, Rome, Italy, 8 May 2023*. Ed. by Elisa Gonzalez Boix and Pierre Sutra. ACM, 2023, pp. 37–43. URL: https://doi.org/10.1145/3578358.3591324.

[9] *YJS shared editing*. URL: https://yjs.dev/.

Detecting Data-Flow Anomalies in BPMN-Based Process-Driven Applications

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

Abstract

Process-Driven Applications (PDAs) flourish through the interaction between an executable BPMN process model, human tasks, and external software services. All these components operate on shared process data, so it is all the more important to check the correct data flow. However, data flow is in most cases not explicitly defined but hidden in model elements, form declarations, and program code. We elaborate on data-flow anomalies acting as indicators for potential errors and how such anomalies can be detected despite implicit and hidden data-flow definitions. By considering an integrated view, it goes beyond other approaches which are restricted to separate data-flow analysis of either process model or source code. It also improves previous work which suffered from rather primitive means to uncover accesses to process data in the source code. The main idea is to merge call graphs representing programmed services into a control-flow representation of the process model, to label the resulting graph with associated data operations, and to detect anomalies on that labeled graph using a dedicated data-flow analysis. The applicability of the solution is demonstrated by a prototype designed for the Camunda BPM platform. It can be employed, e.g., as additional static analysis check within a deployment pipeline. More detailed information, also on test-case generation and test-case maintenance for PDAs, can be found in [1–5].

References

[1] Konrad Schneid, Sebastian Thöne, and Herbert Kuchen. “Modification-Impact based Test Prioritization for Process-Driven Applications”. In: *IEEE International Conference on Software Testing, Verification and Validation, ICST 2023 - Workshops, Dublin, Ireland, April 16-20, 2023*. IEEE, 2023, pp. 365–372. URL: https://doi.org/10.1109/ICSTW58534.2023.00068.

[2] Konrad Schneid, Sebastian Thöne, and Herbert Kuchen. “Semi-automated Test Migration for BPMN-Based Process-Driven Applications”. In: *Enterprise Design, Operations, and Computing - 26th International Conference, EDOC 2022, Bozen-Bolzano, Italy, October 3-7, 2022, Proceedings*. Ed. by João Paulo A. Almeida et al. Vol. 13585. Lecture Notes in Computer Science. Springer, 2022, pp. 237–254. URL: https://doi.org/10.1007/978-3-031-17604-3%5C_14.

[3] Konrad Schneid et al. “Automated Regression Tests: A No-Code Approach for BPMN-based Process-Driven Applications”. In: *25th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2021, Gold Coast, Australia, October 25-29, 2021*. IEEE, 2021, pp. 31–40. URL: https://doi.org/10.1109/EDOC52215.2021.00014.

[4] Konrad Schneid et al. “Static analysis of BPMN-based process-driven applications”. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*. Ed. by Chih-Cheng Hung and George A. Papadopoulos. ACM, 2019, pp. 66–74. URL: https://doi.org/10.1145/3297280.3297289.

[5] Konrad Schneid et al. “Uncovering data-flow anomalies in BPMN-based process-driven applications”. In: *SAC ’21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*. Ed. by Chih-Cheng Hung et al. ACM, 2021, pp. 1504–1512. URL: https://doi.org/10.1145/3412841.3442025.

Mutating Sample Solutions to Improve Prolog Exercise Tasks and Their Test Suites

Ivan Khu Tanujaya, Janis Voigtländer, Oliver Westphal
University of Duisburg-Essen
{janis.voigtlaender, oliver.westphal}@uni-due.de

Abstract

We report on a tool for analyzing and adapting test suites for Prolog in an educational setting. The key idea is to mutate a known-correct sample solution in order to "simulate" possible student mistakes, and use that to "harden" the test suite that an e-learning system uses for judging submissions and giving feedback.
Specifically, at our department, Prolog is taught in two courses with non-overlapping audiences, including one where it precedes the introductory programming course. In both cases, practical exercises play an important role in the teaching and self-study. That is, students are given weekly Prolog modeling/programming tasks which they are supposed to solve and submit to an online tutoring system. The system gives immediate feedback about correctness of the submission and potentially information about program misbehavior, helping the students to improve their solution attempts step by step. The feedback functionality depends on testing, and thus, on appropriate test suites written by the instructors beforehand. It turns out that assembling appropriate collections of test cases can be challenging. Even after several semesters and iterations with similar or even the same exercise tasks, we still encounter student submissions that are accidentally mis-characterized as correct by the system (more seldom the opposite situation), or where the feedback is not as helpful as we would hope. So, to help us in the role of instructors, a tool was developed that assists in test suite construction, based on mutation testing [1, 3]. The tool comes with a parameterizable and extensible catalog of mutations, drawing on earlier work [2, 5], but trying to put emphasis on (new) mutations effective in our educational application, as well as a GUI that supports a workflow we found useful. In the talk and the extended article [4], we report on the rationale, realization, and experiences.

References

[1] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". In: *Computer* 11.4 (1978), pp. 34–41.

[2] A. Efremidis et al. "Measuring Coverage of Prolog Programs Using Mutation Testing". In: *26th International Workshop on Functional and Constraint Logic Programming, WFLP 2018, Revised Selected Papers*. Ed. by J. Silva. Vol. 11285. Lecture Notes in Computer Science. Springer, 2018, pp. 39–55.

[3] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678.

[4] I.K. Tanujaya, J. Voigtländer, and O. Westphal. “Mutating Sample Solutions to Improve Prolog Exercise Tasks and Their Test Suites”. In: *36th Workshop on (Constraint) Logic Programming, WLP 2022*. URL: https://wlp2022.dfki.de/data/papers/004.pdf.

[5] J. Toaldo and S. Vergilio. “Applying Mutation Testing in Prolog Programs”. In: *VII Workshop de Testes e Tolerância a Falhas, Proceedings*. 2006.

Determinism-Types for functional Logic Programming

Kai-Oliver Prott
Kiel University
Department of Computer Science
Christian-Albrechts-Platz 4, D–24118 Kiel
kpr@informatik.uni-kiel.de

1 Introduction

It is known that in functional logic languages, the combination of non-determinism and I/O operations is problematic. For instance, it is unclear what should happen during the execution of the following program:

```
main = writeFile "foo.txt" "content" ?
       writeFile "bar.txt" "content"
```

Here, `(?)` denotes the non-deterministic choice operator. Since the evaluation of `main` is non-deterministic, it is unclear whether the file `foo.txt` or `bar.txt` is written to (one cannot "duplicate the world"). In the functional logic language Curry [4], this program throws a run-time error, but it may execute at most one of the two I/O operations depending on the concrete implementation of the compiler.

By introducing a type system for determinism, we can avoid such misuses of non-determinism in I/O operations. The main idea is to distinguish between deterministic and non-deterministic functions in the type system. A similar approach was presented by Steimann for a lambda calculus with sequences or strings of terms [6]. With determinism types, one can safely approximate (similar to run-time type errors in strongly typed languages) the class of programs where such crashes do not occur. If the main operation of a program to be executed has determinism type Det, we can safely execute it. If it has type Any, our type system should allow us to pinpoint the source of (potential) non-determinism in a function.

While the main application of the type system is the safe execution of Curry programs containing I/O operation, we can also use the type information to optimize the execution of Curry programs. A deterministic function in Curry can be executed without the logic component of the language, avoiding (most of) the overhead of non-determinism.

This approach is still work in progress, but we have a prototype implementation of the type system available for the frontend of most Curry compilers available at `https://git.ps.informatik.uni-kiel.de/curry/curry-frontend/-/tree/det_sigs`.

2 Determinism Types

In order to simplify the presentation and reasoning about the type system, we restrict ourselves to a simple functional language with non-determinism. Its syntax for expressions and determinism types is given in fig. 1. Our language allows for determinism polymorphism and determinism function types in order to elegantly type higher-order functions.

The meaning of monomorph determinism types is as follows. The order of this listing is also the ordering of the corresponding abstract values, e.g., $Det \sqsubseteq Any$ and $Det \sqcup Any = Any$.

Det**:** The expression evaluates (w.r.t. standard Curry semantics) in a deterministic manner, i.e., without evaluation of choices.

$Any \to Det$**:** If a function of this type is applied to some argument, its evaluation is deterministic, either because the argument is not used or all non-determinism is encapsulated.

$Det \to Det$**:** If a function of this type is applied to a Det argument, its evaluation is deterministic, otherwise the evaluation is arbitrary (thus, it includes type $Any \to Any$).

$$
\begin{array}{lcll}
e & ::= & x & \text{(variable } x \in X\text{)} \\
 & | & C & \text{(constructor)} \\
 & | & e_1\ e_2 & \text{(application)} \\
 & | & \lambda x \to e & \text{(abstraction)} \\
 & | & e_1\ ?\ e_2 & \text{(choice)} \\
 & | & case\ e\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(case expression)} \\
p & ::= & c\ x_1 \ldots x_n & \text{(pattern)} \\
TVar & ::= & \alpha & \text{(type variables)} \\
\Delta & ::= & TVar \mid Det \mid Any \mid \Delta \to \Delta & \text{(determinism types)}
\end{array}
$$

Figure 1: Basic functional logic language with determinism types

$Det \to Any$**:** If a function of this type is applied to some argument, its evaluation is arbitrary.

Any**:** The expression evaluates (w.r.t. standard Curry semantics) in a possibly nondeterministic manner, i.e., with evaluation of choices. Importantly, choices between functions (e.g. `id ? not` are also of this type. Thus, the result of evaluating such an expression can be anything: deterministic, nondeterministic or a function.

Note that the type $Any \to Any$ is formally allowed but its usage is similar to $Det \to Any$.

These are only the descriptions for monomorph types. Polymorphic types will be instantiated at a functions call site. A type like $\alpha \to \alpha$ can be instantiated to both $Det \to Det$ and $Any \to Any$, which makes it equivalent in its semantics to $Det \to Det$.

To make it possible to supply a deterministic argument to a function of type $Any \to Det$, we use the sub-typing relation $\sqsubseteq$ on determinism types defined in fig. 2.

Some examples for determinism types are as follows:

```
-- simple functions
not :? Det → Det
id :? α → α
-- higher-order and utility functions
const :? α → β → α
flip :? (α → β → γ) → β → α → γ
map :? (α → α) → α → α
-- encapsulation of non-determinism
allValues :? Any → Det
-- set function of nondet. function f
fS :? Det → Det → Det
--
(?) :? α → β → Any
```

Due to the sub-typing, the type for `not` essentially covers both types $Det \to Det$ and $Any \to Any$ in order to get the following types from the type of `map` given above:

```
map not [True] : Det
map not [True ? False] : Any
```

3 Rules for Determinism Typing

To determine the correctness of a determinism type, we use a set of inference rules similar to standard type correctness.

A *determinism type environment* associates determinism types to names:

$$\Gamma : X \to \Delta$$

The system of inference rules is intended to derive statements of the form

$$\Gamma \vdash e : \delta$$

stating that the expression e has determinism type δ w.r.t. the determinism type environment Γ.

In the inference rules, we use instances of polymorphic types: if δ is a determinism type and σ is a substitution of type variables by determinism types, then $\sigma(\delta)$ is a *type instance* of δ. Our rules are given in fig. 3.

While most rules are relatively straightforward, some remarks are in order:

- Rule AppFunD is required to type functions passed as pattern variables, e.g.,

```
case x of Nothing  → False
          Just f   → f True
```

- There is no case rule for discriminating arguments of polymorphic types (all types are instantiated).

$$\delta_1 \sqsubseteq \delta_2 := \begin{cases} true & \text{if } \delta_1 = Det \vee \delta_2 = Any \\ \delta_{11} \sqsupseteq \delta_{21} \wedge \delta_{12} \sqsubseteq \delta_{22} & \text{if } \delta_1 = \delta_{11} \to \delta_{12} \wedge \delta_2 = \delta_{21} \to \delta_{22} \\ false & \text{otherwise} \end{cases}$$

$$\delta_1 \sqsupseteq \delta_2 := \delta_2 \sqsubseteq \delta_1$$

Figure 2: Ordering of determinism types

- The case rule for functional types is required for the completeness of the inference system. However, in a real program a case on a function is incorrectly typed. Thus, this cannot happen.
- The sub-typing relation is used in the rule AppFunArrow to allow supplying a deterministic argument to a function of type $Any \to \alpha$.

4 Applications and the Prototype in Curry

As stated in section 1, our main application of the type system is the safe execution of Curry programs containing I/O operation. Since non-deterministic I/O operations cannot be applied (one cannot "duplicate the world"), a non-deterministic I/O operation causes a crash of the program when such operations occur at run time. Thus, if the main operation of a program to be executed has determinism type Det, e.g.,

```
main : Det
```

it can be safely executed. Or in a REPL, expressions of type `IO` should be executed only if they have the determinism type Det.

By allowing the user to annotate functions with determinism types in the source code, we can detect unsafe or unexpected uses of non-determinism in more places as well. For example, our prototype compiler implementation would reject the following program because the function is not actually deterministic.

```
deterministicList :? Det
deterministicList :: [Int]
deterministicList = [1, 2, 3 ? 4]
```

For the syntax of determinism signatures/annotations, we chose `:?` since it is a combination of the symbol for the type signatures `::` and the symbol for the non-deterministic choice operator `(?)`. Giving a determinism annotation is optional. If none is present, it will be inferred by the compiler.

Additionally, with our prototype it is possible to restrict functions of a type class to a certain determinism type. For example, the comparison operator of the `Data` type class [3], or the arithmetic functions in `Num` can be restricted to be deterministic.

```
class Data a where
  (===) :? Det  → Det  → Det
  (===) :: a  → a  → Bool

  aValue :? Any
  aValue :: a
```

These determinism annotations will be checked for each instance of the type class by the compiler. Naturally, the function implemented by a concrete instance is allowed to be more specific than the determinism type that is annotated in the type class.

Determinism signatures are optional in type class definitions as well. In contrast to normal functions however, there is no way to infer the determinism type of a type class function because we do not know all possible instances of that class. One of the consequences is that functions that are constrained by a type class might have unexpended determinism types, unless the type class contains a determinism signature. In the following example, the function `print` is not known to be deterministic, because we have not added determinism signatures to the `Show` type class.

```
class Show a where
        -- Not in the class definition:
        -- show :? a  → a
        show :: a  → String

putStrLn :: String  → IO ()
putStrLn = <...>

print :? Any
print :: Show a => a  → IO ()
print x = putStrLn (show x)
```

Var	$\Gamma \vdash x : \delta$	if $x \in Dom(\Gamma)$ and δ is a type instance of $\Gamma(x)$
Free	$\Gamma \vdash x : Any$	if $x \notin Dom(\Gamma)$
Cons	$\Gamma \vdash C : Det \to \cdots \to Det \to Det$	if C is an n-ary constructor
AppFunND	$\dfrac{\Gamma \vdash e_1 : Any \quad \Gamma \vdash e_2 : \delta}{\Gamma \vdash e_1\ e_2 : Any}$	
AppFunD	$\dfrac{\Gamma \vdash e_1 : Det \quad \Gamma \vdash e_2 : \delta}{\Gamma \vdash e_1\ e_2 : \delta}$	
AppFunArrow	$\dfrac{\Gamma \vdash e_1 : \delta_{11} \to \delta_{12} \quad \Gamma \vdash e_2 : \delta_2}{\Gamma \vdash e_1\ e_2 : \delta_3}$	$\delta_3 = \begin{cases} \delta_{12} & \text{if } \delta_2 \sqsubseteq \delta_{11} \\ Any & \text{otherwise} \end{cases}$
Abs	$\dfrac{\Gamma[x \mapsto \delta_1] \vdash e : \delta_2}{\Gamma \vdash \lambda x \to e : \delta_1 \to \delta_2}$	
Choice	$\dfrac{\Gamma \vdash e_1 : \delta_1 \quad \Gamma \vdash e_2 : \delta_2}{\Gamma \vdash e_1 \texttt{?} e_2 : Any}$	
CaseND	$\dfrac{\Gamma \vdash e : \delta_1 \to \delta_2}{\Gamma : case\ e\ of\ \{\overline{p_k \to e_k}\} : Any}$	
CaseFun	$\dfrac{\Gamma \vdash e : Any}{\Gamma : case\ e\ of\ \{\overline{p_k \to e_k}\} : Any}$	
CaseD	$\dfrac{\Gamma \vdash e : Det \quad \Gamma[\overline{x_{n_i} \mapsto Det}] \vdash : e_i : \delta_i\ (i = 1, \ldots, k)}{\Gamma \vdash case\ x\ of\ \{\overline{p_k \to e_k}\} : \delta_1 \sqcup \ldots \sqcup \delta_k}$	where $p_i = c\ \overline{x_{n_i}}$

Figure 3: Typing rules for determinism typing

5 Relation to Operational Semantics

In the future, we want to prove the following theorem:

Theorem (Preservation)**.** *If $\Gamma \vdash e : \delta$ and $e \Rightarrow e'$, then $\Gamma \vdash e' : \psi$ with $\psi \sqsubseteq \delta$, where $\Rightarrow$ denotes a single step in a small-step semantics.*

In other words: an evaluation step can only refine the determinism type of an expression to a more specific type. Thus, if at any point during a multi-step evaluation of an expression, it is evaluated to some kind of choice, the original expression must have had type Any, since a choice between two expressions is always of type Any.

However, we already noticed a problem with this theorem when we tried to prove it using the Coq proof assistant [1]. Notably, it only holds for expressions that are well-typed with respect to a "normal" type system. Adding such a type system and relating it to the determinism type system makes the proof more complicated. It is not finished yet, but the remaining problems should be solvable.

Another important characteristic of our determinism type system is that every valid expression should have a determinism type that can be proven correct using the inference rules. This, too, requires a "normal" type system so that we only need to consider well-typed expressions. This proof is not finished as well, but we are confident that it can be completed.

6 Type Inference

We have implemented a type inference algorithm for determinism types in the curry compiler. The implementation is still work in progress, but at the moment it works by traversing each function to generate a set of constraints on the determinism

types of the function's arguments and its return type. These constraints are then solved using a special constraint solver for determinism types. The solver is complicated by the fact that we have to deal with the sub-typing relation ($\sqsubseteq$) and thus cannot simply use unification. The fact that Curry also includes type classes [7] in a Hindley-Milner type system [2, 5] makes type inference even more complicated since we do not always know the concrete implementation, and thus determinism type, of a function.

7 Conclusion

We have presented a type system for determinism types in Curry. The main application of the type system is detecting unsafe uses of non-determinism in I/O operations. At the moment this is still work in progress, but our prototype implementation already shows promising results.

References

[1] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642058809.

[2] Luis Damas and Robin Milner. "Principal Type-Schemes for Functional Programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212.

[3] M. Hanus and F. Teegen. "Adding Data to Curry". In: *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*. Springer LNCS 12057, 2020, pp. 230–246.

[4] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-lang.org`. 2016.

[5] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60.

[6] Friedrich Steimann. "A Simply Numbered Lambda Calculus". In: *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 24:1–24:12.

[7] P. Wadler and S. Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76.

Time-Travel Debugging with Visualization of Data-Structures Based on Instrumentation

Tony Listner, Kim Mönch & Mark Minas
University of the Bundeswehr Munich
Institute for Software Technology
Department of Computer Science
Neubiberg, Germany
{tony.listner, kim.moench, mark.minas}@unibw.de

Abstract

This extended abstract presents a concept of a debugging approach for Java programs that is based on instrumented source code instead of the usual Java Platform Debugger Architecture. Source code instrumentation informs the debugger of all state changes of the target program. By keeping a history of these state changes, time travel debugging is made possible. This approach particularly supports application and domain-specific, but also user-specific visualizations of the target program state, which makes this approach particularly suited for programming education.

1 Introduction

Despite the large number of different debugging tools being available, very few CS students use them in programming courses. Instead, they try to understand and correct their code by adding countless print statements. We identified several reasons for these observations: (1) Novice programmers consider standard debugging tools too complicated, in particular how they show the current program state, (2) it is hard for them to trace a problem back to its origin, and (3) it is not yet possible to provide them with a visualization of data structures that match the representation used in class. We address these issues by BUGVIS, an alternative debugging approach. We describe its concept and basic ideas in the following because BUGVIS has not been completely realized yet.

BUGVIS will provide the usual debugging capabilities for Java. But it will also support visualizations that are custom-made by the teacher for specific programming assignments, which will address issues (1) and (3). And in order to address issue (2), BUGVIS will support time-travel debugging, i.e., users can reverse the execution order and revisit earlier program states. Of course, there are tools available which meet one or the other requirement, e.g., JELIOT 3, JGRASP and OMNICODE (see Section 4), but to the best of our knowledge there is no solution that fulfills all of them.

The usual approach for realizing a debugging tool for Java is using the Java Platform Debugging Architecture. JPDA provides access to objects and their attributes at runtime. When visualizing data structures based on this information, one is more or less restricted to drawing boxes and arrows (except in tools like JGRASP; see Section 4), but it is difficult to show, e.g., a binary tree with a nice layout if the student does not realize her classes in the expected way.

BUGVIS does not use JPDA, but uses a different approach: The original source code of the target program is instrumented, producing instrumented source code and, after compilation, instrumented bytecode (called *instrumented target*). When executed, the instrumented target informs the debugging front-end of all state changes of the target (changes of variables and objects, method calls, etc.). The debugging front-end records

these state changes. Time-travel debugging is supported by rewinding these changes. Custom-made visualizations can be realized by visualization components, e.g., provided by the teacher in a programming assignment, and applied to all objects that provide a certain interface. Students are entirely free how they implement their data structures as long as they implement the required interface (see Section 2 for an example). The visualization component can immediately call these interface methods in the debugging front-end. This is made easy by code instrumentation, but rather difficult for JPDA-based debuggers.

Section 2 describes an example how BUGVIS may be used in a programming assignment. Section 3 describes the BUGVIS approach in more detail and discusses problems as well as open points, and Section 4 discusses related work. Section 5 concludes the paper.

This paper is an extended version of a paper presented at VL/HCC '22 Graduate Consortium [5].

2 Example

Consider the following scenario in a programming course with an assignment on binary trees. The teacher provides a Java interface `TreeNode` that requires the usual `getLeft`, `getRight` and `getInfo` methods for retrieving left and right child as well as node information. The assignment requires the students to implement binary tree nodes by implementing this interface, but they are completely free in their implementation otherwise. The teacher also provides the students with a BUGVIS instance that contains a visualization component for drawing binary trees with a nice layout as long as tree nodes implement the `TreeNode` interface. This visualization component can also draw the student's data structure if child references go wrong, e.g., producing cycles, which makes these errors quite obvious.

The debugging front-end of BUGVIS (and each of its instances) will also provide the usual debugging capabilities. In an information area, e.g., the user can see the currently visible objects, attributes, variables and constants known at the time of execution and check for correctness too. Together with further information about the stack trace and stack frames, this area provides everything that is needed for code analysis. Of course,

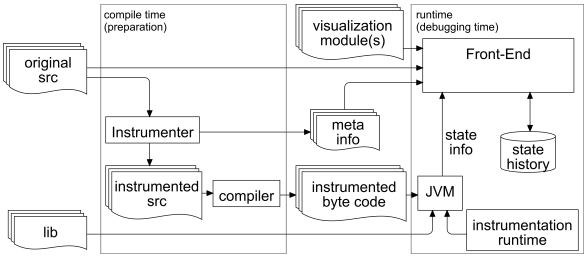


Figure 1: Architecture of BUGVIS.

the original source code is displayed and the current line, which corresponds to the associated visualized state, is also highlighted.

A student then works on the assignment. Because a BUGVIS instance is provided with the assignment, it is more likely that she will make use of this debugging tool, and she proceeds as follows: After opening her original source code, the code will already have been instrumented and compiled. Now she can select the variables which are to be visualized, in the given example it might be the variable containing the tree root. Afterwards, she can press the "Play" button and watch the program step by step while visually observing the continuously displayed object during runtime. The tree structure is visualized according to her own implementation. Once identifying an error, e.g., observing a cycle in the visualization, the student can "Pause" the visualization and trace back to the cause of this error. Using control elements, going back can not only be done step by step (which can be very click and time consuming under certain circumstances), but it is possible to use a time slider or a log for selecting specific moments to which the user can jump back. The usual known execution of the program up to a defined breakpoint should of course also be possible.

3 The BUGVIS Approach

Figure 1 shows the architecture of our BUGVIS approach, which is described below. The system consists of the three active components *instrumenter*, *instrumentation runtime* and *front-end*, which are used in the preparation and the debugging phases. In the preparation phase the instrumenter converts the original source code of the target into instrumented source code, which is compiled by the Java compiler into instrumented bytecode, the instrumented target. In the debugging phase the JVM (Java Virtual Machine) executes the instrumented target and the instrumen-

tation runtime, which create a stream of state information to the debugger front-end. The front-end saves the state information into the state history and uses them as well as the original source code and if available visualization modules to present and visualize one state at a time of the executed instrumented target.

In more detail, BUGVIS loads the original source code of the target and uses the instrumenter to convert it into instrumented source code in the **preparation phase**. The instrumented source code is one of two products of the instrumenter. The instrumenter is based on the established JAVAPARSER project [8]. With this tool it is capable of adding method calls to the original source code, which call methods of the instrumentation runtime, before and/or after the execution of every (possibly) state changing statement. Furthermore, it ensures that the instrumented target behaves in exactly the same way as the target (except informing the debugger) and no unwanted side effects are caused, for example by calling methods in different order or quantity than in the original source code. Afterwards, the instrumented source code is compiled by the compiler into the instrumented target.

The second product of the instrumenter is meta information about the original source code. While instrumenting the code, the instrumenter extracts further information, e.g. possible lines for breakpoints, and makes them available to the debugger.

In the **debugging phase**, the instrumented target and the instrumentation runtime are executed within the JVM. The instrumentation runtime is a Java library which offers necessary predefined methods to generate and send state information objects to the debugging front-end and holds a mapping of file ids to file names to reduce the amount of data of state information. These methods are called by the statements which were added to the instrumented target by the instrumenter.

The stream of generated state information is stored in the state history log. The debugging front-end allows the user to move through the state history and visualize the selected state in all debugging views. The state history is presented as a structured log to the user, which can be searched, to enable the user to select arbitrary states and visualize them. Within a state of the target none of the displayable information in any of the views of the debugger front-end changes. The other way around, every change of a displayable information is a state change. Therefore the instrumented target has to send state information about every change of state (exceptions are discussed later).

To fulfill the desired functionality and enable the user to see the status of the target, the debugger has to show all relevant state information to the user. This is the task of the debugging front-end. It is the component with which the user interacts to control and display the entire debugging process. The currently selected state of the target is presented at least in a stack-trace, the variables and objects view and the source code view. The source code view shows the original source code and the active statement and allows setting breakpoints.

In addition to these standard display elements for a debugger, the front-end also contains the visualization area. In the visualization area, single or multiple objects selected by the user and their relationship to each other can be displayed. Besides a default representation for arbitrary classes, BUGVIS contains visualization modules for the collection classes and Maps of the JCL (Java Class Library), e.g., `ArrayList`, `LinkedList`, `TreeSet`, `TreeMap`, etc.

In order to support the user in debugging as best as possible, she can create her own visualization modules and use them to replace the default visualizations or to visualize new or her own data structures (compare Section 2).

3.1 Instrumentation details

Listing 1 shows an instrumentation example. Lines 1 & 2 contain some *original* source code written by a student for an assignment (see Section 2). This original source code is fed into the instrumenter that produces the *instrumented* source code shown in Lines 3-19.

The instrumented code must inform the instrumentation runtime of every state change of the debugging target by adding calls to the instrumentation runtime of the form `Debug.*`. Every statement of the original source code may trigger many state changes. Consequently, the instrumenter must split them into separate statements. For instance, Line 1 of the original source code is translated into Lines 3-11 of the instrumented code. In order to refer back to the original source code, each call to the instrumentation runtime always contains at

Listing 1: Original/instr. source code (top/bottom)

```
1  MyTreeNode n1 = new MyTreeNode(5);
2  boolean success = n1.add(1);
3  ----------------------------------------------------------
4  Debug.decl(1,5,9,5,21,"n1","ex.MyTreeNode");
5  final var _var1 = 5;
6  Debug.callConstr(1,5,25,5,42,"ex",
7      "MyTreeNode","MyTreeNode(int)",_var1);
8  final var _var2 = new MyTreeNode(_var1);
9  Debug.returnedConstr(1,5,25,5,42,_var2);
10 Debug.assign(1,5,9,5,42,"n1",_var2);
11 MyTreeNode n1 = _var2;
12 Debug.decl(1,6,9,6,23,"success","boolean");
13 final var _var3 = 1;
14 Debug.callInstanceMethod(1,6,27,6,36,"ex",
15     "MyTreeNode","add(int)",n1,_var3);
16 final var _var4 = n1.add(_var3);
17 Debug.returnedValue(1,6,27,6,36,_var4);
18 Debug.assign(1,6,9,6,36,"success",_var4);
19 boolean success = _var4;
```

least the file id (first parameter) and the position, as line and column numbers of the start and end (second to fifth parameter) of the original statement.

For every declaration in the original source code, the instrumented code contains a `Debug.decl` call with the name and the type of the variable (Lines 4 & 12). In case of assignments in the original source code, `Debug.assign` is called with the name of the variable and its new value (Lines 10 & 18). When the original source code calls a constructor or a method, the instrumented code calls `Debug.callConstr` or `Debug.callInstanceMethod` with the necessary information before actually calling the constructor or method (Lines 6 & 14), and the return value is transmitted afterwards (`Debug.returnedValue`, Lines 9 & 17).

3.2 Discussion

There are three issues that need to be discussed and their solutions will affect each other. The first one is the specific way of execution of the target and the possible impact on the memory usage. The second one is the performance of the target and the debugger and the third topic is not instrumented code.

There are two different ways for the execution of the target. The asynchronous execution runs until it needs input or generates output. For a simple target without any in-/output, this means that the application runs from start to end, generates the appropriate state information objects, sends them to the debugger and terminates. If there are breakpoints set, the visualization stops at the first breakpoint, but not the execution. On the other hand, the synchronous one stops at every breakpoint and the user has to unpause the execution manually. Without set breakpoints asynchronous and synchronous execution are equal to each other.

Both ways of execution may lead to memory problems, especially considering that endless loops are not uncommon and lead to infinitely many state information objects. Therefore the debugger must be able to cope with a nearly endless stream of state information. Possible solutions for this problem are (1) deleting the oldest state information, when necessary, (2) writing the state info to a file, (3) stopping the execution or (4) pausing the execution and prompt the user to select the appropriate choice.

Solution (1) is the most simple one, but the deleted information is lost and debugging in this part of the state history is impossible. (2) is a valid solution, if the available main memory is limited and the execution is nearly terminated. In the endless loop case, this is not the best way, because the state information ultimately fills the hard drive. Another problem with (2) is the speed limiting factor of the hard drive, which is significant slower than the main memory. (3) is another basic way but with the consequence of not being able to debug applications after they generated to much state information. The best and most situation related solution is (4) which requires the implementation of some or all of the other solutions.

A possible tweak could be pausing the execution of the target after a specified amount of state information after the next breakpoint. The first breakpoint is usually set where the user is looking for the error. The execution can be unpaused again when the user reaches the current state in the history or unpauses manually.

Another performance issue results from the added method calls, because as shown in the example from Listing 1, eight method calls and four declarations with initialization have been added here to two declarations with initialization and method calls each. Therefore, the final impact on the execution speed has to be evaluated.

One way to reduce the impact on performance would be to not instrument the entire original

source code, but only the parts that seem important or relevant to the current task. BUGVIS does not instrument the JCL and external dependencies, because the complete source code would have to be available, which is usually not the case. Parts of the original source code could be treated like external dependencies.

When the target is executed and comes across a method call to not instrumented code, information about the call, the parameters and possibly the return value is still obtained (see Listing 1). Only information about the execution of the not instrumented code is missing, but this was classified as irrelevant anyway or belongs to a dependency and must be considered correct.

But not instrumented code still leads to a problem. The debugger will not be able to recognize state changes in objects created by not instrumented code as shown in the following short example. The target application creates an `java.util.ArrayList` instance, fills it with integers and calls `java.util.Collections::sort(List)` with the `ArrayList` instance to sort the content. The problem is that in the status information it says that an `ArrayList` object exists and that it has been filled with integers, but there is nothing in the status information about the changes made by the `sort` method of `Collections`.

One way to mitigate this problem is to check the attributes of every object after it was passed to not instrumented code and synchronize the front-end with the correct values. But this solution needs a filtering mechanism to detect calls to not instrumented code, otherwise it would be done after every method call. Another way is to encapsulate the object in an instrumentation container. For the example above, this means the `ArrayList` instance gets encapsulated in a container, which has all methods and types of the encapsulated object and forwards all calls to it. But the methods of the container are instrumented and call the instrumentation runtime whenever a method is called. This way the front-end gets state information even if the `ArrayList` instance is modified by not instrumented code.

It has to be evaluated which ways are the best concerning performance and state density.

4 Related Work

In the last decades, a bunch of tools for visual programming and debugging has been developed, most of them dedicated to the animation and visualization of algorithms and data structures [7]. In [9], an overview of about 40 different tools which are more or less still maintained is presented. The underlying motivation to this area is to appeal the potential of the human visual system. Since the dynamics of computer programs can be difficult to grasp when presented in textual format, it is expected that animated graphical formats can contribute to a better understanding [2].

Most closely related approaches that our work is based on are JELIOT 3 [6], JGRASP [1] and OMNICODE [4]. All of them provide a visual animated debugger while using the information provided on the heap during runtime and partially meet our requirements.

Therefore, it is especially in JELIOT 3 or JGRASP not possible to undo a step of execution or visualize backwards in any way. However, our approach differs significantly from JGRASP and OMNICODE in its target audience. Whereas JGRASP focuses on complex structures such as linked lists, trees and the data structure identifier as well as OMNICODE focuses on scatterplot matrix visualization which can be less than intuitive, but offers the possibility of tracing the change of a variable backwards. We focus (at the moment) on simple concepts like assignments, arrays, lists and trees presented by intuitive visualizations coupled with the functionality of easy moving forward and backward in a program execution while debugging.

JELIOT 3 is a Program Visualization application, targeting novice programmers. It visualizes how a Java program is interpreted while displaying animated method calls, variables and operations, allowing the student to follow step by step the execution of a program. Unfortunately JELIOT 3 isn't maintained since 2014 and there is neither typical IDE support while writing the code, such as import organization, nor compiler error handling before the compiler runs manually.

JGRASP is a lightweight development environment to provide automatic generation of software visualizations. Along with an integrated debugger the object viewers automatically generate dynamic, state-based visualizations of objects and primitive variables in Java. Multiple synchronized visualizations of an object, including complex data

structures, are available to users. The unique selling point of JGRASP is surly the data structure identifier mechanism, which recognizes objects that represent traditional data structures such as stacks, queues, linked lists, binary trees, and hash tables, and then displays them in a textbook-like presentation view.

The live IDE called OMNICODE ("Omniscient Code") continually runs the user's Python code and sends the code and test cases to run on a server. The server uses a modified version of the ONLINE PYTHON TUTOR [3] backend to execute Python code and generate a full runtime value trace. The visualization is done by a scatterplot matrix to visualize the entire history of all of its numerical values both relative to all execution steps as well as relative to all other values. To filter the visualizations and hone in on specific points of interest, the user can brush and link over the scatterplots or select portions of code. They can also zoom in to view detailed stack and heap visualizations at each execution step.

5 Conclusion

This extended abstract outlines an approach to time travel debugging with visualization of data-structures based on instrumentation. We expect that BUGVIS will provide novice programmers with a debugging tool that allows easy time travel while analyzing the behavior of their code and, most importantly, is easier to use than usual debuggers. The possibility of individual visualization of given data structures should further enable the teacher to point out problems in the implementation. After the completion of the program, its benefits and added value when learning the Java programming language will be ascertained and discussed on the basis of a user study.

Author Profiles

Mark Minas is a Full Professor at the Universität der Bundeswehr München and head of the Programming Systems Group. His focus of research includes visual, graph-based and domain-specific languages as well as programming support systems.

Tony Listner and **Kim Mönch** are PhD students at the Universität der Bundeswehr München in the Programming Systems Group. Their teaching activities range from software engineering to visual languages and environments to computational geometry.

References

[1] James H. Cross et al. "Dynamic Object Viewers for Data Structures". In: *Proc. of the 38th SIGCSE Technical Symp. on Computer Science Education*. SIGCSE '07. Covington, Kentucky, USA: ACM, 2007, pp. 4–8.

[2] Thomas Green and Marian Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages & Computing* 7 (1996), pp. 131–174.

[3] Philip Guo. "Online Python Tutor: Embeddable Web-based Program Visualization for CS Education". In: *Proc. of the 44th ACM Technical Symp. on Computer Science Education*. SIGCSE 2013. New York, NY, USA: ACM, 2013, pp. 579–584.

[4] Hyeonsu Kang and Philip J. Guo. "Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations". In: *Proc. of the 30th Annual ACM Symp. on User Interface Software and Technology*. UIST '17. Québec City, QC, Canada: ACM, 2017, pp. 737–745.

[5] Kim Mönch. "Time-Travel Debugging with Visualization of Data-Structures Based on Instrumentation". In: *2022 IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*. Rome, Italy, Sept. 2022, pp. 1–3.

[6] Andrés Moreno et al. "Visualizing Programs with Jeliot 3". In: *Proc. of the Working Conf. on Advanced Visual Interfaces*. AVI '04. Gallipoli, Italy: ACM, 2004, pp. 373–376.

[7] Thomas Naps et al. "Exploring the Role of Visualization and Engagement in Computer Science Education". In: *SIGCSE Bulletin* 35 (2003), pp. 131–152.

[8] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *JavaParser: Visited*. Leanpub, 2019.

[9] Juha Sorva. "Visual Program Simulation in Introductory Programming Education". English. PhD thesis. Aalto University publication series, 2012, p. 428. URL: http://urn.fi/URN:ISBN:978-952-60-4626-6.

Poor Man's Compilers
How Forth treats its source code

Klaus Schleisiek
kschleisiek@freenet.de

Abstract

The way Forth processes its source code is peculiar and unfortunately quite unknown. The compilation in particular is unusual. It is extremely simple and allows the syntax to be expanded on the fly. For specific application situations, specific syntactic extensions can be defined as "syntactic sugar", which make the source code easier to read and therefore more reliable.

1 How Forth came about and where the name comes from

Forth was developed by Charles "Chuck" Moore, who had worked as a freelance programmer since the late 1950's. Primarily in automation projects with real-time requirements. First on mainframes, since the mid-1960s on minicomputers, which at that time often only offered an assembler as software support.
Over a 10-year period, he experimented with different programming concepts to increase his productivity as a programmer. Finally, in 1968, he realized that he had developed a complete programming language, which he wanted to call "Fourth" by the time the third generation of computers was talked about. But the assembler he used only allowed five-letter identifiers. So it became "Forth".
Chuck Moore: "I developed Forth over a period of some years as an interface between me and the computers I programmed. The traditional languages were not providing the power, ease, or flexibility that I wanted."

2 The Forth System

The language is based on a virtual machine[1] as a programming model, with Forth as the "Assembler" of this machine.

The Forth machine has no registers, instead it has two stacks. The data stack serves the calculation of results. The return stack for storing return addresses of function calls and within a function, which is called "word" in Forth, additionally as loop counter and temporary memory. When a function is called, the input parameters are on the data stack. At the end of the call, these are replaced by the function results. The ALU inputs are connected with TopOfStack (TOS) and NextOfStack (NOS), the ALU output leads back to TOS. Words for stack manipulation allow to control the sequence of parameters on the data stack in order to modify (**`swap`**), or duplicate (**`dup`**), or destroy (**`drop`**) parameters. For arithmetic expressions, Forth uses reverse Polish notation[2], which needs getting used to. Because of the stacks, Forth is a null-address machine, and functional style programs can be written.

Two stacks are required because the dynamics of the two stacks are different. A word such as **`+`** consumes two input values and leaves one result on the data stack. When a function is called, the return address is placed on the return stack and must be there again as top element for the **`EXIT`** at the end of the function[3]. The PC is used to address the sequential program code in memory.

[1] Due to its simplicity, it can be efficiently implemented as a "real machine".
See https://github.com/microCore-VHDL
[2] Older engineers still fondly remember HP's calculators.

[3] The runtime environments of most programming languages have only one stack. This complicates the handling of return addresses and data and led to the invention of "stack frames", which, however, significantly increase the cost of a function call at runtime..

The dictionary is a tree structure of word lists that contain the commands of the Forth machine. Humans and other systems communicate with the Forth machine via ASCII encoded Character IO.

3 The Dictionary

A central element of Forth is the symbol table, which is called dictionary and is still available after compiled as a principle. That's why Forth code can be executed interactively over a Character-IO interface.
The dictionary is a tree structure of word lists, the so-called vocabularies. The root of the dictionary is the `Forth` vocabulary that contains the basic words of a Forth system[4]. After the lexical analysis, the `Context` is searched, a dynamically configurable list of vocabularies.
New word definitions are entered at the top of the `Current` word list. That's why it is possible to redefine already existing entries under the same name, e.g. to add debugging features.
Because of the vocabulary structure, identical word names can appear in different vocabularies that have different semantics. Which semantics apply at any point in time is well defined by `Context`.
In addition to the word name, each word list entry contains a reference to executable code that defines the semantics of the word.

4 Parsing in Forth

The lexical analysis is very simple: each character string surrounded by spaces (blank, tab, cr etc.) is a token. That's basically how we read text[5]. Not only is this easy, but it also has a collateral benefit: names in Forth may contain any ASCII character except spaces[6].
In the next step, the parser searches the word lists of `Context` to see whether the token exists in the dictionary. If so, its associated code is executed. If not, it is checked whether the token is a number. If so, the number is pushed on the data stack.
If it is neither found in the dictionary, nor a number, then the error message `?` displayed.
This is how the Forth system works in interpretation state.
To debug a word, numbers are placed on the stack as input parameters (if the word has any

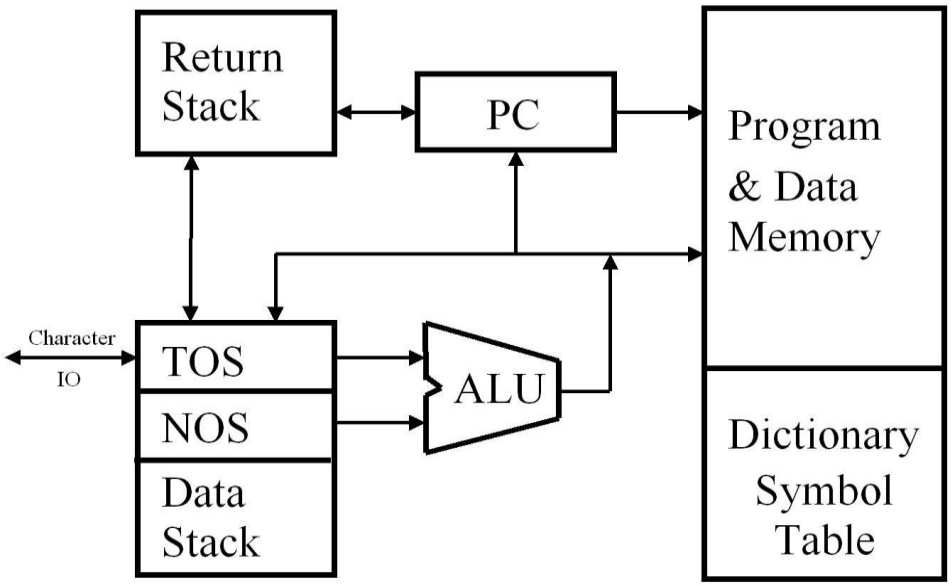


Fig 1: The Forth Maschine

[4] Forth is standardized as ISO/IEC 15145:1997. The standard is continuously developed, see: https://forth-standard.org.
[5] Why do most programming languages do it differently? Two reasons have occurred to me so far:
1. Fortran was geared toward mathematical formulas, and mathematicians traditionally omit the spaces around operators.
2. Columns on the punch cards were "expensive" and therefore the strategy serves a primitive data compression.
In any case, the result is that the "Fortran" method was copied without questioning and gave us RegEx. And the programmers have to live with annoying restrictions on naming.
[6] Therefore, special characters can be used in names to provide semantic hints. E.g. `"` for string functions, `.` for output on the display, `#` in the first position for constants.

input parameters at all), the word to be examined is executed by entering its name and then the contents of the stack are displayed, which should now contain the output parameters of the word at the top of the data stack.

5 Compilation in Forth

The system variable **State** determines whether the Forth system is in interpretation state (**State = 0**) or in compilation state (**State = 1**).

1.1 Colon und other Compilers

The following line is input to the Forth system:
: xlerb word1 word2 word3 ;
: creates a new entry in the **current** vocabulary named **xlerb** and sets **State** to 1, i.e. the system switches to compilation state. This is followed by the words **word1**, **word2** and, **word3**, which must already exist in the dictionary. They are now not executed but compiled as function calls. This sequence of executable code is associated with **xlerb** and represents its semantics. The **;** has the immediate bit set (see 1.2) and is therefore executed instead of being compiled as function call. It compiles an **EXIT** and resets **State** to 0. This completes the definition of **xlerb** and the system is back in interpretation state.
Besides **:** there are other words that create vocabulary entries but do not enter compilation state:
Variable <varname> reserves a memory cell and puts its absolute address on the data stack when **<varname>** is subsequently executed.
<number> Constant <constname> pushes **<number>** onto the data stack if **<constname>** is subsequently executed.
Vocabulary <vocname> creates the new vocabulary **<vocname>**.

1.2 The Immediate Bit

The immediate bit is a marker bit in the name of a word entry in the dictionary[7]. If set, then that word will also be executed in compile state instead of being compiled as a function call. So an immediate word is a small compiler. The **;** we have already met above.
The immediate bit now makes it possible to create control structures. An example:

```
: conditional ( flag -- ) IF word4
ENDIF word5 ;
```

: creates the word entry **conditional**.
The string in parentheses is a comment[8] and means that **conditional** has one input parameter **flag** and no output result.
This is followed by **IF**, whose immediate bit is set and is therefore executed. The **IF** compiles **0=branch** and reserves space for the branch target address, which is not known at this point. Therefore, a 0 is temporarily entered there during compilation. And its address is placed on the data stack. The compiler **IF** is already finished.
Word4 follows, which is compiled as a function call.
Then comes **ENDIF**, another compiler. The previously missing target address of the preceeding **IF** is now known and is entered after **0=branch** at the address that is on the stack, because **IF** put it there. This completes **ENDIF**.
The function call of **word5** follows and then **;** finishes the definition of **conditional**.
If there is a 0 on the stack when **conditional** is executed, then **0=branch** consumes this flag and jumps directly to **word5**.
If there is a number /= 0 on the stack, then **0=branch** also consumes this flag, but no branching occurs, and **word4** and **word5** are both executed.

In Forth there is hardly any predefined grammar and no compiler that checks the source code for compliance with grammar rules. Instead, there are individual small compilers that interact synergistically and thereby generate syntax. Additional compilers can be defined as part of an application. Forth is not only expandable in terms of its range of functions, but also in terms of its syntax.

Below is a striking example.

[7] The immediate bit is set in that the **;** of a colon definition is immediately followed by the word **immediate**.

[8] Because of reverse Polish notation, the brackets are not used for arithmetic expressions, so they can be used for comments.

6 Syntactic Sugar

ASCII text is to be output as Morse tones. To do this, a Morse code table must be created. And that's potentially error-prone.
Here is a "desired syntax" that came about after a few mental iterations between "easily realizable" and "well readable". This is followed by the code for the compilers[9] required for this.

```
morsetable:
   . _        | A
   _ .        | N
   _ . . .    | B
   _ _ _      | O
   _ . _ .    | C
   . _ _ .    | P
   _ . .      | D
   _ _ . _    | Q
   .          | E
   . _ .      | R
   . . _ .    | F
   . . .      | S
   and so forth ...
;morsetable
```

And this is what the compilers for the desired syntax look like:

```
$80 cells Constant #codes
\ size of the code table

Create Morsetable  #codes allot
Morsetable #codes erase

Vocabulary <morse>

: morsetable: ( -- 0 )
   also <morse>
\ make <morse> first vocab. in CONTEXT
   0
\ end marker for the first code
;

also <morse> definitions
\ set CURRENT to <morse>

1 Constant .
2 Constant _

: morsecode!  ( count bits <character>
-- )
   swap 8 lshift or
\ pack count and code into 16 bit
\ value
   char cells
   dup #codes < 0= abort" character
out of range"
   Morsetable + !
\ store at char's position
;
: | ( 0 n1 .. nr -- 0 )
   0 ( count )  0 ( morsebits )
   BEGIN  rot ?dup
   WHILE
      1- swap 2* or
\ add next morsebit
      swap 1+ swap
\ and increment count
   REPEAT   morsecode!
   0
\ end marker for the next code
;
: ;morsetable ( 0 -- )
   0 - abort" malformed morse table"
   previous
\ remove <morse> from CONTEXT
;
previous definitions
\ reset CURRENT to previous vocabulary
```

The vocabulary **<morse>** now contains the words **.**, **_**, **|**, **morsecode!** and **;morsetable**.

References

[1] C.H.Moore: "Forth: A new way to program a mini-computer", 1974, Astron. Astrophys. Suppl. 15, 497-511.

[2] Leo Brodie: "Starting Forth", 1981,https://www.forth.com/wp-content/uploads/2018/01/Starting-FORTH.pdf

[3] Leo Brodie: "Thinking Forth", 1984,https://www.forth.com/wp-content/uploads/2018/11/thinking-forth-color.pdf

[9] In the executable system it has to be exactly the other way round, because Forth traditionally compiles its source code in one pass and therefore words that are used must have been defined beforehand.

Pause,
the Janus-faced sister of the interrupt

Klaus Schleisiek
kschleisiek@freenet.de

Abstract

Interrupts have been known for a long time: An event happened that the software was NOT expecting.
For the first time in the transputer, there was another input signal to the processor, the pause: An event did NOT happen that the software was expecting.
In the transputer, the pause mechanism was used to establish serial communication channels between several transputers working in parallel and to synchronize them.
If a transputer wants to read data from a channel, this is of course only possible if it was actually sent by the remote station. If no data was received, the pause signal was activated, the transputer hardware called up the scheduler and took care of other tasks. At some point, data was (hopefully) received and the paused task could continue working.
The pause mechanism also ensured that no message could be passed to the sender, before the previous one had been completely transferred.
An unpleasant feature of the transputer was that the scheduler was hardwired into an internal ROM and therefore could not be adapted to a users operating system.
In microCore, a real-time processor for FPGAs realized in VHDL, the pause is implemented as a trap whose functionality can be freely programmed by the user. In my presentation I will show how the pause is realized in hardware. And that it is suitable for solving the problem of resource management (MUTEX) completely in hardware, so that these difficult to debug source for errors is completely eliminated.
For microCore see: https://github.com/microCore-VHDL

Generating Diverse Exercise Tasks on UML Class and Object Diagrams, Using Formalisations in Alloy

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

Abstract

Class diagrams are used to statically design object-oriented software. Object diagrams are a closely related concept. Understanding class diagrams and their connection to object diagrams is crucial for object-oriented software design. Hence, teaching of these concepts is an important part of many undergraduate computer science curricula. We present support for such teaching via automatically generated exercise tasks, of different types, to be used in an e-learning setting. We use the Alloy specification language and analyser as crucial part of the generation process, via modelling the relevant concepts and meta-concepts inside Alloy.
All presented task types are integrated into the e-learning system Autotool providing an interface to learners for receiving automatic feedback and most recently also automatic grading. This enables the use of these task types as support for individual learning as well as tasks of an (online) examination. These task types have been used as part of take home exams during recent semesters.
This work has previously been published in part by Siegburg and Voigtländer [1].

References

[1] Marcellus Siegburg and Janis Voigtländer. “Generating Diverse Exercise Tasks on UML Class and Object Diagrams, Using Formalisations in Alloy”. In: *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers co-located with Modellierung 2020, Vienna, Austria, February 19-21, 2020*. Ed. by Judith Michael et al. Vol. 2542. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 89–100. URL: https://ceur-ws.org/Vol-2542/MOHOL5.pdf.

A Set Semantics for Hehner's Bunch Theory

Marius Freitag
Fern Universität in Hagen
Lehrgebiet Programmiersysteme

Abstract

Bunch theory is a unique and simple formulation of multiples, which was conceptualized by Eric C. R. Hehner. The uniqueness of bunches stems from their *unpackaged* nature; elements are collected but the collection itself is not regarded as a *thing* in its own right. This allows bunches to generalize elements, which are equated with singular bunches. Through numerous publications [3–5] and a comprehensive textbook [2], the definition of bunch theory was extended to form a full theory of programming with bunches as a base.
Despite the seminal new way of defining data aggregations, Hehner's formalisms comprise some ambiguities and dormant contradictions. In my work, I have developed both a simple and a more complex, functional bunch language based on formal syntax and denotational semantics to create a more explicit semantic basis for bunch theory.
The characteristic property of these languages is that bunches are mapped to nested set structures as defined by set theory. Bunch theory functions are not mapped to regular set theory functions but to relations as part of a relational algebra. This form of mapping proves to be a natural choice due to the multi-valued properties of bunch theory functions.
Some assumptions about and changes to the original bunch theory are postulated to create consistent language systems. After their construction, the semantics systems are used to systematically map Hehner's axioms and prove that they still form tautologies in their set theory representations. This shows that the introduced languages are faithful representations of Hehner's concepts.
In this talk, I will discuss the main concepts of my work by motivating and introducing Hehner's bunch theory, giving an intuitive mapping to set theory, reviewing required assumptions, and hinting at a formalization. Further details and formalizations are elaborated in [1].

References

[1] Marius Freitag. *A Set Semantics for Hehner's Bunch Theory*. Master's Thesis. Supervised by Friedrich Steimann. FernUniversität in Hagen, Lehrgebiet Programmiersysteme, 2022.

[2] Eric C. R. Hehner. *A Practical Theory of Programming*. Sept. 30, 2021.

[3] Eric C. R. Hehner. "Bunch theory: A simple set theory for computer science". In: *Information Processing Letters* 12.1 (1981), pp. 26–30.

[4] Theodore S. Norvell and Eric C. R. Hehner. "Logical Specifications for Functional Programs". In: *Mathematics of Program Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 269–290.

[5] Richard F. Paige and Eric C. R. Hehner. "Bunches for Object-Oriented, Concurrent, and Real-Time Specification". In: *FM'99 – Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 530–550.

de Linguis Musicam Notare

Markus Lepper
semantics gGmbH, Berlin
post@mlepper.de

Baltasar Trancón y Widemann
Nordakademie Elmshorn

Zusammenfassung

Die Methode der *mathematischen Remodellierung* versucht, Symbolsysteme, ihre Grammatik und Semmantik, die sich im praktischen menschlichen Diskurs eines bestimmten Sachgebietes, oder in dessen theoretischer Reflexion, historisch entwickelt haben, mit mathematischen Methoden nachzubauen.

Ziel ist zuförderst, diese Praxen und Fachdiskussion durch Bereitstellung einer mit mathematischen Methoden, also viel exakter als bisher, definierten Nomenklatur auf eine klarere Grundlage zu stellen. Zwangsläufige Voraussetzung der Mathematisierung ist u.a., sämtliche allerorten anzutreffenden *unausgesprochene Vorannahmen* explizit zu machen, wodurch im zwischenmenschlichen Diskurs Missverständnisse vermieden werden können.

Weiteres Ziel ist, eine öffentliche und transparente Diskussion über die Grundlagen jeder Automatisierung zu beginnen.

Wichtige und oft fruchbare Handlungsmaxime dabei ist, dass nicht sämtliche historisch entwickelten Anti-Orthogonalitäten nachgebaut werden müssen: Die Erstellung des Modelles erfolgt inspiriert, aber nicht determiniert durch die historische Praxis. Es entsteht oftmals eine “konvexe Hülle” des vor-formalen Modelles. Historisch bedingte Einschränkungen und Nicht-Kompositionalitäten werden so als solche sichtbar und können, wenn für die Fachdiskussion erforderlich, als einschränkende Bedingungen hinzugefögt werden.

Fast immer stellte sich heraus, dass nicht ein einziges Modell, sondern eine Familie von Modellen gefunden wurde. Diese können u.a. durch Parametrisierung aus einander hervorgehen, was ein exakt definiertes Klassifikationsraster unterschiedlicher Sprachpraxen ergibt.

Forschungspolitisch wichtig ist, dass die Verfasser in keinem Falle ein bestimmtes dieser Modelle als adäquat, notwendig, historisch zutreffend, Erklärungen liefernd, etc., bezeichnen wollen, sondern lediglich den Fachexpertinnen einen Katalog von möglichen Modellbildungen zur Verfügung stellen, aus denen diese wählen können, entsprechend den strukturellen Anforderungen der von ihnen zu formulierenden Erkenntnissen. Damit dann auch eine exakte Benennung des von ihnen gewählten Modelles angeben können, und unter dessen Prämissen ihre Thesen deutlich exakter formulieren.

Die Verfasser haben dieses Verfahren bisher auf so unterschiedliche Bereiche wie das System der physikalischen Maßeinheiten, funktionalharmonische Musiktheorie (beides siehe Vorträge in diesem Band) und Strukturierung von Gesetzestexten [2] angewendet.

Bisher umfangreichste Anwendung ist die kürzlich vom Erstautor vorgelegte Dissertation [1], eine Formalisierung der gesamten “CWN” = “Common Western Notation” = modernen Musiknotation, soweit eher konventionell angewendet. Diese wurde hier vorgetragen.

Literatur

[1] Markus Lepper. “de Linguis Musicam Notare”. www.epos.uni-osnabrueck.de/buch.html?id=150[20231013]. PhD thesis. Osnabrück: Universität Osnabrück, 2021. ISBN: 978-3-94025-88-4.

[2] Markus Lepper and Baltasar Trancón. “Bezugnahme auf Bestandteile von Gesetzestexten als Problem der angewandten Rechts- und Informationswissenschaften”. In: *jurPC* (2019). http://dx.doi.org/10.7328/jurpcb20193412154.

Revision Control Revised: Some Contributions to the Algebraic Theory of Patches

Baltasar Trancón y Widemann
Nordakademie Elmshorn
baltasar@trancon.de

Markus Lepper
semantics gGmbH, Berlin

Abstract

Revision control manages changes and deviations of document repositories. Theoretical foundation on an algebra of patches has been investigated only partially. In this paper, the connection to the mathematical theory of inverse semigroups is made rigorous. The conceptual framework of repository states and patches is mapped to semigroup actions, words and representations. A reference model of patches, μ-ed, is defined for laboratory study, abstracting from technical detail while retaining interesting basic behavior. Some fundamental tools of semigroup theory are used to characterize the algebra of the reference model, namely the idempotent semilattice and the algorithmic solution of the word problem. These characterizations allow simple proofs of relevant behavioral properties. On the negative side, the objective construction of a parallel composition or merge operation for patches is demonstrated to be fundamentally problematic.

Revision (or *version*) *control systems* (RCSs), such as the eponymous `rcs` [12] or `git`, are indispensable tools of modern real-world software engineering. Their capabilities cover two largely orthogonal domains, namely a *database* organization for storing a software development project in so-called *repositories*, and the *tracking of change* of the content of a repository in the presence of sequentially and concurrently acting developers.

The present work is focused on the latter aspect, which has a peculiar status as a discipline of software tool-making. Its most basic operations are the extraction of the difference between documents (or versioned states of the same document object), the application of such a difference to one state of the document transitioning to the other, and the reconciliation of concurrent diverging changes. They are embodied operationally in the Unix tool archetypes `diff`, `patch` and `diff3/merge`, respectively, as well as in numerous similar sub-tools packaged in RCSs.

The scientific rigor of the state of the art in such tools is paradoxical. On the one hand, their operation is based on well-designed and understood algorithms founded in formal language theory. On the other hand, the actual use of those algorithms is governed by pure pragmatics. The maturity of tools is judged by time-tested heuristics and trustworthy best-effort behavior, rather than by deep understanding of semantics — an attitude that stands out in marked contrast to other software development tools such as *compilers*.

This paper is an attempt to pinpoint the appropriate theoretical framework for reasoning about the semantics of change tracking in document modification operations. The various aspects of general theoretical discussion are illustrated in turn with a running example model instance, carefully chosen to abstract away the messy details but not the interesting problems of the practice that we wish to model. This model is referred to as "μ-ed" (the *micro editor* model), and the pertaining definitions are decorated with the superscript μ.

The main contributions of the present work are:

- A clear terminology that specifies patch syntax and semantics in terms of a partial semigroup action;
- a reference model for changes to textual document content, which is simple enough for theoretical study but yet poses the relevant challenging problems;
- a characterization of the idempotent lattice of such changes;
- an automata-based solution to the word problem for the reference model;
- a clarification of the problems on the way to a general theory of parallel composition (merging) of patches.

1 Basic Definitions

Definition 1.1 (States)**.** *$\mathcal{S}$ denotes a fixed set of* states *with a distinguished* initial *element $s_0 \in \mathcal{S}$.*

Patches are syntactic expressions of intended changes. Thus their semantics should be state transition operations. We call the inhabitants of the semantic domain *transactions*, and specify their properties abstractly.

Requirement 1 (Transactions)**.** *Transactions shall be*

1. abstract – *defined independently of particular pre-/post-states;*
2. partial – *not necessarily applicable to all conceivable pre-states;*
3. invertible – *subject to roll-back without loss of information;*
4. associative – *subject to arbitrary sequential (de)composition.*

This set of axioms strongly suggests a natural encoding in the representation theory of inverse semigroups. [3]

Definition 1.2 (Inverse Semigroup)**.** *A* semigroup *is a set endowed with an associative binary operation, written multiplicatively. A semigroup is* inverse *iff every element x has a unique weak inverse:*

$$xx^{-1}x = x \qquad x^{-1}xx^{-1} = x^{-1}$$

Definition 1.3 (Symmetric Inverse Semigroup)**.** *The* symmetric inverse semigroup $\mathcal{I}_X$ *over a set X is the space of partial bijections on X [10, 13], equipped with the relational composition operation $\fatsemi$.*

The first goal is to define both patches and transactions in these terms. Fortunately, inverse semigroups share the associated syntactic notation of group elements by words, and thus give rise to a generalized word problem, with partial interpretations.

Definition 1.4 (Elementary/Composite Patches, Group Words)**.** *Consider a set $\mathcal{P}_0$ called* elementary patches*, assumed to be a formal language, and hence countable. This set generates a language of* group words*, that is a* free monoid-with-involution *$\mathcal{P}$, called (composite)* patches*.*

We write ε and $^{-1}$ for the empty group word and the involution, respectively.

Definition 1.5 (Patch Interpretation)**.** *Consider a monoid-with-involution homomorphism $f : \mathcal{P} \to \mathcal{I}_\mathcal{S}$,*

$$f(\varepsilon) = \mathrm{id}_\mathcal{S} \qquad f(vw) = f(v) \fatsemi f(w)$$
$$f(w^{-1}) = f(w)^{-1}$$

and the binary form $\varphi : \mathcal{S} \times \mathcal{P} \nrightarrow \mathcal{S}$ obtained by right uncurrying*:*

$$\varphi(s, w) = f(w)(s)$$

- The map f is determined completely by the generators, $f_0 : \mathcal{P}_0 \to \mathcal{I}_\mathcal{S}$.
- The relation $v \equiv w \iff f(v) = f(w)$ is the *word problem* for f_0.
- The binary map φ is a *partial right semigroup action*.

Definition 1.6 (Transactions)**.** *Given a generating map $f_0 : \mathcal{P}_0 \to \mathcal{I}_\mathcal{S}$, the* transactions *$\mathcal{T}$ are the image of the unique homomorphic extension $f : \mathcal{P} \to \mathcal{I}_\mathcal{S}$.*

Any model of this theory needs to specify just $\mathcal{S}$, $\mathcal{P}_0$ and f_0/φ_0.

- The binary form $\varphi : \mathcal{S} \times \mathcal{P} \nrightarrow \mathcal{S}$ formalizes the `patch` operation.
- The word problem formalizes the semantic equivalence of patch expressions.

The state space is assumed to contain no unreachable elements:

Definition 1.7 (Construction)**.** *A transaction t is called a* construction *of state s iff $t(s_0) = s$.*

Requirement 2 (Constructivity)**.** *Every state shall have a construction.*

A composite patch denoting a particular construction of the current state is the essential information in the *history* file of a repository.

2 Introducing the µ-ed Model

Definition 2.1 (µ-ed Model States)**.** *$\mathcal{S}^\mu = \Sigma^*$ is the set of words, over some finite alphabet Σ with more than one element, without loss of generality taken as the printable ASCII characters, with initial state $s_0^\mu = \varepsilon$.*

Our examples shall only use some letters $\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z} \in \Sigma$. In software terms, a µ-ed state is a single string which can be understood as the current content of a certain anonymous text document, which is initially empty. The model does not consider multiple named files or line numbers, which are prevalent in other formal approaches [3, 5, 7]; see also below.

Definition 2.2 (µ-ed Patches)**.** *Patches are generated by a single parametric elementary patch operation.*

$$\mathcal{P}_0^\mu = \{\mathsf{ins}(k, x) \mid k \in \mathbb{N}; x \in \Sigma\}$$

The semantics is specified concisely as

$$\varphi^\mu\big(s_0 \ldots s_{n-1}, \mathsf{ins}(k, x)\big) = s_0 \ldots s_{k-1} x s_k \ldots s_{n-1}$$

where $(0 \le k \le n)$.

This suffices to specify the model completely within the framework detailed above. The verbalized semantics of $\mathsf{ins}(k, x)$ is "insert character x after the first k characters if there are enough; otherwise fail."

For convenience, we write the inverse operation $\mathsf{ins}(k, x)^{-1}$ as $\mathsf{del}(k, x)$. The verbalized semantics of $\mathsf{del}(k, x)$ is "delete character x after the first k characters if it does occur there; otherwise fail." A patch word is a finite sequence of ins and del operation instances.

Note that the, perhaps intuitively more natural, family of deletion operations $\mathsf{del}(k)$ with the semantics "delete the next character (whatever) after the first k characters" is not admissible, since these are not invertible. This demonstrates that patches in the theoretical sense discussed here are not user commands, but a specification language for the history of a repository, to be inferred by tools.

Proposition 2.3. *The µ-ed model is not particularly well-behaved with respect to various basic semigroup properties:*

1. *$\mathcal{T}^\mu$ is not* finite*, thus it cannot be characterized easily by combinatorial techniques.*
2. *$\mathcal{T}^\mu$ is not* commutative*. More precisely, the situation is complex:*
   a) *There are no two distinct elementary patches whose associated transactions commute.*
   b) *But there are many compound patches for which they do, for example pairs of length-preserving character replacements at disjoint positions:*

$$\frac{\begin{array}{c} v = \mathsf{del}(1, \mathtt{a})\,\mathsf{ins}(1, \mathtt{b}) \\ w = \mathsf{del}(2, \mathtt{b})\,\mathsf{ins}(2, \mathtt{a})\,\mathsf{del}(3, \mathtt{c})\,\mathsf{ins}(3, \mathtt{d}) \end{array}}{f^\mu(vw) = f^\mu(wv)}$$

3. *$\mathcal{T}^\mu$ has a zero element $0 = \mathrm{id}_\varnothing$.*
   a) *Thus the semigroup is not* cancellative*, and cannot be mapped straightforwardly to a group for further study. [1]*
   b) *There are* orthogonal *elements, $t, u \neq 0$ such that $tu = 0 = ut$; for example contradictory pairs of deletion followed by reinsertion:*

$$\frac{\begin{array}{c} v = \mathsf{del}(1, \mathtt{a})\,\mathsf{ins}(1, \mathtt{a}) \\ w = \mathsf{del}(1, \mathtt{b})\,\mathsf{ins}(1, \mathtt{b}) \end{array}}{f^\mu(vw) = 0 = f^\mu(wv)}$$

4. *The action of $\mathcal{T}^\mu$ on $\mathcal{S}^\mu$ is not* free*: In general for a given state pair s, s' there are many transactions t such that $t(s) = s'$. For example:*

$$\begin{aligned} f^\mu\big(\mathsf{del}(1, \mathtt{o})\big)(\mathtt{root}) &= \mathtt{rot} \\ &= f^\mu\big(\mathsf{del}(2, \mathtt{o})\big)(\mathtt{root}) \end{aligned}$$

This implies that the `diff` operation is not a formally well-posed problem. Note that the construction case $s = s_0$ is special; see section 4 below.

5. *The* group of units *of $\mathcal{T}^{\mu}$ is trivial; again see section 4. Thus there is no obvious group-like substructure of interest.*

Conjecture 2.4. *Some other questions are not yet settled, but appear plausible:*

1. *$\mathcal{T}^{\mu}$ is likely not* finitely generated.
2. *$\mathcal{T}^{\mu}$ is likely E^**-unitary.

Real-world tools that deal with patches on text documents, such as the aforementioned `diff` and `patch` families, usually implement a more complex two-dimensional indexing scheme, where the text is divided into *lines* of arbitrary length, and elementary insertion and deletion operate on whole lines without having to preserve individual line length. (Contiguous affected lines are furthermore aggregated into *hunks*.) This pragmatic variant enables subsuming many changes that occur frequently in practice (but which preserve line numbers) under the commutative case discussed above, at the cost of a coarser cover of the actual changes, hence narrower applicability of each patch, and more complicated interval logic.

For example the effect of the character-based patch $\mathsf{del}(217321, \mathtt{l})\,\mathsf{ins}(217322, \mathtt{u})$ on a particular document could be mapped to the following `diff` output, where the first line indicates the (identical) line numbers in the source and target documents:[1]

```
5464c5464
< non veni pacem mittere sed gladium
---
> non veni pacem mittere sed gaudium
```

3 Inverse Semigroups and Idempotents

The role of a possible neutral element 1 is deemphasized in inverse semigroups, such that an *inverse monoid* is not a particularly remarkable special case. Instead, a richer class of near-neutral elements is of interest, namely the *idempotents*, elements e that satisfy $ee = e$, which can be thought of as *tests*. In an inverse semigroup S, the idempotents form a *commutative* subsemigroup $E(S)$ which, by idempotence and commutativity combined, is conveniently seen as a *semilattice*. The neutral element, if one exists, is merely the maximum element of this structure.

In a symmetric inverse semigroup $\mathcal{I}_X$, the idempotents coincide with the *partial identities* id_Y where $Y \subseteq X$ [13]. Hence

- they are characterized uniquely by their support Y;
- composition coincides with intersection of supports;

$$\mathrm{id}_Y \fatsemi \mathrm{id}_Z = \mathrm{id}_{Y \cap Z}$$

- and the lattice order coincides with the inclusion order of supports.

$$\mathrm{id}_Y < \mathrm{id}_Z \iff Y \subset Z$$

Proposition 3.1. *With $\mathcal{T}^{\mu}$ being a fairly restricted subsemigroup of $\mathcal{I}_{S^{\mu}}$, the idempotents $E(\mathcal{T}^{\mu})$ are not all the partial identities on $S^{\mu} = \Sigma^*$. The $\cap$-semilattice $E(\mathcal{T}^{\mu})$ can be specified as follows:*

1. *There is the minimum element $0 = \mathrm{id}_{\varnothing}$; for instance denoted by a contradictory pair of insertion and subsequent deletion:*

$$f^{\mu}\big(\mathsf{ins}(0, \mathtt{a})\,\mathsf{del}(0, \mathtt{b})\big) = \mathrm{id}_{\varnothing}$$

2. *For every state word length n, there is the identity of all words of length $\geq n$; for instance denoted by a consistent pair of insertion and subsequent deletion:*

$$f^{\mu}\big(\mathsf{ins}(n, \mathtt{a})\,\mathsf{del}(n, \mathtt{a})\big) = \mathrm{id}_{\Sigma^n \circ \Sigma^*}$$

   Here and henceforth $\circ$ denotes the concatenation of languages. Note that the case $n = 0$ yields the maximum element $1 = \mathrm{id}_{\Sigma^*}$.

3. *For every $k \in \mathbb{N}; x \in \Sigma$, there is the identity of all words that have the character x at the position k; for instance denoted by a consistent pair of deletion and subsequent insertion:*

$$f^{\mu}\big(\mathsf{del}(k, x)\,\mathsf{ins}(k, x)\big) = \mathrm{id}_{\Sigma^k \circ \{x\} \circ \Sigma^*}$$

4. *The semilattice is closed under intersection of supports.*
5. *By the free nature of $\mathcal{T}^{\mu}$ generated by $\mathcal{P}_0^{\mu}$, there are no further idempotents.*

[1] This notorious typographical error has been committed in actual history in the *Book of Kells* (Matthew 10:34) [8]; however the character and line indices are fictitious.

In particular, tests for length $\leq n$ and all of their logical consequences are conspicuously absent from $E(\mathcal{T}^{\mu})$, and every idempotent support is closed under string extension: $Y \circ \Sigma^* \subseteq Y$. In the same vein of asymmetry, the semilattice $E(\mathcal{T}^{\mu})$ has *no atoms*; for every nonzero idempotent e, there is an even smaller nonzero idempotent $e' < e$: For instance, let $e = \mathrm{id}_Y$ and n be the length of the shortest word in Y. Now set $Y' = Y \cap (\Sigma^{n+1} \circ \Sigma^*)$; then clearly $0 < \mathrm{id}_{Y'} < \mathrm{id}_Y$.

The idempotents can be characterized by a simple regular expression-like formal language for prefix specification, and the semilattice operations by algorithms on that language:

Definition 3.2 (Patterns, Matching)**.** *Let $\Sigma_\square = \Sigma + \{\square\}$ be the alphabet Σ extended by a distinct wildcard symbol $\square$. Words in $R = {\Sigma_\square}^* + \{\infty\}$ are called* patterns*. A* matching *function $M : R \to \mathbb{P}(\Sigma^*)$ maps patterns to sublanguages:*

$$M(\varepsilon) = \Sigma^* \qquad M(x\alpha) = \{x\} \circ M(\alpha)$$
$$M(\infty) = \varnothing \qquad M(\square\alpha) = \Sigma \circ M(\alpha)$$

Note that the extra pattern ∞ is in fact a completion: $M(\infty) = \bigcap_{n=0}^{\infty} M(\square^n)$.

Proposition 3.3. *Idempotents are in one-to-one correspondence with patterns. The function M is injective and covers exactly the support of idempotents:*

$$E(\mathcal{T}^{\mu}) = \{\mathrm{id}_{M(p)} \mid p \in R\}$$

Definition 3.4 (Idempotent Subscripting)**.** *Idempotents are addressed by a pattern subscript.*

$$e_p = \mathrm{id}_{M(p)}$$

Some examples:

$$f^{\mu}\big(\mathsf{ins}(2,\mathtt{a})\,\mathsf{del}(2,\mathtt{a})\big) = e_{\square\square} \qquad f^{\mu}\big(\mathsf{del}(1,\mathtt{b})\,\mathsf{ins}(1,\mathtt{b})\big) = e_{\square\mathtt{b}}$$

The lattice order and meet operation can be defined as simple symbolic algorithms on patterns:

Definition 3.5 (Pattern Information Order)**.** *The* information order $\sqsubset$ *on patterns is the smallest partial order that entails the* extension*,* substitution *and* limit *rules, respectively:*

$$\alpha \sqsubset \alpha\square \qquad \alpha\square\beta \sqsubset \alpha x \beta \qquad \alpha \sqsubset \infty$$

Proposition 3.6. *The pattern information order is precisely the* opposite *of the semilattice order on $E(\mathcal{T})$.*

$$p \sqsubset q \iff M(p) \supset M(q) \iff e_p > e_q$$

Proposition 3.7. *Each pattern, except for ∞, has only finitely many predecessors in the information order.*

Definition 3.8 (Pattern Join)**.** *Let $\sqcup$ be a binary join operation on patterns, recursively defined:*

$$\varepsilon \sqcup \alpha = \alpha \qquad \square\alpha \sqcup \square\beta = \square(\alpha \sqcup \beta)$$
$$\infty \sqcup \alpha = \infty \qquad \square\alpha \sqcup x\beta = x(\alpha \sqcup \beta)$$
$$\alpha \sqcup \beta = \beta \sqcup \alpha \qquad x\alpha \sqcup x\beta = x(\alpha \sqcup \beta)$$
$$x\alpha \sqcup y\beta = \infty \qquad (x \neq y)$$

Proposition 3.9. *The pattern join is equivalent to composition/meet of idempotents.*

$$e_p e_q = \mathrm{id}_{M(p) \cap M(q)} = e_{p \sqcup q}$$

For any substructure of a symmetric inverse semigroup, $T \subseteq \mathcal{I}_X$, the idempotents of T correspond exactly to the subsets of X that actually occur as domain/range of the partial bijections in T. Thus, for the special case of the μ-ed model, the patterns can serve as a type language for transactions:

Definition 3.10 (Domain/Range Pattern)**.** *The* pattern domain/range *operators* $\mathrm{dop}, \mathrm{rap} : \mathcal{T}^{\mu} \to R$ *yield the unique pattern, respectively, such that:*

$$\mathrm{dom}\, t = M(\mathrm{dop}\, t) \iff t \fatsemi t^{-1} = e_{\mathrm{dop}\, t}$$
$$\mathrm{ran}\, t = M(\mathrm{rap}\, t) \iff t^{-1} \fatsemi t = e_{\mathrm{rap}\, t}$$

Proposition 3.11. *If $t(s) = s'$, then $\mathrm{dop}\, t \sqsubseteq s$ and $\mathrm{rap}\, t \sqsubseteq s'$.*

4 The Word Problem and Finitary Representations

The *word problem* is a question of particular interest in (semi)group theory: Given two group words v and w, does $v \equiv w$ hold? The problem is both hard and interesting; there is a vast body of literature on partial solutions [2, 4, 6, e.g.], but even for some finitely generated groups it is algorithmically undecidable. [9]

For the µ-ed model, a finitary unique representation of semigroup elements can be designed and computed effectively, thus the word problem for $f^{\mu} : \mathcal{P}^{\mu} \to \mathcal{T}^{\mu}$ is decidable, even though the generator set $\mathcal{P}_0^{\mu}$ is infinite.

The general idea is to assign to each transaction a canonical instance from a suitable class of string-transducing automata as its implementation. The chosen automaton class, termed *µ-ed-automata*, is much simpler than some that have been used to decide word problems. [11] It is neither of the classical Mealy or Moore types, but has a distinct alternating input–output behavior, befitting the independent elementary transaction types ins and del. The automata are state-finite and their transition graphs are linear; neither branches nor cycles occur. Thus their transition graph structure can be encoded as a collection of algebraic data types, and constructed and interpreted efficiently with simple recursive algorithms.

Definition 4.1 (Syntax of µ-ed-Automata)**.**

$$\mathrm{A} ::= \textsf{TRY}\,\mathrm{I} \mid \textsf{FAIL} \qquad \begin{aligned} \mathrm{I} &::= \textsf{INS}\,\Sigma^*\,\mathrm{C} \\ \mathrm{C} &::= \textsf{DEL}\,\Sigma\,\mathrm{I} \mid \textsf{SKIP}\,\mathrm{I} \mid \textsf{RET} \end{aligned}$$

The operation of an automaton (A) alternates between *insertion* (I) and *consumption* (C) mode. The intended operations are fairly evident: FAIL rejects any input right away; TRY starts processing with an initial insertion; INS inserts a substring into the output; SKIP consumes an arbitrary input character and passes it to the output, or rejects empty input; DEL consumes an input character if matched and produces no output, or rejects empty input or unexpected characters; RET passes all remaining input to the output and halts.

This informal specification can be implemented by a family of mutually linearly recursive partial transition functions.

Definition 4.2 (Semantics of µ-ed-Automata)**.** *The family of partial functions* $\delta_{\Diamond} : \Sigma^* \times \Diamond \rightharpoonup \Sigma^*$ *with* $\Diamond = \mathrm{A}, \mathrm{I}, \mathrm{C}$ *is defined by*

$$\begin{aligned} \delta_{\mathrm{A}}(s, \textsf{TRY}\,i) &= \delta_{\mathrm{I}}(s, i) \\ \delta_{\mathrm{I}}(s, \textsf{INS}\,t\,c) &= t\delta_{\mathrm{C}}(s, c) \end{aligned} \qquad \begin{aligned} \delta_{\mathrm{C}}(xs, \textsf{SKIP}\,i) &= x\delta_{\mathrm{I}}(s, i) \\ \delta_{\mathrm{C}}(xs, \textsf{DEL}\,x\,i) &= \delta_{\mathrm{I}}(s, i) \\ \delta_{\mathrm{C}}(s, \textsf{RET}) &= s \end{aligned}$$

where all missing cases count as undefined. We also consider the curried variant $d_{\mathrm{A}} : A \to \mathcal{I}_{\Sigma^*}$*.*

Proposition 4.3. *For any µ-ed-automaton* a*,* $d_A(a)$ *is in fact a transaction. More precisely, there is a recursive choice function* $z : A \to \mathcal{P}^{\mu}$ *such that:*

$$d_A(a) = f^{\mu}\big(z(a)\big)$$

Note that, whereas the elementary µ-ed-patches ins and del operate as a pipeline, each on the *output* of the preceding patch, the corresponding µ-ed-automaton operations INS and DEL operate on (the current remainder of) the *input* string, which makes reasoning a lot easier.

The above representation of transactions is very nearly unique. The only remaining degree of freedom is due to the fact that INS operations immediately before and after a DEL operation are behaviorally undistinguishable. This is easily remedied by lumping INS arguments on one side, say before DEL:

$$\textsf{INS}\,t\ \textsf{DEL}\,x\ \textsf{INS}\,u \quad\longmapsto\quad \textsf{INS}(tu)\ \textsf{DEL}\,x\ \textsf{INS}\,\varepsilon$$

Definition 4.4 (Normal Form)**.** *A µ-ed-automaton* $a \in A$ *is called* normal *iff any subexpression of the form* DEL x INS u *has* $u = \varepsilon$*. We write* $A_0 \subset A$ *for the set of normal µ-ed-automata, and* $N : A \to A_0$ *for the rewriting of automata according to the above rule.*

Proposition 4.5. *Normal µ-ed-automata are unique representatives for the semantic equivalence classes of µ-ed-automata.*

$$\begin{gathered} d_{\mathrm{A}}\big(N(a)\big) = d_{\mathrm{A}}(a) \\ N(a) = N(a') \iff d_{\mathrm{A}}(a) = d_{\mathrm{A}}(a') \end{gathered}$$

Sketch. The former equation, preservation of semantics, is easy. For the latter, a counterexample s such that $\delta_{\mathrm{A}}(s, a) \neq \delta_{\mathrm{A}}(s, a')$ can be constructed by a structurally recursive function for every pair of distinct normal automata. □

The µ-ed-automata are suitable for deciding the word problem for $f^{\mu} : \mathcal{P}^{\mu} \to \mathcal{T}^{\mu}$, provided that an equivalent normal µ-ed-automaton for any patch can be constructed effectively.

Proposition 4.6. *Given a computable function* $Q : \mathcal{P}^{\mu} \to A$*, such that* $f^{\mu}(p) = d_{\mathrm{A}}\big(Q(p)\big)$*, the word problem is decidable.*

$$p \equiv q \iff N\big(Q(p)\big) = N\big(Q(q)\big)$$

$$
\begin{aligned}
Q(p) &= \begin{cases} \mathsf{TRY}\, Q_{\mathrm{I}}(\mathsf{INS}\, \varepsilon\, \mathsf{RET}, p) & \text{if defined} \\ \mathsf{FAIL} & \text{otherwise} \end{cases} \\
Q_{\mathrm{I}}\big(\mathsf{INS}\, s\, c, \mathsf{ins}(k,x)\big) &= \begin{cases} \mathsf{INS}\, \varphi^{\mu}\big(s, \mathsf{ins}(k,x)\big)\, c & \text{if } k \le |s| \\ \mathsf{INS}\, s\, Q_{\mathrm{C}}\big(c, \mathsf{ins}(k-|s|, x)\big) & \text{otherwise} \end{cases} \\
Q_{\mathrm{I}}\big(\mathsf{INS}\, s\, c, \mathsf{del}(k,x)\big) &= \begin{cases} \mathsf{INS}\, \varphi^{\mu}\big(s, \mathsf{del}(k,x)\big)\, c & \text{if } k < |s| \\ \mathsf{INS}\, s\, Q_{\mathrm{C}}\big(c, \mathsf{del}(k-|s|, x)\big) & \text{otherwise} \end{cases} \qquad (*) \\
Q_{\mathrm{C}}\big(\mathsf{DEL}\, x\, i, \mathsf{ins}(k,x)\big) &= \mathsf{DEL}\, x\, Q_{\mathrm{I}}\big(i, \mathsf{ins}(k,x)\big) \\
Q_{\mathrm{C}}\big(\mathsf{DEL}\, x\, i, \mathsf{del}(k,x)\big) &= \mathsf{DEL}\, x\, Q_{\mathrm{I}}\big(i, \mathsf{del}(k,x)\big) \\
Q_{\mathrm{C}}\big(\mathsf{SKIP}\, i, \mathsf{ins}(k,x)\big) &= \mathsf{SKIP}\, Q_{\mathrm{I}}\big(i, \mathsf{ins}(k-1,x)\big) \\
Q_{\mathrm{C}}\big(\mathsf{SKIP}\, i, \mathsf{del}(k,x)\big) &= \begin{cases} \mathsf{DEL}\, x\, i & \text{if } k = 0 \\ \mathsf{SKIP}\, Q_{\mathrm{I}}\big(i, \mathsf{del}(k-1,x)\big) & \text{if } k > 0 \end{cases} \\
Q_{\mathrm{C}}\big(\mathsf{RET}, \mathsf{ins}(k,x)\big) &= \mathsf{SKIP}\, Q_{\mathrm{I}}\big(\mathsf{INS}\, \varepsilon\, \mathsf{RET}, \mathsf{ins}(k-1,x)\big) \\
Q_{\mathrm{C}}\big(\mathsf{RET}, \mathsf{del}(k,x)\big) &= \begin{cases} \mathsf{DEL}\, x\, \mathsf{INS}\, \varepsilon\, \mathsf{RET} & \text{if } k = 0 \\ \mathsf{SKIP}\, Q_{\mathrm{I}}\big(\mathsf{INS}\, \varepsilon\, \mathsf{RET}, \mathsf{del}(k-1,x)\big) & \text{if } k > 0 \end{cases}
\end{aligned}
$$

Figure 1: µ-ed Automaton Construction Algorithm

Definition 4.7 (Automaton Construction)**.** *We give a construction of µ-ed-automata for patches that follows the functional pattern of right actions, leading naturally to an* insertion sort *strategy, where patch words act (insert) elementwise. The algorithm is depicted in Fig. 1.*

- *The main function $Q : \mathcal{P}^{\mu} \to \mathrm{A}$ initializes the automaton to be acted on with the identical one, and catches undefined cases with the* FAIL *operation.*
- *The actual work is done by two mutually recursive partial functions, $Q_{\Diamond} : \Diamond \times \mathcal{P}^{\mu} \nrightarrow \Diamond$, that are right monoid actions of patches on automaton fragments. Thus it suffices to specify the action of generators* ins *and* del*, complemented by*

$$Q_{\Diamond}(x, \varepsilon) = x \quad Q_{\Diamond}(x, pq) = Q_{\Diamond}\big(Q_{\Diamond}(x,p), q\big)$$

- *Q_{I} descends into strings that are already being inserted into the output if the concerned index is in range, or otherwise adjusts indices and defaults to the remainder of the automaton. Note that equation (*) has two special features:*
  1. *φ^{μ} there is the only source of undefined cases that get mapped to* FAIL*, by attempting to delete characters inconsistent with what has been inserted beforehands;*
  2. *Q_{C} is invoked for $\mathsf{ins}(k,x)$ only with $k > 0$.*
- *The complementary Q_{C} acts with compensation for characters being deleted from the output, and extends the automaton if necessary.*

Proposition 4.8. *This construction is indeed semantically sound.*

$$f^{\mu}(p) = d_{\mathrm{A}}\big(Q(p)\big)$$

The resulting algorithm does not only decide the word problem for µ-ed patches, but is also reasonably efficient.

Proposition 4.9. *If the operation* SKIP *is replaced by the slightly more complex operation* $\mathsf{SKIP}^*\, n$ *with $n > 0$ and the meaning*

$$\mathsf{SKIP}^*\, 1 = \mathsf{SKIP}$$
$$\mathsf{SKIP}^*(m+n) = \mathsf{SKIP}^*\, m\, \mathsf{INS}\, \varepsilon\, \mathsf{SKIP}^*\, n$$

then the time complexity of the algorithm is $\mathcal{O}(n^2)$, as for ordinary insertion sort, while the size of the resulting automaton is $\mathcal{O}(n)$. Without SKIP^**, exponential blow-up is possible, e.g. consider $Q\big(\mathsf{ins}(999999999, \mathtt{a})\big)$.*

Conjecture 4.10. *The µ-ed word problem should be similar enough to ordinary sorting that an algorithm of optimal time complexity $\mathcal{O}(n \log n)$ exists.*

5 Consequences

The unique normal form of µ-ed-automata makes reasoning not only about the word problem, but also about some other basic semigroup properties remarkably easy. In contrast to the rest of the present paper, proofs are given fully, because they are both concise and illuminating.

Corollary 5.1. *Constructing transactions are unique in the µ-ed model.*

$$t(s_0^{\mu}) = t'(s_0^{\mu}) \implies t = t'$$

Proof. Given a state $s \in \Sigma^*$ to be constructed, find a normal µ-ed-automaton a_s such that $\delta_{\mathrm{A}}(\varepsilon, a_s) = s$. It is easy to see that any automaton a such that $\delta_{\mathrm{A}}(\varepsilon, a)$ is even defined, has to be of the form $a = \mathsf{TRY}\,\mathsf{INS}\,u\,\mathsf{RET}$; neither FAIL, SKIP nor DEL can be involved, since they would reject ε. Then we find also that $\delta_{\mathrm{A}}(\varepsilon, a) = u$. It follows that $a_s = \mathsf{TRY}\,\mathsf{INS}\,s\,\mathsf{RET}$. □

Corollary 5.2. *The group of units of $\mathcal{T}^{\mu}$ is trivial.*

$$tt^{-1} = 1 = t^{-1}t \implies t = 1$$

Proof. For the idempotents $tt^{-1}, t^{-1}t$ to equal $1 = \mathrm{id}_{\Sigma^*}$, t must be both left-total and right-total. By extension of the preceding proof, it is easy to see that any µ-ed-automaton a, such that $t = d_{\mathrm{A}}(a)$ has this property, must be of the exact form $a = \mathsf{TRY}\,\mathsf{INS}\,\varepsilon\,\mathsf{RET} = Q(\varepsilon)$, since any other operation would cause a either not to accept or not to produce certain strings. It follows that $d_{\mathrm{A}}(a) = 1$. □

The latter result can be strengthened in interesting ways.

5.1 The Diff Problem, Well-Posed

We have argued above that the `diff` problem of finding the transaction that connects two given states is ill-posed. Interestingly, this does not hold in a more abstract form, namely if certain well-shaped sets of states are given as the domain and range of a transaction.

Consider the specification of a transaction by domain and range patterns as defined in section 3.

Definition 5.3 (Box Counting)**.** *The* box counting *function on patterns, $b : R \to \mathbb{N} + \{\infty\}$ is defined in the obvious way.*

$$b(\varepsilon) = 0 \qquad b(x) = 0$$
$$b(\infty) = \infty \qquad b(\square) = 1$$
$$b(\alpha\beta) = b(\alpha) + b(\beta)$$

Corollary 5.4. *Transactions are uniquely sorted and preserve box count.*

$$\#\{t \in \mathcal{T}^{\mu} \mid \mathrm{dop}\, t = p \wedge \mathrm{rap}\, t = q\} = \begin{cases} 1 & b(p) = b(q) \\ 0 & b(p) \neq b(q) \end{cases}$$

Proof. Similarly to Corollaries 5.1 and 5.2, implement dop and rap as nearly orthogonal projections of automata, $\alpha_{\Diamond}, \omega_{\Diamond} : \Diamond \to R$:

$$\alpha_{\mathrm{A}}(\mathsf{TRY}\,i) = \alpha_{\mathrm{I}}(i) \qquad \alpha_{\mathrm{I}}(\mathsf{INS}\,s\,c) = \alpha_{\mathrm{C}}(c)$$
$$\alpha_{\mathrm{A}}(\mathsf{FAIL}) = \infty \qquad \alpha_{\mathrm{C}}(\mathsf{DEL}\,x\,i) = x\,\alpha_{\mathrm{I}}(i)$$
$$\alpha_{\mathrm{C}}(\mathsf{SKIP}\,i) = \square\,\alpha_{\mathrm{I}}(i) \qquad \alpha_{\mathrm{C}}(\mathsf{RET}) = \varepsilon$$
$$\omega_{\mathrm{A}}(\mathsf{TRY}\,i) = \omega_{\mathrm{I}}(i) \qquad \omega_{\mathrm{I}}(\mathsf{INS}\,s\,c) = s\,\omega_{\mathrm{C}}(c)$$
$$\omega_{\mathrm{A}}(\mathsf{FAIL}) = \infty \qquad \omega_{\mathrm{C}}(\mathsf{DEL}\,x\,i) = \omega_{\mathrm{I}}(i)$$
$$\omega_{\mathrm{C}}(\mathsf{SKIP}\,i) = \square\,\omega_{\mathrm{I}}(i) \qquad \omega_{\mathrm{C}}(\mathsf{RET}) = \varepsilon$$

It is easy to see that these are semantically sound,

$$\alpha_{\mathrm{A}}\big(Q_{\mathrm{A}}(p)\big) = \mathrm{dop}\, p \qquad \omega_{\mathrm{A}}\big(Q_{\mathrm{A}}(p)\big) = \mathrm{rap}\, p$$

and that, furthermore,

- distinct *normal* automata are mapped to distinct pattern pairs, and
- box count is preserved.

□

This last result has more far-ranging implications than meet the eye, both positive and negative.

On the upside, it allows for the objective identification of `diff` transactions if enough instance pairs are given. For example, the ambiguous pair `root` ↦ `rot` has two solutions, namely $\mathrm{del}(1, \mathtt{o})$ and $\mathrm{del}(2, \mathtt{o})$. They can be disambiguated by just one additional, suitably chosen pair; say `boat` ↦ `bat` or `buoy` ↦ `buy`, respectively.

Furthermore, any transaction can be specified uniquely by naming its domain and range patterns:

Definition 5.5 (Transaction by Type)**.** *Let $p, q \in R$ be patterns with $b(p) = b(q)$. Then we write q/p for the transaction $t \in \mathcal{T}^{\mu}$ such that $p = \mathrm{dop}\, t$ and $q = \mathrm{rap}\, t$.*

Note that, as a funny coincidence of abuse of notation, we have $0 = \infty/\infty$.

Using Propositions 3.7 and 3.11, it can be decided whether an arbitrary finite set of instance pairs is *consistent*, i.e., included in any transaction. For each instance pair $s \mapsto s'$, the amenable transactions $\{t \mid t(s) = s'\}$ can be enumerated with brute force, by searching the finite candidate set $\{q/p \mid p \sqsubseteq s \wedge q \sqsubseteq s'\}$. The pair set is consistent iff the intersection of the elementwise results is nonempty.

On the downside however, the innocuous-looking Corollary 5.4 thoroughly obstructs an intuitively appealing approach to parallel composition of patches and the `diff3`/`merge` problem. This line of reasoning shall be explored in the next section.

6 A Theoretical Account of Parallel Composition

The third basic tool archetype, `diff3`, solves a more complex problem than either `patch` or `diff`: Given two patches acting divergently on the same base state s,

$$\varphi(s, p_1) = s_1 \qquad \varphi(s, p_2) = s_2$$

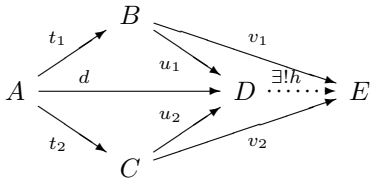
find a supposedly intended joint target state s', or fail if there is a conflict. More abstractly, given two patches p_1, p_2, find *complements* q_1, q_2 such that $p_1 q_1 \equiv p_2 q_2$, where q_1 is "essentially p_2 but after p_1", and vice versa. The `merge` operation is a more heuristic variant, where only s_1, s_2 are given, and s is inferred as the plausible common ground.

There is an obvious solution for *commuting* transactions [3]: If $p_1 p_2 \equiv p_2 p_1$ holds, simply put $q_1 = p_2$ and $q_2 = p_1$. In the non-commuting case, the problem is generally ill-posed: The informal specification given in the preceding paragraph does not suffice to decide whether solutions exist at all, or which solution is the canonical one.

Actual `merge`-like tools solve particular benign cases heuristically with results that appeal to intuition in an 'I-know-it-when-I-see-it' way. [3] For example, the concurrent insertion of two lines into disjoint regions of a document is routinely resolved by `merge` implementations, with the line number of the textually later insertion implicitly increased to account for the earlier one.

Nevertheless, these workable cases are not well-understood theoretically, and the power of heuristics is practically limited. As a prominent counterexample, consider the case where p_1 denotes an *encryption* function, such as DSA. It is a manifest purpose of encryption to make the problem of finding q_1, such that an attacker can choose p_2 and edit the document in encrypted form accordingly, as hard as possible. In general, a patch p_i may change the state globally in such a way that intuition fails to recognize any solution q_i as the essentially intended one.

By contrast, in pure theory the situation seems clear enough, at least to the unwary. The problem of parallel composition in the semantic domain is posed very naturally [7] in the language of category theory as a *pushout*:



Given two morphisms t_1, t_2, construct two complementary morphisms u_1, u_2, such that $t_1 \fatsemi u_1 = t_2 \fatsemi u_2 = d$, and that, for every other such pair v_1, v_2 with $t_1 \fatsemi v_1 = t_2 \fatsemi v_2$, there is a unique morphism h such that $u_1 \fatsemi h = v_1$ and $u_2 \fatsemi h = v_2$. Then the diagonal d is the parallel composition of t_1 and t_2. [2]

Most unfortunately, the approach conflicts with the partiality and invertibility of transactions in subtle ways. There are several obvious ways to form a category of states and transactions, but none of those has quite the right properties.

For one, the inverse semigroup $\mathcal{T}$ can be seen as a monoid, and naturally represented as a category with one object, S, and partial morphisms. This category has generally *too few* pushouts: Consider the problem of merging the µ-ed transaction pair $t_1 = f^{\mu}(\mathsf{ins}(0, \mathtt{a}))$ and $t_2 = f^{\mu}(\mathsf{ins}(1, \mathtt{b}))$, whose I-know-it-when-I-see-it solution is $u_1 = f^{\mu}(\mathsf{ins}(2, \mathtt{b}))$ and $u_2 = f^{\mu}(\mathsf{ins}(0, \mathtt{a}))$; cf. [3]. Clearly, these form a commuting square $t_1 \fatsemi u_1 = t_2 \fatsemi u_2 = \mathtt{a}\square\mathtt{b}/\square$.

However, there is another completion, $v_1 = f^{\mu}(\mathsf{del}(0, \mathtt{b}))$ and $v_2 = f^{\mu}(\mathsf{del}(1, \mathtt{a}))$, that forms a commuting square $t_1 \fatsemi v_1 = t_2 \fatsemi v_2 = 0$, but there

[2] Note that we keep using relational order of morphisms, whereas category theoreticians usually prefer the opposite functional order.

is no mediating morphism h, such that $u_1 \fatsemi h = v_1$ and $u_2 \fatsemi h = v_2$. To see this, instantiate both equations with the argument `ba`:

$$h(u_1(\texttt{ba})) = v_1(\texttt{ba}) \qquad h(u_2(\texttt{ba})) = v_2(\texttt{ba})$$
$$h(\texttt{bab}) = \texttt{a} \qquad h(\texttt{aba}) = \texttt{b}$$

The instance pairs `bab` $\mapsto$ `a` and `aba` $\mapsto$ `b` can be shown to be inconsistent by the method described in section 5.1. Thus the advertised solution u_1, u_2 is not a pushout.

Alternatively, a category with total morphisms can be constructed, given a representation of the domains and ranges of transactions. For the μ-ed model, this requires one object per pattern. All morphisms are then of the form $q/p : p \to q$. This category is *thin* and *invertible*, and has *too many* pushouts: Consider an arbitrary pair of transactions $t_1 : A \to B$ and $t_2 : A \to C$. The completion $u_1 = t_1^{-1} : B \to A$ and $u_2 = t_2^{-1} : C \to A$ is not only a commuting square with diagonal $d = \mathrm{id}_A$. Since all morphisms are unique iso, it is also a pushout. This leads to the absurd interpretation that any two transactions are correctly "merged" by performing neither!

Conjecture 6.1. *Invertible partial transactions are fundamentally incommensurable with a pushout-based definition of parallel composition.*

7 Conclusion

7.1 Related Work

Many heuristic algorithms for the `patch`, `diff` and `merge` scenarios have been implemented in practice and studied in operational theory, that is with respect to their efficiency and coverage of frequently occuring cases. See [3] for a summary of the literature on their algebraic foundation.

The approach that comes closest to the one presented here, both in spirit and outcomes, is the conceptual framework behind the `darcs` system. There, in contrast to the state focus of most RCSs, patches take center stage. The algorithms have been founded on arguments using the language of non-commutative groups originally, and reworked to inverse semigroups later [3]. The approach considers both patches that commute by themselves and those that can be transposed by taking complements. However, complements are defined in the I-know-it-when-I-see-it way, and the formal study in terms of standard semigroup concepts has not been worked out in detail.

The pushout approach to parallel composition has been worked out in great detail in [7]. Their interesting approach is an axiomatic one that postulates closure under pushouts first, and works out a model only subsequently. However, it appears that their theory does not solve the problems discussed in section 6. Ominously, results are developed for the insertion-only theory first, where uniqueness is cheap (cf. Corollary 5.1). There is no mention of an investigation what it means for a pushout to be unique up to isomorphism in the full, invertible theory with deletions.

In [5] an algebraic model is given that deals mostly with aspects of repository structure that are complementary to our reference model, such as file organization. In that work, an inverse semigroup action structure is implied, but not worked out in detail.

7.2 Summary and Outlook

We have characterized the μ-ed model using various standard concepts of semigroup theory, most notably an automata-theoretic approach leading to efficient decidability of the word problem and various corollaries. Potential variations and extensions of the elementary patch vocabulary can be judged with respect to how well they preserve these beneficial properties.

Other tools of semigroup theory may come to mind, but have not yet been tried. Notably, we do not yet have a clear picture of (a) ideals and Green's relations for transactions, (b) Munn representations, and (c) the algebraic behavior of the zero transaction.

On the negative side, we have also explored some fundamental difficulties on the way to a comprehensive theory, in particular with respect to a high-level theory of parallel composition. This leaves interesting opportunities for future work, in the sense of either a dialectical resolution or a proof of impossibility.

References

[1] A. H. Clifford and G. B. Preston. *The algebraic theory of semigroups*. Vol. 2. American Mathematical Society, 1967.

[2] Max Dehn. “Über unendliche diskontinuierliche Gruppen”. In: *Mathematische Annalen* 71.1 (1911), pp. 116–144.

[3] Judah Jacobson. *A formalization of darcs patch theory using inverse semigroups*. Tech. rep. UCLA, 2009. URL: ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf.

[4] D. Knuth and P. Bendix. “Simple word problems in universal algebras”. In: *Computational Problems in Abstract Algebra*. 1970, pp. 263–297.

[5] Andreas Löh, Wouter Sierstra, and Daan Leijen. “A principled approach to version control”. In: *Proc. FASE*. 2007.

[6] C. F. Miller. “Decision problems for groups – survey and reflections”. In: *Algorithms and Classification in Combinatorial Group Theory*. Springer, 1991, pp. 1–60.

[7] Samuel Mimram and Cinzia Di Giusto. “A Categorical Theory of Patches”. In: *Electronic Notes in Theoretical Computer Science* 298 (2013). Proc. MFPS XXIX, pp. 283–307.

[8] George Jean Nathan and Henry Louis Mencken. In: *The American Mercury* (1951), p. 572.

[9] P. S. Novikov. “On the algorithmic unsolvability of the word problem in group theory”. In: *Proceedings of the Steklov Institute of Mathematics* 44 (1955), pp. 1–143.

[10] G. B. Preston. “Representations of inverse semi-groups”. In: *Journal of the London Mathematical Society* 29.4 (1954), pp. 411–419.

[11] Robert Shapiro and Michael Gilman. *On groups whose word problem is solved by a nested stack automaton*. Tech. rep. 1998. arXiv: math/9812028.

[12] Walter F. Tichy. “RCS – a system for version control”. In: *Software: Practice and Experience* 15.7 (1985), pp. 637–654.

[13] V. V. Wagner. “Generalised groups”. In: *Proceedings of the USSR Academy of Sciences* 84 (1952), pp. 1119–1122.

Maßeinheiten als Typen: eine mathematische Remodellierung und Die nächsten 197 Maßeinheiten-Bibliotheken – Realistische Algebra trifft auf realistische Typsysteme

Baltasar Trancón y Widemann
Nordakademie Elmshorn
baltasar@trancon.de

Markus Lepper
semantics gGmbH, Berlin

Zusammenfassung

Ein dankbares Anwendungsgebiet für die Methode der *mathematischen Remodellierung* (siehe die Zusammenfassung Lepper und Trancón 2021 in diesem Band) ist das System der physikalischen Maßeinheiten.

Es gibt zwar, wie zu erwarten, eine lang andauernde und breite Diskussion auf professionellem Niveau über Detailfragen und Einzelentscheidungen, aber überraschenderweise kaum eine Diskussion der mathematischen Grundlagen dieses Systems.

Die hier vorgestellte Theorie [1] basiert auf einem System von Abel'schen Gruppen, das sowohl die Algebra der multiplizierbaren Maßeinheiten als auch deren faktorbestimmenden Präfices einheitlich beschreiben kann.

Aus diesem ergibt sich auf natürliche Weise eine Hierarchie von Abschluss-Eigenschaften. Aus diesen können Transformationen abgeleitet werden, die auf einfache Weise zur programmierten Lösung von Standardproblemen verwendet werden können (Normierung, Vergleich, etc.)

Es zeigt sich, dass einerseits nicht wenige der in der konventionellen und semiformalen Behandlung des Themas vorgenommenen Vereinfachungen bei genauer Betrachtung unzulässig sind. Aber auch, dass umgekehrt Sonderbehandlungen die in Praxis und semiformaler Theorie anzutreffen sind, besser aufgegeben würden, da ihnen keine mathematisch-strukturelle Substanz zugrunde liegt.

Literatur

[1] Baltasar Trancón y Widemann and Markus Lepper. "Towards a Theory of Conversion Relations for Prefixed Units of Measure". In: *Proceedings of the 20th International Conference on Relational and Algebraic Methods in Computer Science*. https://arxiv.org/abs/2212.11580[20231013]. Augsburg: Springer LNCS 13896, 2023.

Funcode – ausführbare Spezifikation in Prolog. Ein Erfahrungsbericht.

Markus Lepper
semantics gGmbH, Berlin
post@mlepper.de

Baltasar Trancón y Widemann
Nordakademie Elmshorn

Zusammenfassung

Ein dankbares Anwendungsgebiet für die Methode der *mathematischen Remodellierung* (siehe die Zusammenfassung Lepper und Trancón 2021 in diesem Band) sind die unterschiedlichen Symbolsysteme der *funktional-harmonischen Musiktheorie*.

Dies ist eine von im weitesten Sinne (beginnend mit Rameau) seit dreihundert Jahren, im engeren Sinne seit einhundertfünfzig Jahren (Huge Riemann) sich entwickelnde Familie von Theorien, welche die harmonische Wirkung von musikalischen Zusammenklängen verschiedener Tonhöhen dadurch versucht systematisch zu erklären, dass sowohl den Einzeltönen im Verhältnis zu ihren Nachbarn, also auch den so entstehenden Gruppierungen (“Akkorden”) untereinander bestimmte *Funktionen* zusgeordnet werden.

Die kognitive und emotionale Wirkung von Tonmengen und -folgen beruhe demnach auf der Abfolge der verschiedenen Funktionen. Diese Folgen sind relative, transponierbare Phänomene.

Im Laufe der Zeit haben sich verschiedenartigste Theorien herausgebildet, teils in scharfem Streit. Alle kommen mit ihrem *je eigenen Symbolsystem* einher: Buchstaben, Zahlen, Graphiken und den Regeln zu deren Kombinierung, mit denen die erkannten und vermeintlich wirksamen Funktionen eindeutig bezeichnet werden sollen.

In der kontinentaleuropäischen Lehr- und Forschungspraxis hat sich seit fast achtzig Jahren das System von Grabner und Maler durchgesetzt. Obwohl dieses nur eine einzige der ca. dreißig oder mehr Schulen repräsentiert, gibt es allein innerhalb seiner eine weite Streuung unterschiedlicher Verwendung und damit der mutmaßlichen Semantik.

Die mathematischen Remodellierung wurde diesmal sogar *ausführbar* durch die Verwendung von Prolog als Meta-Sprache: Bei Eingabe eines beliebig tief geschachtelten Funktionstermes und eines tonalen Zentrums berechnet die Spezifikation die Abfolge der Tonklassen als Euler-Koordinaten.

Die Veröffentichung erfolgte zusätzlich als technischer Bericht im Stile des “literate programming” [2] und enthält den gesamten Quelltext.

Die Funktionalharmonische Theorie steht in deutlichem Gegensatz zu der “Stufentheorie” oder “skalenbasierten Hamonielehre”, die im englischsprachigen Raum dominiert, und in dortigen Publikationen oft gar als “allgemein anerkanntes Standardmodell” behauptet wird.

Die Positionierung unserer Publikationen in einem englischsprachigen Journal [1] war deshalb nicht zuletzt auch eine interkulturelle Vermittlungsarbeit, in deren Zuge auch einige Gemeinsamkeiten zwischen Funktions- und Stufentheorie sowie Verbindungen zu sehr viel avantgardistischeren, eher mathematisch formulierten zeitgenössischen Theorien (“Neo-Riemannism”), aufgedeckt werden konnte.

Literatur

[1] Markus Lepper, Baltasar Trancón y Widemann, and Michael Oehler. “funCode – Versatile Syntax and Semantics for Functional Harmonic Analysis Labels”. In: *Music & Science* (2022). https://doi.org/10.1177/20592043221085659.

[2] Markus Lepper, Baltasar Trancón y Widemann, and Michael Oehler. *funCode 1.0 – Technical Report*. Tech. rep. http://doi.org/10.48693/28. Osnabrück: Universität Osnabrück, 2022.

Visitor Optimization Revisited

Markus Lepper
semantics gGmbH, Berlin
post@mlepper.de

Baltasar Trancón y Widemann
Nordakademie Elmshorn

Zusammenfassung

Die von den Verfassern entwickelte und unterhaltene Programmsammlung “metatools” ist ein Werkzeugkasten für Compiler- und Programmentwicklung basierend auf Java und XML. [6] [1] [2]

Das darin enthaltene “umod” geneneriert aus sehr kompakten mathematischen Beschreibungen von Modellobjekten und ihren Assoziationen den Java Quelltext der entsprechenden Klassen. Der Faktor an Quelltextzeilen ist 1:25 bis 1:40. [3]

Visitoren sind ein verbreitetes und sehr komfortables Entwurfsmuster, dass z.B. auf einfache Weise Verfeinerungen und Spezilisierungen von Verhaltensweisen erlaubt, sowie weitgehende Robustheit von Analyse- und Veränderungsprogrammen gegenüber Veränderungen der Modelldefinition. Umod generiert den infrastrukturellen Code für solche Visitoren, gesteuert durch knappe Annotationen im Modellquelltext.

Vor etlichen Jahren schon haben die Autoren ein Optimierungsalgorithmus erfunden, der alle überflüssige Traversierungen durch Visitoren abschneidet, also solche, die ihrerseits nur Infrastrukturcode erreichen können, aber keinen benutzerdefinierten, also den einzig semantikrelevanten. [5]

Ein jüngerer Bestandteil von metatools ist LLJava, eine Abbildung des Bytecodes der Java Virtual Machine (JVM) in eine bequem zu handhabende, hinreichend abstrakte Modellstruktur. Das Subsystem LLJava-live erlaubt es, während des Laufes eines Programmes dynamisch Bytecode zu generieren, in die Klassenwelt zu laden und auszuführen, wobei dieser transparent auf alle momentan bekannten, vorcompilierten Definitionen zugreifen kann. [7] [8]

Anlässlich des Eelco Visser Commemorative Symposiums wurde diese neue Technik der Codegenerierung auf das ältere Visitoroptimierungsverfahren übertragen, und die alte Lösung (durch explizites Abfragen von Bitsets, also datengesteuert) durch das dynamische Überschreiben mit dynamisch generiertem Bytecode ersetzt (also kontrollflussgesteuert). [4]

Literatur

[1] Edgar Jakumeit et al. “A survey and comparison of transformation tools based on the transformation tool contest”. In: *Science of Computer Programming* 85, Part A (2014). http://www.sciencedirect.com/science/article/pii/S0167642313002803[20231013], pp. 41–99. ISSN: 0167-6423.

[2] Markus Lepper and Baltasar Trancón y Widemann. “Solving the TTC 2011 Compiler Optimization Task with metatools”. In: *Proceedings Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011*. Ed. by Pieter Van Gorp, Steffen Mazanek, and Louis Rose. Vol. 74. Electronic Proceedings in Theoretical Computer Science. http://cgi.cse.unsw.edu.au/~rvg/eptcs/Published/TTC2011/Papers/35/arXiv.pdf[20231013]. Open Publishing Association, 2011, pp. 70–115.

[3] Markus Lepper and Baltasar Trancòn y Widemann. “Rewriting Object Models With Cycles and Nested Collections”. English. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by Tiziana Margaria and

Bernhard Steffen. Vol. 8802. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 445–460. ISBN: 978-3-662-45233-2.

[4] Markus Lepper and Baltasar Trancón y Widemann. “Visitor Optimization Revisited”. In: *Proceedings of Eelco Visser Commemorative Symposium (EVCS 2023)*. https://drops.dagstuhl.de/opus/portals/oasics/index.php?semnr=16272[20231013]. 2023.

[5] Markus Lepper and Baltasar Trancón y Widemann. “Optimization of Visitor Performance by Reflection-Based Analysis”. In: *Theory and Practice of Model Transformations, ICMT 2011*. Ed. by Jordi Cabot and Eelco Visser. Vol. 6707. LNCS. http://bandm.eu/metatools/docs/papers/icmt11_ml.pdf[20231013]. Organization. Berlin, New York, Heidelberg: Springer, June 2011, pp. 15–30.

[6] *The BandM Metatools Homepage*. http://www.bandm.eu/metatools/[20231013].

[7] Baltasar Trancón y Widemann and Markus Lepper. “LLJava: Minimalist Structured Programming on the Java Virtual Machine”. In: *Proc. Principles and Practices of Programming on the Java Platform (PPPJ 2016)*. http://bandm.eu/metatools/docs/papers/lljava-final.pdf[20231013]. ACM, 2016. ISBN: 978-1-4503-4135-6.

[8] Baltasar Trancón y Widemann and Markus Lepper. “LLJava Live at the Loop – A Case for Heteroiconic Staged Meta-Programming”. In: *MPLR — Managed Prorgramming Languages 2021*. http://markuslepper.eu/papers/hsmp.pdf[20231013]. 2021, pp. 113–126.

Idee einer Programmiersprache für verteilte Anwendungen

Thomas M. Prinz

Zusammenfassung

Service-Orientierung empfiehlt, eine Software in separate, unabhängige Komponenten (Services) zu zerlegen. Jeder Service sollte dabei in derjenigen Programmiersprache implementiert werden, welche am besten zur Umsetzung des Services geeignet ist. Es müssen jedoch Daten zwischen den verteilten Services ausgetauscht und somit die dazu notwendigen Datenmodelle und Schnittstellen in allen verwendeten Programmiersprachen umgesetzt werden. Dies führt zu unnötigen Abhängigkeiten zwischen dem Datenmodell und seinen Implementierungen sowie zu einem erhöhten Entwicklungsaufwand, welche die Vorteile der Softwarezerlegung mindern — insbesondere für kleine Entwicklungsteams.
Diese Arbeit beschreibt eine neue Idee, welche ein verteiltes Programm wie ein nicht-verteiltes Programm behandelt. Zu diesem Zweck wird eine Meta-Netzwerkprogrammiersprache zur Beschreibung benötigt. Ein solches Programm wird von einem Übersetzer in verschiedene Services transformiert. Die Services sind in verschiedenen Programmiersprachen und Technologiestapeln umgesetzt und auf einer sogenannten Netzwerkmaschine lauffähig. Der Übersetzer und die Netzwerkmaschine verbergen dabei die Komplexität, die während der Entwicklung verteilter Systeme aufkommt. Als Ergebnis sollte diese Idee den Mehraufwand für die Entwicklung von allgemeiner, verteilter Software reduzieren.

1 Einleitung

Die Entwicklung und Umsetzung von Softwareprogrammen sind komplexe Prozesse. Um mit dieser Komplexität umzugehen, haben Wissenschaftler und Praktiker verschiedene Strategien entwickelt. Eine dieser Strategien ist es, die Software in verschiedene Module zu unterteilen, wobei jedes Modul separat von den anderen Modulen entwickelt werden kann. Um den Aufwand während der Umsetzung eines Moduls zu reduzieren, sollte jedes Modul in derjenigen Programmiersprache und dem Technologiestapel (Tool Stack) implementiert werden, welche am besten zur Lösung geeignet sind. Aufstrebende Technologien, wie die Service-Orientierung, machen dies möglich. Sind Services in ihrer Funktionalität in sich abgeschlossen (atomar), so werden sie Microservices genannt. Microservices kommunizieren über ein Netzwerk, werden unabhängig von anderen Services in Betrieb genommen und nutzen genau die Programmiersprache und den Technologiestapel, die am besten geeignet sind. Wird zum Beispiel eine Software entwickelt, welche mit Objekten aber auch mit maschinellem Lernen arbeitet, so könnten Teile dieser Software in Java, einem typischen Beispiel einer Objektorientierten Programmiersprache, und in Python, bekannt für seine datenwissenschaftlichen Pakete, umgesetzt werden.

Wenn Software in verschiedene, unterschiedliche Teilsysteme untergliedert wird, müssen diese Teilsysteme miteinander kommunizieren, um das Ziel bzw. den Anwendungszweck der Software zu erfüllen. Aus diesem Grund benötigen die Systeme Kommunikationsschnittstellen. Diese Schnitt-

stellen werden in der nicht-verteilten Softwareentwicklung üblicherweise durch Funktionsdeklarationen realisiert. Wenn verschiedene Programmiersprachen bzw. verteilte Systeme verwendet werden, sind jedoch nicht alle Funktionen in einer Ausführungsumgebung; sie könnten überall im Netzwerk oder auf der physischen Maschine sein und müssen mittels Netzwerkaufrufen oder anderen Ansätzen aufgerufen werden.

Der große Unterschied zwischen Aufrufen innerhalb einer Programmiersprache oder außerhalb mittels Services liegt hauptsächlich in deren Unterstützung [6]. In der Programmierung mit einer Sprache ist der Großteil des Programmcodes während der Übersetzungszeit und somit auch während der Entwicklung bekannt. Integrierte Entwicklungsumgebungen (Integrated Development Environments, IDEs) können diese Informationen nutzen und die Entwicklung unterstützen. So identifizieren sie unter anderem Aufrufe zu undefinierten Methoden, Zugriffsverletzungen und inkorrekt platzierte Methodenparameter. In solchen Fällen gibt die IDE unverzüglich Rückmeldung während der Programmierung und bietet in der Regel auch eine Fehlerbehandlung an. Ein weiterer Vorteil einsprachiger Programmierung liegt darin, dass Übersetzer diese Programme direkt in (virtuellen) Maschinencode übersetzen und — in den meisten Fällen und wenn die Ausführungsumgebung definiert ist — dieser auch direkt ausführbar ist. Im Gegensatz dazu ist die IDE-Unterstützung diesbezüglich für Servicebasierte und verteilte Softwarearchitekturen nicht stark ausgeprägt. Ein Grund dafür könnte sein, dass die IDE nur einen Ausschnitt der Software (den Service) und nicht die komplette Architektur kennt. Demzufolge müssen Entwickler jeden Service kontrollieren, ob er auch vordefinierten Spezifikationen genügt. Ohne diese Qualitätsüberprüfung können die Services nicht ordnungsgemäß aufgerufen werden, selbst wenn sie korrekt implementiert wurden.

Neben den beschriebenen Nachteilen in der IDE-Unterstützung hat der (Micro)-Service-Ansatz noch weitere Nachteile, welche in einem erhöhten Entwicklungsaufwand resultieren. Das zwischen den Services geteilte Datenmodell muss in allen Programmiersprachen Implementierung finden, in denen sie verwendet werden. Dies führt zu erhöhten und unnötigen Programmierkosten. Zusätzlich benötigen Objekte, welche zwischen Services ausgetauscht werden, Serialisierer und Deserialisierer. Außerdem müssen für jeden Service die Schnittstellen (Application Programming Interfaces, APIs) und die Aufrufe zu diesen Umsetzung finden. Apel et al. [2] haben in einem praktischen Projekt gezeigt, dass das Verhältnis zwischen funktionellem und organisatorischem Code teils bei $1:3$ liegt, d. h., für 100 Zeilen funktionellen Codes werden 300 Zeilen organisatorischen Codes benötigt. *Im Gesamten führt der aktuelle Ansatz zu einem erhöhten Arbeitsaufwand während der Programmierung, Entwicklung, Spezifikation und Wartung und demzufolge auch zu erhöhten Kosten und erhöhtem Zeitaufwand.* Dies ist insbesondere entscheidend für kleine Entwicklungsteams.

Abbildung 1 (linke Seite) illustriert den aktuellen Ansatz verteilter Anwendungsentwicklung. Entwickler definieren, implementieren und liefern dabei Services separat. Diese Services werden schlussendlich auf unterschiedlichen virtuellen Maschinen ausgeführt und realisieren die Anwendung.

Auf der rechten Seite von Abbildung 1 ist eine Veranschaulichung unseres neuen Ansatzes zu sehen. Dieser nutzt die Idee eines Übersetzers für verteilte und Service-orientierte Software. Zunächst reduziert dieser Ansatz allgemeingültige Eigenschaften verteilter, Multiprozessor- und Service-orientierter Systeme zu einer einzigen, abstrakten Basis, welche wir *Netzwerkmaschine* nennen. Eine Netzwerkmaschine verbirgt die Komplexität des Netzwerks von (virtuellen) Maschinen und vereinfacht so die Vorstellung einer einzigen Ausführungsumgebung während der Entwicklung. Zusätzlich organisiert die Netzwerkmaschine den Austausch von Daten, Nachrichten und Aufrufen verteilter Funktionalität (z. B. Aufrufe von Services). Demzufolge können sich Entwickler vorstellen, dass deren Programme auf einer einzigen Maschine laufen. Wissenschaftler argumentieren, dass dies die Softwareentwicklung vereinfacht und die Qualität des Programmcodes verbessert [14]. Eine Netzwerkmaschine als Ausführungsumgebung führt sogenannte *Netzwerkprogramme* aus. Netzwerkprogramme beschreiben die Kombination aller verteilter Systeme einer verteilten Software, verbergen dabei jedoch die Komplexität der Verteilung bei gewohnter Verwendung von Daten und Funktionsaufrufen. Der Vorteil eines Netzwerkprogramms liegt darin, dass verschiedene Funktionen der Software in verschiedenen Programmiersprachen umgesetzt werden. Es ist die Aufgabe des *Netzwerkübersetzers*, das Programm in verschiedene Module zu zerlegen und jedes Modul inklusive eigener Ausführungsumgebung zu übersetzen.

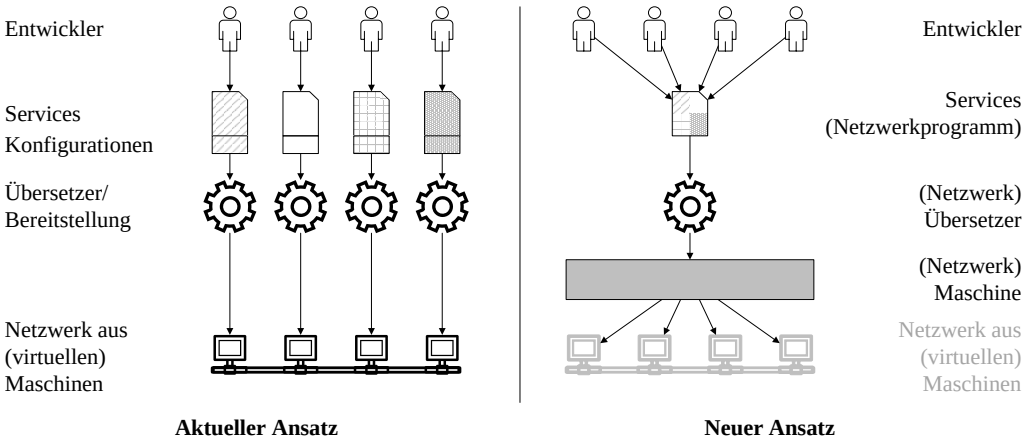


Abbildung 1: Der aktuelle Ansatz zur Entwicklung verteilter und Service-orientierter Software (links) und unser neuer Ansatz (rechts).

Ein weiterer Vorteil unseres Ansatzes liegt darin, dass IDEs Entwickler über Service- und Sprachgrenzen hinaus unterstützen können. Wenn gute Algorithmen gefunden werden, könnten Programme automatisch derart in verschiedene Teilsysteme unterschiedlicher Sprache zerlegt werden, dass die Performanz der Software insgesamt gesteigert wird. Zusammenfassend können die Programmierung, Entwicklung und Spezifikation im Gegensatz zu konventionellen Ansätzen Vereinfachung und Beschleunigung finden. Zeitgleich wird der Einsatz von verschiedenen Programmiersprachen auch im nicht-verteilten Kontext gefördert.

Diese Arbeit ist, wie folgt, strukturiert: Abschnitt 2 betrachtet verwandte Arbeiten in verteilten Systemen und deren Implementierung. Die Abschnitte 3 und 4 beschreiben den neuen Ansatz und die Idee von Netzwerkmaschinen und Netzwerkprogrammen. Anschließend erklärt Abschnitt 5 eine mögliche Übersetzerarchitektur zur Übersetzung von Netzwerkprogrammen. Abschnitt 6 beendet und diskutiert diese Arbeit.

2 Verwandte Arbeiten

Die Idee, eine Software in einer einheitlichen Sprache zu definieren, ist nicht neu und kam bereits zeitig in der Forschung und Industrie auf. Aus diesem Grund entwickelten sich in den letzten Jahren viele Sprachen für Datenmodelle und Funktionalität. Die *Unified Modeling Language* (UML) [18] ist ein Beispiel einer solchen einheitlichen Sprache, wovon einige Notationen von großen Teilen der Softwareentwickler als Modellierungsstandard verwendet werden [21] (insbesondere Klassendiagramme, Use-Case- und Sequenzdiagramme [7]). Die *Action Language for Foundational UML* (Alf) [17] und die *Semantics of a Foundational Subset for Executable UML Models* (fUML) [19] sind Standards, um vollständige Systeme in UML bis hinunter zu individuellen Funktionen zu beschreiben. Jedoch ist die Anwendung von Alf und fUML nicht trivial und benötigt wissenschaftliches Hintergrundwissen [4]. Der in dieser Arbeit vorgestellte Ansatz soll im Gegensatz dazu ähnlich der klassischen Softwareentwicklung und somit einfacher zu erlernen sein.

Alternative Ansätze versuchen den Aufwand zwischen den Softwareentwicklungsphasen Design und Implementierung zu minimieren. *Swagger* [29] erlaubt beispielsweise die Spezifikation von REST APIs mittels einer *OpenAPI* Spezifikation [28], woraus dann Codeskelette für die Client- und Serveranwendungen generiert werden. *JHipster* [25] erzeugt vollständige Anwendungen innerhalb eines vordefinierten Technologiestapels. Sowohl Swagger als auch JHipster haben ihre Grenze in der Beschreibung des Verhaltens der Anwendung. Das Framework *Flutter* [13] beschreibt Anwendungen in der Sprache *Dart* und übersetzt diese in native plattformübergreifende Systeme. Flutter kann jedoch einzelne Teile der Anwendung nicht in individuelle Zielsprachen und Technologiestapel kompilieren. Einige Web-Templates und Verhaltensbeschreibungssprachen, wie beispielsweise *pug* [24], *Haml* [3] und *GWT* [30], erlauben die Beschreibung von Software und deren Übersetzung in Webanwendungen. Diese sind für bestimmte Anwendungszwecke mächtig, jedoch limitiert auf Webanwendungen, JavaScript und Java.

OpenMP [20] hat Ähnlichkeiten zum Ansatz in dieser Arbeit, auch wenn es auf C++, C und Fortran eingeschränkt ist. OpenMP ist ein Standard im Hochperformanzrechnen (High Performance Computing, HPC) geworden, um Systeme mit verteiltem Speicher zu programmieren. Es erlaubt hochsprachliche, parallele Programmierung sowie die Portierung auf andere Netzwerkstrukturen [32]. Nach der Kompilierung verschiebt die OpenMP-Umgebung automatisch verschiedene Maschinencoderepräsentationen des Programms auf die Prozessoren und verwaltet den Datenaustausch zwischen diesen. Unsere Idee, die Details über die Infrastruktur zu verbergen, ist ähnlich. Jedoch liegt unser Fokus auf der Programmierung von Software für den generellen Gebrauch.

Die Spieleentwicklungsumgebung *Unity* [31] verfolgt ebenfalls einen ähnlichen Ansatz wie der hier vorgestellte. Sie erlaubt es, Spiele in einer Teilsprache von C# zu beschreiben und dann auf dem Open-Source .NET-Framework Mono [16] ausführen zu lassen. Mono basiert auf dem *Common Language Infrastructure* (CLI) Standard [9], der es erlaubt, ähnlich wie die *Java Virtual Machine* (JVM) [15], Programme auf unterschiedlichen Betriebssystemen auszuführen. Die Verwendung von C# als de-facto Meta-Programmiersprache folgt unserem Ansatz, Datenmodelle und Services unabhängig der Programmiersprache zu implementieren. Anstatt die Modelle und Services in eine einzige Sprache und Laufzeitumgebung zu übersetzen, ist es jedoch das Ziel dieser Arbeit, diese über verschiedene Sprachen und Umgebungen hinweg zu verteilen.

Ein anderer Ansatz in der Entwicklung von Spielen ist die *GameEngine* von Apel [1] für die Beschreibung von *Massively Multiplayer Online Games* (MMOG). Sein Ansatz umfasst die automatische Generierung von Code während der Laufzeit für die Kommunikation, die Schnittstellen zwischen Visualisierung und Funktionalität (Controller), usw. Dieser Ansatz führt dazu, dass Spiele leichtgewichtiger in Hinblick auf die Anzahl der Codezeilen und die Länge der Implementierungszeiten ausfallen. Auch wenn viele seiner Konzepte adaptiert werden können, macht der starke Fokus auf MMOGs den Ansatz schwierig für die allgemeine Softwareentwicklung.

Anstatt die Entwicklung von Software mit Hilfe einer speziellen Ausführungsumgebung zu verbessern (wie in einigen der vorangegangenen Beispiele), verbirgt der *Web Computer* von Kozlovics [14] Komplexität während der Entwicklung und Ausführung in einem eigenen Betriebssystem. Dieser Ansatz adressiert ähnliche Probleme in der Softwareentwicklung wie diese Arbeit: eine Fokussierung auf die Annahme, dass eine Software für einen einzigen Computer, einen einzigen Nutzer und als einzig ausgeführtes Programm entwickelt wird. Kozlovics' Ansatz erscheint vielversprechend, jedoch ist es noch unklar, wie genau Anwendungen für den Web Computer entwickelt und übersetzt werden sollen. Außerdem ist er limitiert auf Webanwendungen.

Neben Arbeiten in der Softwareentwicklung gibt es auch einige wenige Arbeiten von Singer et al. [12, 27] und Prinz et al. [22, 23], welche untersuchen, wie (Micro)Servicesysteme in Form von Geschäftsprozessen übersetzt werden können. Jedoch besitzen beide Arbeitsgruppen einen Hintergrund im Geschäftsprozessmanagement, wobei heutige Modellierungssprachen oftmals nicht in der Lage sind, die volle Funktionalität von allgemeiner Anwendungssoftware abzubilden.

3 Netzwerkmaschinen

Geschäftsprozesse, HPC, verteilte und Service-orientierte Software haben ihre Unterschiede in der Umsetzung und Ausführung. Sie besitzen jedoch auch starke Ähnlichkeiten: Jeder Multiprozessor eines Multiprozessorsystems aus dem HPC-Bereich kann eine andere Funktion mit einer anderen Instruktionsmenge als die anderen Maschinen ausführen; in Service-orientierten Architekturen bilden Services die Schnittstellen zu Funktionen, wobei die Services in verschiedenen Programmiersprachen umgesetzt sind, auf verschiedenen Maschinen laufen und ihre eigenen geteilten und verteilten Speicher besitzen. Von einem abstrakten Blickwinkel betrachtet, werden all diese verteilten Systeme in einem Netzwerk aus möglicherweise unterschiedlichen (virtuellen) Maschinen ausgeführt, wobei alle Maschinen wiederum ihre eigenen unterstützten Programmiersprachen und Technologiestapel aufweisen können. Aufgrund der starken Ähnlichkeiten zwischen diesen ist die Idee, diese zu einer gemeinsamen Basis von verteilter Software zu kombinieren, welche wir *Netzwerkmaschine* nennen.

Eine Netzwerkmaschine ist eine leichtgewichtige Laufzeitumgebung für verteilte Software. Aus Sicht des Entwicklers führt eine Netzwerkmaschine ein *Netzwerkprogramm* aus (welche detaillierter im nächsten Abschnitt 4 eingeführt werden). Netzwerkprogramme werden in verschie-

dene *Container* übersetzt, die mehrere Services beinhalten können. Jedoch entspringen alle Services innerhalb eines Containers der selben Programmiersprache, z. B. könnte es Container für Java-, R-, C- und Python-Services geben. Eine Technologie für Container könnte Docker [8] sein. Eine Netzwerkmaschine lädt solche Container und macht deren Services erreichbar. Ab diesem Moment läuft die Software und sie kann verwendet werden. Da die Netzwerkmaschine alle Services kennt, kann sie Serviceaufrufe von innerhalb und außerhalb auflösen und den korrekten Service ansprechen. Abbildung 2 illustriert die konzeptionelle Idee einer Netzwerkmaschine.

Aus einem technischen Blickwinkel verbirgt eine Netzwerkmaschine die Komplexität einer Menge verschiedener, möglicherweise virtueller, Maschinen (*Geräte*), welche miteinander über ein Netzwerk verbunden sind (wie in Abbildung 2 zu sehen ist). Das Netzwerk dient als *Bus* für den Austausch von Daten und Nachrichten zwischen den Geräten. Jedes Gerät beherbergt mindestens einen Container und bietet die Services des Containers an. Da die Netzwerkmaschine das Netzwerk kennt, kann es jedem Service eine spezifische Adresse zuweisen (wie eine Speicheradresse auf einem gewöhnlichen Rechner) und diese in einem *Adressregister* speichern. Alle Services werden über eine *allgemeine Schnittstelle* angesteuert, welche den Dateneingang und -ausgang behandelt. Diese Schnittstelle ist vergleichbar zu der Behandlung von Funktionen in Maschinensprachen: Sie deserialisiert jeden Eingabeparameter und gibt den Wert in den Servicekontext, so dass der Service auf diesen zugreifen kann. Anschließend ruft die Schnittstelle die Servicefunktionalität auf. Die Ergebnisse des Serviceaufrufs werden von der allgemeinen Schnittstelle serialisiert und zum Serviceanrufer zurückgesendet.

Die Funktionalität des Services wird in einer eigenen (möglicherweise virtuellen) Umgebung ausgeführt. Dies sind tatsächliche Laufzeitumgebungen für Programmiersprachen, welche in den Services genutzt werden, wie die JVM [15] oder Python- und R-Laufzeitumgebungen. Da eine Netzwerkmaschine letztlich auch eine Ausführungsumgebung ist, kann sich hinter einem Service wiederum eine komplette Netzwerkmaschine verstecken. Zusätzlich besitzt jeder Service einen eigenen lokalen Speicher und teilt seine Daten nur über die Schnittstellen. Globale Daten sollten, wie in der Programmierung üblich, mit Vorsicht verwendet werden. Netzwerkmaschinen bieten globale Daten über eine *Persistenzschnittstelle* an, die für alle Datenstrukturen zur Verfügung steht, welche im Netzwerkprogramm Definition und zwischen Services Austausch finden. Diese Persistenzschnittstelle erlaubt eine einfache und durchgehende Speicherung von Daten, deren Zugriff, Modifizierung und Löschung. Solch ein globales Gedächtnis ist erfahrungsgemäß natürlich ineffizienter als temporäre Daten innerhalb der Services. Da Services im Falle von asynchronen Serviceaufrufen parallel ausgeführt werden können, muss die Persistenzschnittstelle Wettkampfbedingungen von parallelen Zugriffen auflösen, bspw. über Verriegelungsmechanismen.

Aufrufe von Services können synchron (wie in vielen Programmiersprachen) oder asynchron durchgeführt werden. Asynchrone Aufrufe könnten den *Promise*-Ansatz verwenden. Das heißt, der Aufruf liefert unmittelbar ein Versprechen (Promise) zurück. Das Promise besitzt eine Referenz auf eine Funktion, wenn das Versprechen erfüllt ist und eine andere Referenz auf eine Funktion, falls das Versprechen nicht erfüllt wurde. Der Vorteil eines Promises liegt darin, dass das resultierende Programm sequentiell wirkt und somit einfacher zu verstehen ist.

4 Netzwerkprogramme

Die Programmierung von verteilter Software ist eine Herausforderung, denn der Entwickler kann nicht einfach eine Funktion in einem Programm in einer anderen Programmiersprache aufrufen, sondern muss einen externen Service mit den richtigen Parametern, im korrekten Format, usw. ansteuern. Aus unserer Sicht und der Sicht einiger Wissenschaftler [14] wäre es hilfreich, sich während der Entwicklung vorstellen zu können, dass alle Funktionen (interne und externe) in der gleichen Umgebung vorliegen, egal wo sie tatsächlich lokalisiert sind. Der Ansatz der Netzwerkprogramme schafft diese Situation. Netzwerkprogramme werden in den vorher definierten Netzwerkmaschinen ausgeführt und werden in einer *Netzwerkprogrammiersprache* beschrieben. Diese beschreibt hauptsächlich Datenmodelle und strukturiert die Software als Services. Die Services wiederum werden direkt in den verschiedenen Zielsprachen implementiert. Die Vorteile eines solchen Netzwerkprogramms sind somit: (1) Sie definieren die Schnittstellen aller Services, (2) Serviceaufrufe sind analog zu Funktionen-/Methodenaufrufen und (3) sie beschreiben ausgetauschte Datenmodelle lediglich einmal.

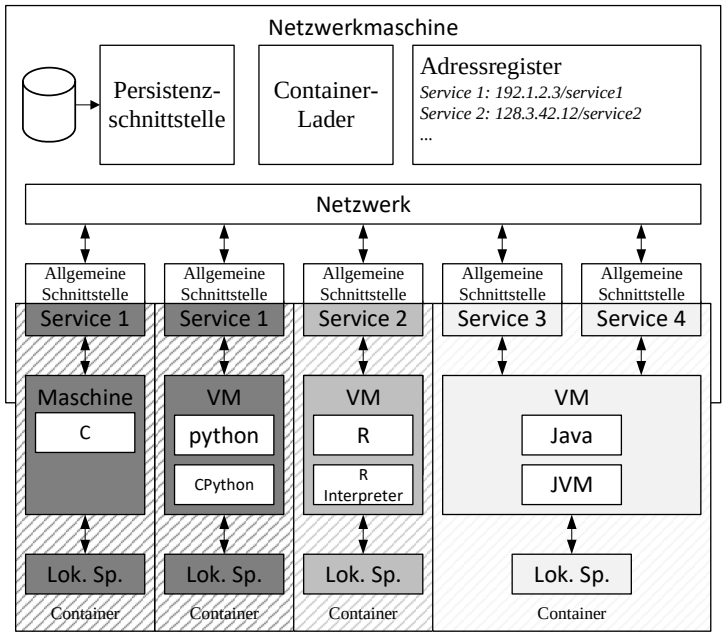


Abbildung 2: Eine Netzwerkmaschine mit (virtuellen) Maschinen, genutzten Programmiersprachen und Ausführungsumgebungen mit lokalen und verteilten Speichern und Containern, welche die Services definieren. Die Netzwerkmaschine besitzt einen Lader für die Container, eine Persistenzschnittstelle, allgemeine Schnittstellen für Services und ein Adressregister, in dem alle Netzwerkadressen der Services gespeichert werden.

Abbildung 3 zeigt ein Beispiel eines kleinen Netzwerkprogramms. Das Beispiel nutzt als Netzwerkprogrammiersprache eine Java-ähnliche Syntax. Eine andere Syntax ist natürlich auch möglich. Das Programm definiert zu Beginn das Datenmodell (die Klasse) `Pair` mit zwei Feldern `a` und `b` sowie einen Konstruktor. Es definiert außerdem die Klasse `Computation`, die zwei Methoden beinhaltet. Die Methode `handlePairs` ist in Java implementiert und übersetzt eine verschachtelte Reihung von Integer-Paaren in eine Liste von Objekten vom Typ `Pair`. Zum Schluss gibt die Methode die Summe der Liste von Paaren zurück, in dem sie die `computeSums`-Methode aufruft, welche in R geschrieben ist. Die Methode `computeSums` erfordert eine Reihung (oder Liste) von `Pair`-Objekten. Sie berechnet anschließend die Summe der beiden Felder `a` und `b` jedes `Pair`-Objekts und gibt diese als Vektor (Reihung) zurück.

```
1  class Pair {
2    public int a, b;
3    Pair(int a, int b) {
4      this.a = a; this.b = b;
5    }
6  }
7  class Computation {
8    @Java
9    public int[] handlePairs(int[][] pairs) {
10     Pair[] pairList = new Pair[pairs.length];
11     for (int i = 0; i < pairs.length; i++) {
12       int a = pairs[i][0], b = pairs[i][1];
13       pairList[i] = new Pair(a, b);
14     }
15     return this.computeSums(pairList);
16   }
17   @R
18   public int[] computeSums(Pair[] pairs) {
19     sapply(pairs, function(pair) {
20       pair$a + pair$b
21     })
22   }
23 }
```

Abbildung 3: Ein Beispiel eines Netzwerkprogramms.

Die Implementierung des Beispielprogramms aus Abbildung 3 mit dem aktuellen Stand der Technik ist keine Herausforderung, jedoch zeitaufwändig. Zunächst muss die Datenstruktur `Pair` in Java und R implementiert werden. Danach wird der Java-Anteil des Programms umgesetzt und als Service zugänglich gemacht, beispielsweise mit REST [11]. Zusätzlich müssen die Entwickler den R-Anteil des Programms umsetzen und ebenfalls als Service zugänglich machen. Da die Services Daten untereinander austauschen, müssen die Entwickler ein Austauschprotokoll definieren oder ein existierendes, wie JSON [10], nutzen. Weiterhin müssen die Dateninformationen den korrekten Parametern zugewiesen werden.

Der Mehraufwand in der Umsetzung kann Entwickler dazu veranlassen, das komplette Pro-

gramm in einer einzigen Sprache zu implementieren anstelle in mehreren. Manche Probleme finden jedoch in bestimmten Programmiersprachen einfachere und effizientere Lösungen, Berechnungen, Implementierungen und Ausführungen. Dieser Vorteil entfällt dann und führt wiederum auch zu steigenden Implementierungskosten [2]. Netzwerkprogramme sollten diese Situation verbessern, da Entwickler die oben genannten Schritte nicht durchführen müssen.

5 Compilerarchitektur für Netzwerkprogramme

Innerhalb der hier vorgestellten Idee werden Netzwerkprogramme in eine lauffähige Software mit Hilfe eines Übersetzers (Compilers) überführt. Die Architektur eines solchen Compilers kann grundsätzlich der selben Struktur folgen wie für einen klassischen Compiler (siehe z. B. Cooper and Torczon [5]). Diese besteht aus einem *Frontend*, welche das Quellprogramm einliest und in eine interne *Zwischencoderepräsentation* (ZR) überführt; und einem *Backend*, dass die ZR in eine Ausgabesprache übersetzt. Auch wenn sogenannte *Cross*-Compiler Ausgabecode für verschiedene Ausgabesprachen erzeugen können, produzieren sie jedoch den kompletten Ausgabecode in einer einzigen Sprache. Die vorgestellte Übersetzerarchitektur setzt sich über diese Limitierung hinweg und erlaubt die Zerlegung des Quellprogramms in kleinere Teilprogramme, wobei jedes Programm in eine andere Sprache übersetzt werden kann. Des Weiteren generiert der Compiler zusätzlichen Code, damit die einzelnen Teilprogramme miteinander kommunizieren können (im Sinne von Methoden-/Funktionsaufrufen). Die Zerlegung des Quellprogramms ist Teil der Optimierungsalgorithmen des Compilers.

Abbildung 4 zeigt die verschiedenen Phasen der angestrebten Compilerarchitektur für Netzwerkprogramme. Das Frontend des Compilers ist das eines klassischen Übersetzers. Zunächst produziert ein *Scanner* einen Zeichenstrom aus dem Quellprogramm. Dieser Zeichenstrom wird vom *Parser* verwendet, um die Grammatik des Quellprogramms zu kontrollieren und den Zeichenstrom in einen *abstrakten Syntaxbaum* (ASB) zu überführen. Die *semantische Analyse* beinhaltet verschiedene Algorithmen, um Fehler im Programm so schnell wie möglich zu finden; unter anderem (statische) Typfehler, Zugriffsfehler und undefinierte Variablen. Der Übersetzer überführt anschließend den kontrollierten ASB mit Hilfe eines *Transformierers* in eine ZR. Auf die ZR lassen sich dann *Optimierungen* leichter anwenden. In dieser Phase des Compilers sind alle Optimierungen unabhängig von den Zielsprachen.

An dieser Stelle ist es wichtig zu erwähnen, dass das Frontend hauptsächlich jene Programmteile des Netzwerkprogramms behandelt, welche in der Netzwerkprogrammiersprache notiert vorliegen. Quellcode, welcher bereits in der Zielsprache vorliegt, wird vom Frontend nicht überprüft. Für diesen wird später im Backend ein sprachspezifischer Compiler verwendet, dem der vollständige Kontext zur Verfügung gestellt wird (z. B. Datenmodelle und Methodensignaturen).

Nach dem Frontend besitzt die Compilerarchitektur eine *Zerlegungsphase*. Dabei versucht der Übersetzer die ZR in eine verteilte ZR zu zerlegen, wobei jeder verteilte Programmteil der ZR in eine andere Zielsprache und auf eine andere Maschine übersetzt werden kann. Dabei nimmt der Compiler den bereits in der Zielsprache vorliegenden Quellcode und fügt gegebenenfalls Übersetzungen des allgemeinen Datenmodells hinzu. Nehmen wir dazu das Netzwerkprogramm aus Abbildung 3 als Beispiel, welches die Klasse `Pair` abstrakt beschreibt. Da diese Klasse sowohl in den Java- als auch in den R-Programmteilen Verwendung findet, muss diese Klasse in beiden Sprachen zur Verfügung stehen. Aus diesem Grund muss der Compiler die abstrakte Klassendefinition der Netzwerksprache in konkrete Definitionen der Programmiersprachen Java und R überführen.

Da die Zerlegung des Codes zu einer komplexeren Softwarearchitektur führt, sollte die resultierende Architektur für die Wartung gut dokumentiert werden. Aus diesem Grund sollte der Compiler über einen *Dokumentationsgenerator* verfügen, welcher sowohl eine Architekturbeschreibung als auch Beschreibungen über die Datenmodelle generiert (z. B. in Form eines beschriebenen Klassendiagramms).

Nach der Zerlegung der ZR und der Zuweisung von Zielsprachen für jeden verteilten Programmteil erweitert der Compiler die verschiedenen Teile des Programms mit zusätzlichem Code in der *Erweiterung für Serviceaufrufe*. Diese Erweiterung überführt jeden Programmteil in einen Service. In anderen Worten werden die Programmteile derart modifiziert, dass sie von anderen Programmteilen mit Hilfe der allgemeinen Schnittstelle der Netzwerkmaschine verwendet werden kön-

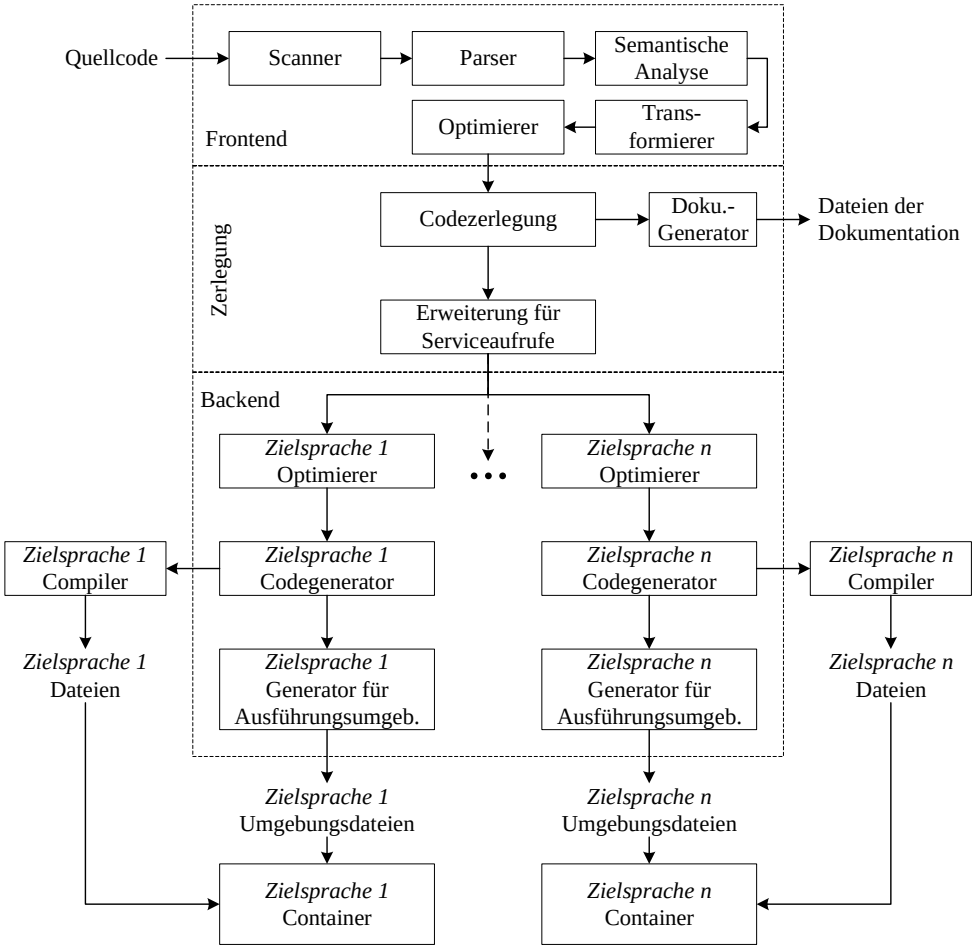


Abbildung 4: Die angestrebte Compilerarchitektur für Netzwerkprogramme.

nen. Aus diesem Grund muss die Erweiterung auch Code für die Serialisierung und Deserialisierung von Dateninformationen zur Verfügung stellen. Als Beispiel sei hier die Möglichkeit genannt, RESTful Services [11] zu verwenden und Dateninformationen mit JSON [10] oder CSV [26] zu übertragen.

Im Anschluss an die Erweiterungsphase übersetzt das Backend alle Programmteile Schritt für Schritt in die zugewiesenen Zielsprachen. Da diese Übersetzung sprachabhängig ist, können durch den Compiler speziellere Optimierungen in einem weiteren *Optimierer* zur Anwendung kommen. Die Übersetzung der Programmteile in die Zielsprache findet außerhalb des Compilers mit Hilfe bereits existierender Übersetzer für die Zielsprachen statt.

Viele moderne Zielsprachen, wie Java, R, Python und JavaScript, benötigen spezielle Umgebungen, um ausgeführt werden zu können. Glücklicherweise gibt es Werkzeuge, um solche Umgebungen zu erzeugen (z. B. Docker [8]). Es sollte das Ziel des Compilers sein, für jeden Programmteil einen eigenen ausführbaren Container mit der korrekten Ausführungsumgebung und den korrekten Services zu erzeugen. Dies geschieht im *Generator für Ausführungsumgebungen*. Als Ergebnis produziert der Übersetzer ein vollständig ausführbares Netzwerkprogramm.

6 Schlussworte und Diskussion

Diese Arbeit präsentierte eine neue Idee zur Entwicklung von verteilten Programmen. Diese Idee erlaubt es ein Programm, welches aus mehreren Teilen in möglicherweise verschiedenen Programmiersprachen besteht, derart zu formulieren, als wäre es nicht verteilt. Die Verteilung übernimmt dann ein Compiler, der das Programm in verschiedene Teile in verschiedenen Sprachen zerlegt. Um dies zu erreichen, muss das Programm in einer sogenannten Netzwerkprogrammiersprache

vorliegen. Ein Netzwerkprogramm definiert dabei Datenmodelle und Services, wobei die Funktionalität eines jeden Services in einer eigenen Programmiersprache vorliegen kann, die vom Entwickler ausgewählt wird. Es kann somit genau die Sprache verwendet werden, welche die Funktionalität des Services am einfachsten und effizientesten umsetzt. Netzwerkprogramme werden auf einer Netzwerkmaschine ausgeführt, die wiederum die Architektur von verteilten und Multiprozessorsystemen abstrahiert und verallgemeinert. Dabei beschreibt diese Arbeit nur die Idee; eine Implementierung existiert jedoch bisher noch nicht.

Der Ansatz, verteilte Programme mit Hilfe eines Übersetzers zu behandeln, verallgemeinert bekannte Techniken aus dem Compilerbau. Existierende Programmiersprachen werden dabei als Zielsprachen des Übersetzers anstelle einer Maschinensprache verwendet. Wie der Abschnitt über verwandte Arbeiten zeigt, werden bereits ähnliche Methoden während der Übersetzung für Multiprozessorsysteme und Computerspiele angewandt. Diese Konzepte basieren aber auf speziellen Anwendungsdomänen und sind nur begrenzt für die Verwendung in der allgemeinen Softwareentwicklung geeignet. Wenn demnach eine gute Lösung gefunden wird, ein beliebiges Programm automatisch in Teilprogramme verschiedener Sprachen zu zerlegen, könnte dies auch von Vorteil in anderen Domänen sein. Beispielsweise könnte solch ein Ansatz auch auf Programmcode einer Programmiersprache angewandt werden, für den dann eine IDE vorschlägt, welche Abschnitte des Codes zur Verbesserung der Codequalität und der Performanz in einer anderen Programmiersprache umgesetzt sein sollte. Die IDE kann anschließend auch diese Abschnitte automatisiert übersetzen. Eine Anwendung im Geschäftsprozessmanagement wäre auch denkbar, wenn Geschäftsprozesse in Netzwerkprogramme überführt werden würden. Wenn die Netzwerkprogrammiersprache Konzepte für Parallelisierung aufweist, wäre eine Anwendung selbst im HPC denkbar.

Wenn die vorgeschlagene Idee umgesetzt wird, sollten Vorteile verwandter Arbeiten und bewährte Verfahren in der verteilten Softwareentwicklung sorgsam Berücksichtigung finden. Zum Beispiel bietet das isolierte Entwickeln von Services auch Vorteile, in dem es den Fokus stärkt, saubere Schnittstellen zu definieren und unnötige Parameter und Verknüpfungen zu vermeiden. Insgesamt steigt dadurch die Wiederverwendbarkeit von Services. Auch kann eine solche isolierte Entwicklung einfacher auf mehrere Entwicklungsteams verteilt werden. Konflikte während der Entwicklung entstehen somit seltener. Dies sind grundsätzlich starke Vorteile der aktuellen Verfahren. Die in dieser Arbeit vorgestellte Idee kann diese aber aufgreifen, wenn sie während der Ausreifung der Idee in eine Implementierung in Betracht gezogen werden. Außerdem kommen einige dieser Vorteile nur in größeren jedoch selten in kleineren Entwicklungsteams zum Tragen. Des weiteren sind die Isolation von Funktionen und Services im Sinne der Tradition von hoher Kohäsion und geringer Kopplung überall in der Softwareentwicklung hilfreich und von Bedeutung.

Zukünftige Arbeiten umfassen die Definition einer ersten Grammatik für eine Netzwerkprogrammiersprache und die Erstellung eines ersten Prototypen eines Übersetzers. Glücklicherweise ist das Backend des Übersetzers leichtgewichtig, da alle bereits existierenden Compiler für Programmiersprachen verwendet werden können. Diese zukünftigen Arbeiten umfassen Herausforderungen, sollten jedoch in wenigen Jahren umsetzbar sein.

Danksagung

Besonderen Dank gilt Sebastian Apel für rege Anmerkungen und Diskussionen.

Literatur

[1] Sebastian Apel. „Reducing Development Overheads with a Generic and Model-Centric Architecture for Online Games“. In: *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*. IEEE Computer Society, 2018, S. 21–28.

[2] Sebastian Apel, Florian Hertrampf und Steffen Späthe. „Towards a Metrics-Based Software Quality Rating for a Microservice Architecture - Case Study for a Measurement and Processing Infrastructure“. In: *Innovations for Community Services - 19th International Conference, I4CS 2019, Wolfsburg, Germany, June 24-26, 2019, Proceedings*. Hrsg. von Karl-Heinz Lüke u. a. Bd. 1041. Communications in Computer and Information Science. Springer, 2019, S. 205–220.

[3] Hampton Catlin. *Haml*. Juni 2021. URL: http://haml.info/.

[4] Federico Ciccozzi, Ivano Malavolta und Bran Selic. „Execution of UML models: a systematic review of research and practice“. In: *Software & Systems Modeling* 18.3 (2019), S. 2313–2360.

[5] Keith D. Cooper und Linda Torczon. *Engineering a Compiler*. 2nd. USA: Morgan Kaufmann, 2011.

[6] Flavio De Paoli. „Challanges in Services Research: A Software Architecture Perspective“. In: *Advances in Service-Oriented and Cloud Computing*. Hrsg. von Alexander Lazovik und Stefan Schulte. Cham: Springer International Publishing, 2018, S. 219–227. ISBN: 978-3-319-72125-5.

[7] Brian Dobing und Jeffrey Parsons. „How UML is used“. In: *Communications of the ACM* 49.5 (2006), S. 109–113.

[8] Docker Inc. *Empowering App Development for Developers | Docker*. Juni 2021. URL: https://www.docker.com/.

[9] Ecma International. *Standard ECMA-335: Common Language Infrastructure (CLI) — 6th edition (June 2012)*. Version 6th. Ecma International, Juni 2012. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf.

[10] Ecma International. *Standard ECMA-404: The JSON Data Interchange Syntax — 2nd edition (December 2017)*. Version 2nd. Ecma International, Dez. 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[11] Roy T. Fielding und Richard N. Taylor. „Principled design of the modern Web architecture“. In: *ACM Trans. Internet Techn.* 2.2 (2002), S. 115–150.

[12] Matthias Geisriegler u. a. „Actor Based Business Process Modeling and Execution: A Reference Implementation Based on Ontology Models and Microservices“. In: *43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017, Vienna, Austria, August 30 - Sept. 1, 2017*. IEEE Computer Society, 2017, S. 359–362. ISBN: 978-1-5386-2141-7.

[13] Google. *Flutter - Beautiful native apps in record time*. Juni 2021. URL: https://flutter.dev/.

[14] Sergejs Kozlovics. „The Web Computer and Its Operating System: A New Approach for Creating Web Applications“. In: *Proceedings of the 15th International Conference on Web Information Systems and Technologies, WEBIST 2019, Vienna, Austria, September 18-20, 2019*. Hrsg. von Alessandro Bozzon, Francisco José Domìnguez Mayo und Joaquim Filipe. ScitePress, 2019, S. 46–57.

[15] Tim Lindholm u. a. *The Java Virtual Machine Specification, Java SE 8 Edition*. 8. Aufl. California, USA: Addison-Wesley Professional, 2014.

[16] Mono Project. *Mono — Cross platform, open source .NET framework*. Juni 2021. URL: https://www.mono-project.com/.

[17] Object Management Group. *Action Language for Foundational UML (Alf). Concrete Syntax for a UML Action Language, Version 1.1*. Object Management Group, Juli 2017. URL: https://www.omg.org/spec/ALF/1.1.

[18] Object Management Group. *OMG Unified Modeling Language — Version 2.5.1*. Object Management Group, Dez. 2017. URL: https://www.omg.org/spec/UML/2.5.1.

[19] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.4*. Object Management Group, Dez. 2018. URL: https://www.omg.org/spec/ALF/1.1.

[20] OpenMP Architecture Review Board. *OpenMP Application Programming Interface — Version 5.0*. Version 5.0. OpenMP Architecture Review Board, Nov. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[21] Flavio Oquendo, Jair Leite und Thais Batista. „Executing Software Architecture Descriptions with SysADL“. In: *Software Architecture*. Hrsg. von Bedir Tekinerdogan, Uwe Zdun und Ali Babar. Cham: Springer International Publishing, 2016, S. 129–137. ISBN: 978-3-319-48992-6.

[22] Thomas M. Prinz, Norbert Spieß und Wolfram Amme. „A First Step towards a Compiler for Business Processes“. In: *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Hrsg. von Albert Cohen. Bd. 8409. Lecture Notes in Computer Science. Springer, 2014, S. 238–243. ISBN: 978-3-642-54806-2.

[23] Thomas M. Prinz u. a. „Towards a Compiler for Business Processes - A Research Agenda“. In: *SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing, Nice, France, March 22–27, 2015. Proceedings*. Hrsg. von Marcelo de Barros und Claus-Peter Rückemann. 2015, S. 49–54.

[24] pug. *Getting Started - Pug*. Juni 2021. URL: https://pugjs.org/.

[25] Matt Raible. *The JHipster mini-book*. 5.0.1. USA: C4Media, 2018.

[26] Y. Shafranovich. *RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Version 2nd. The Internet Society, Okt. 2005. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[27] Robert Singer. „Business Process Modeling and Execution - A Compiler for Distributed Microservices“. In: *CoRR* abs/1601.05976 (2016). arXiv: 1601.05976.

[28] Swagger. *OpenAPI Specification*. Swagger, Juni 2021. URL: https://swagger.io/specification/.

[29] Swagger. *The Best APIs are Built with Swagger Tools*. Juni 2021. URL: https://www.swagger.io/.

[30] Adam Tacy u. a. *GWT in Action*. 2nd. Greenwich, CT, USA: Manning Publications Co., Feb. 2013. ISBN: 1935182846.

[31] Unity Technologies. *Unity — Game Engine*. Juni 2021. URL: https://unity.com/.

[32] Chenle Yu, Sara Royuela und Eduardo Quiñones. „OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices“. In: *SCOPES '20: 23rd International Workshop on Software and Compilers for Embedded Systems, St. Goar, Germany, May 25-26, 2020*. Hrsg. von Sander Stuijk und Henk Corporaal. ACM, 2020, S. 42–47.

Auswirkungen feingranularer Elaboration Order auf Programmiersprachen und Compilerarchitektur

Timm Felden
https: // github. com/ tyr-lang
timm.felden@felden.com

Abstract

Elaboration (Erarbeitung) bezeichnet die Verarbeitung von Deklarationen, kann aber auf andere Programmbestandteile, wie Typdefinitionen, Parameterdefinitionen oder Funktionsrümpfe ausgedehnt werden. Die Granularität kann erhöht werden, indem man beispielsweise Aspekte wie die Einführung eines Parameternamens, seines Typs oder ggf. eines Defaultwerts voneinander trennt. Im Wesentlichen gibt es drei Strategien, die Erarbeitungsreihenfolge zu regeln. Die Sprache kann so gestaltet werden, dass die Reihenfolge der Verarbeitung eines Passes nicht erheblich ist, indem Querbezüge zwischen Programmteilen verboten sind; beispielsweise bei der Namensanalyse globaler Definitionen. Ferner gibt es einen top-down Ansatz, der Rückwärtsbezüge gestattet, d.h. auf bereits verarbeite Programmteile – beispielsweise bei Namensanalyse in Funktionsrümpfen. Der dritte Ansatz ist eine bedarfsgetriebene Auswertung (Elaboration Order), bei der der Compiler bei der Verwendung eines noch nicht verarbeiteten Programmteils diesen automatisch zuerst auswertet – beispielsweise Pakete in Ada. In vielen Sprachen finden sich Mischformen der drei Strategien. Ferner ist das Konzept mit der Initialisierung von Werten zur Laufzeit verwandt, da es sich gewissermaßen um die Initialisierung der Definitionsrepräsentanten zur Compilezeit handelt.
Das Forschungsprojekt Tyr ist eine statisch typisierte, kompilierte und systemnahe Programmiersprache. Tyr verzichtet komplett auf explizite Deklarationen und verwendet eine feingranulare Elaboration Order, um Kompaktheit und Ausdrucksstärke zu erreichen. Der Vortrag wird die derzeit verwendete Strategie vorstellen und anhand von Beispielen die Interaktion zwischen Elaboration und anderen Konzepten wie Funktions-Inlining, Templates oder Overloadresolution zeigen. Ferner wird die Realisierung im Compiler skizziert und erläutert, wie sich das traditionelle Pipelinekonzept hier zwangsweise verändert.

Design Patterns for Name and Type Analysis with JastAdd

Uwe Meyer und Björn Lötters
Technische Hochschule Mittelhessen
University of Applied Sciences
Wiesenstraße 14, D–35390 Gießen
{uwe.meyer, bjoern.loetters}@mni.thm.de

Zusammenfassung

Die semantische Analysephase eines Compilers wird in der Praxis oft in großen Teilen von Hand implementiert. Im Laufe der letzten zwei Jahrzehnte wurden deshalb Werkzeuge wie JastAdd auf Grundlage von Reference Attribute Grammars (RAGs) entwickelt, mit Hilfe derer die semantische Analyse aus einer Spezifikation erzeugt werden kann. In diesem Paper stellen wir Design Patterns vor, welche die Übersetzung einer formalen Sprachdefinition in eine Spezifikation für JastAdd erleichtern. Damit steuern wir ein methodisches Vorgehen bei, das bei der Entwicklung einer semantischen Analyse eine systematischere Herangehensweise ermöglicht. Eine einfache imperative Programmiersprache dient uns hierbei als formale Ausgangssituation. Darüber hinaus geben wir einen Ausblick darüber, wie unser Vorgehen um weitere Sprachkonstrukte (wie beispielsweise eine Typinferenz) erweitert werden kann.

Abstract

In the last two decades, tools have been implemented to more formally specify the semantic analysis phase of a compiler instead of relying on handwritten code. In this paper, we introduce patterns and a method to translate a formal definition of a language's type system into a specification for JastAdd, which is one of the aforementioned tools based on Reference Attribute Grammars (RAGs). This methodological approach will help language designers and compiler engineers to more systematically use such tools for semantic analysis. As an example, we use a simple, yet complete imperative language and provide an outlook on how the method can be extended to cover further language constructs or even type inference.

1 Einleitung

Compiler construction is one of the oldest and most mature disciplines in computer science. Although some major results such as Chomsky's work on formal languages and grammars [5] or Knuth's work on Attribute Grammar (AG) [17] are now more than 50 years old, there is one aspect of compiler construction that seems to be a focus of research only since the last two decades. Whilst, mostly due to the groundbreaking work mentioned above and others, lexical analysis and syntax analysis are very well understood and available thanks to tools as Lex and Yacc [15] (as well as their successors), ANTLR [20] and PEGs [8], tool support for type analysis has been very limited. This leads to the fact that, even in teaching, these parts are usually handcrafted in many lines

of code using techniques to traverse the Abstract Syntax Tree (AST) (e.g., pattern-matching or the visitor pattern [9] [2]) and apply code fragments to each node type.

Nowadays, there exist few tools to generate the code for the later phases, such as JastAdd [7], Kiama [22] and Silver [25]. There are several successful language implementations using these tools (e.g., PicoJava [13]) and although they are mainly from the groups who have developed the respective tools, it seems that translating a language's semantics into the specification languages of these tools is very much a major design effort.

This paper aims at providing a systematic approach on how to translate a language's formal type system into rules for JastAdd, a framework developed by G. Hedin and her group at Lund University [7] based on her previous work on RAGs [13]. We will formally describe the type system of a simple imperative language using Cardelli's notation [4] and show how JastAdd rules can be derived from it. Hedin [13] and other members of her group provide hints on how to use JastAdd (e.g., for PicoJava, a subset of Java), but we would like to show how a type system can be transcribed into JastAdd code in a more formal manner.

1.1 The Simple Programming Language (SPL)

The Simple Programming Language (SPL) [10] has been developed as a language for a 2nd-year compiler construction course at our university. Besides learning the theory of automata, formal languages and compiler construction itself, students have to implement a complete compiler for this language from lexical analysis to generation of assembler code within one term. While flex[21], JFlex[14], Bison[6], and Cup [19] are used for the lexical and syntactic analysis, the remaining phases have to be implemented by hand in Java or C.

In order to be simple, SPL provides only a handful of procedural language constructs. Among them are assignments, loops, branches and procedures. Arguments are passed either by call-by-value or call-by-reference. There are primarily two different types available to a user, namely an array type constructor with a static length and the primitive type int. Although boolean values and the boolean type are internally used for conditions, they cannot be accessed by the user in a direct way (e.g. declaring a variable of a boolean type is not allowed). In general, types are compared following reference semantics. That is, the reference of the internal type graph is compared whenever two types need to be checked for their equivalence. Because each use of the array type constructor yields a new type and therefore is inequal to any other type, type synonyms allow to introduce aliases by referencing existing types.

SPL's formal type system is defined using the abstract syntax that is shown in figure 1. The notation $\overline{X}$ is an abbreviation for the sequence $X_1, ..., X_n$.

$$
\begin{array}{rlll}
Identifiers & x & & \\
Numerals & \ell & & \\
Program & P & ::= & program\ \overline{D} \\
Types & \tau & ::= & x \\
 & & | & int \\
 & & | & array[\ell]\ of\ \tau \\
Parameters & \rho & ::= & ref\ x : \tau \\
 & & | & val\ x : \tau \\
Variables & V & ::= & var\ x : \tau \\
Declarations & D & ::= & proc\ x(\overline{\rho})\ \{\ \overline{V}\ S\ \} \\
 & & | & type\ x = \tau \\
Statements & S & ::= & T_i := T_j \\
 & & | & x(\overline{T}) \\
 & & | & \overline{S} \\
 & & | & if\ (T)\ S_i\ else\ S_j \\
 & & | & while\ (T)\ S \\
 & & | & skip \\
Terms & T & ::= & x \\
 & & | & T_i[T_j] \\
 & & | & \ell \\
 & & | & T_i +_{bin} T_j \\
 & & | & T_i \leq_{cmp} T_j \\
 & & | & -T
\end{array}
$$

Figure 1: The abstract syntax of SPL

Listing 1 shows an exemplary SPL program that calls a procedure `gcd` which computes the greatest common divisor of two numbers and returns the result using a third, reference parameter. Except for the `#`-symbol, which stands for the inequality operator, the semantics of SPL is straightforward and should not pose any hurdle to a

reader familiar with typical procedural programming languages.

1.2 JastAdd

JastAdd was initially developed by Hedin in 1999 as the result of her research on Reference Attribute Grammar (RAG) [13]. Many extensions have been proposed and implemented since then.

Its core principle can be best described as aspect-oriented RAGs: Given an abstract syntax specification, attributes are defined by equations assembled in aspect files.

```
proc main() {
  var result: int;
  gcd(25, 15, result);
  printi(result);
}

proc gcd(a: int, b: int, ref result:
    int) {
  while (a # b) {
    if (a < b) {
      b := b - a;
    } else {
      a := a - b;
    }
  }
  result := a;
}
```

Listing 1: An exemplary SPL program

The abstract syntax is specified in form of an extension to RAGs, namely object-oriented RAGs [11]. That is, the underlying Context-Free Grammar (CFG) is interpreted as a hierarchy of classes. While the non-terminal on the left-hand side of a production rule leads to an equally named class, the symbols on the right-hand side denote members of this class. The types of these members as well as inheritance of the class can be expressed using annotations. Most interestingly, attributes can be interpreted as methods of this class, which ultimately led to the introduction of parameterized attributes [12]. Because of this correlation between class hierarchies and CFGs, we sometimes refer to non-terminals as nodes (in terms of the nodes of an AST or derivation tree) or classes.

Besides the object-oriented view on RAGs there are different extensions that relate more to the actual attributes. In the context of this paper the most notable of these extensions are:

- **Broadcasting** automatically propagates the value of an inherited attribute down to all child nodes. This feature is especially useful when many simple copy equations would be necessary to accomplish the same otherwise.
- **Collection Attributes** denote a feature that allows the contribution of values to attributes that are marked as such and furthermore may be arbitrary far away in the AST. In some sense this feature is the synthesizing counterpart of broadcasting, since it allows to propagate values up the AST by "sending" them directly to the destination attribute.
- **Rewriting**, as the name suggests, allows to rewrite nodes or even whole subtrees under a selected condition.

Besides those extensions, JastAdd still offers the basic concept of synthesized and inherited attributes just like AGs.

2 Name Analysis

2.1 Approach

As a basis for our investigations, we firstly developed a compiler using Java with JFlex and CUP for the lexical and syntactical analysis. The remaining phases (i.e. name analysis, type analysis and code synthesis) were implemented by hand. Next, we reused the existing code of the lexical and syntactical analysis to implement another compiler using JastAdd as a framework. While the former compiler comprises approximately 2100 lines of code (excluding the scanner and parser), the latter consists of approximately only 700 lines of JastAdd specification files.

It turned out that development using JastAdd is more efficient compared to conventional approaches such as our handcrafted implementation, as it is more abstract and declarative. Yet unsurprisingly, the performance in terms of space and time is slightly better for the handcrafted compiler for larger SPL programs.

After incrementally generalizing the attributes used in the JastAdd implementation, a pattern emerged in the name and type analysis. Ultimately, this pattern leads us to the proceeding described in this work.

$$
\begin{array}{rl}
\textit{Parameters} & \textit{binders}(\textit{ref}\ x:\ \tau) = \{x\} \\
 & \textit{binders}(\textit{val}\ x:\ \tau) = \{x\} \\
\textit{Variables} & \textit{binders}(\textit{var}\ x:\ \tau) = \{x\} \\
\textit{Declarations} & \textit{binders}(\textit{type}\ x = \tau) = \{x\} \\
 & \textit{binders}(\textit{proc}\ x(\overline{\rho})\ \{\ \overline{V}\ S\ \}) = \{x\} \\
\textit{Sequence} & \textit{binders}(D_1,\ D_2,\ ...,\ D_n) = \\
 & \bigcup_{i=1}^{n} \textit{binders}(D_i)
\end{array}
$$

Figure 2: The sets of *binders* for declarations

$$
\begin{array}{rl}
\textit{Parameters} & \textit{fv}(\textit{ref}\ x:\ \tau) = \textit{fv}(\tau) \\
 & \textit{fv}(\textit{val}\ x:\ \tau) = \textit{fv}(\tau) \\
\textit{Variables} & \textit{fv}(\textit{var}\ x:\ \tau) = \textit{fv}(\tau) \\
\textit{Declarations} & \textit{fv}(\textit{type}\ x = \tau) = \textit{fv}(\tau) \\
 & \textit{fv}(\textit{proc}\ x(\overline{\rho})\ \{\ \overline{V}\ S\ \}) = \\
 & \quad \textit{fv}(\overline{\rho}) \cup \textit{fv}(\overline{V}) \\
 & \quad \cup\ (\textit{fv}(S) \setminus \\
 & \qquad (\textit{binders}(\overline{\rho}) \cup \textit{binders}(\overline{V}))) \\
\textit{Program} & \textit{fv}(\textit{program}\ \overline{D}) = \\
 & \quad \textit{fv}(\overline{D}) \setminus \textit{binders}(\overline{D}) \\
\textit{Sequence} & \textit{fv}(D_1,\ D_2,\ ...,\ D_n) = \bigcup_{i=1}^{n} \textit{fv}(D_i)
\end{array}
$$

Figure 3: The sets of free *variables* for declarations

2.2 Free Variables

We will now start examining the name analysis phase of a compiler [1], which collects information about all names occurring in the source program and maps it to static semantic information such as the kind (variable, type, procedure, ...) and detailed attributes such as the type graph. All this information is stored in a symbol table, which usually consists of multiple levels according to the scoping rules of the language.

For example, SPL defines two different scoping levels: One for local declarations such as variables and parameters and one for global declarations like type synonyms and procedures. These scoping rules can be formalized by specifying the set of free *variables*, which ultimately requires to specify how *variables* are bound by declarations [4]. In this context *variables* are not to be confused with local variables of SPL. While the former is formally used to denote all identifiers (i.e. even those that are associated with procedures or types, for example), the latter denotes only identifiers that are bound by the variable declaration of SPLs. For disambiguation, we write the former in italics. Figure 2 and 3 show the relevant parts of SPL's specification for *binders* and free *variables*.

The free *variables* of types, statements and expressions are defined in a straightforward manner, where each occurrence of an identifier x results in a free *variable* x.

2.3 A Method for Deriving a `lookup`-Attribute

Given a formal definition of free *variables* such as the one above, it is possible to derive the rules that are necessary to create and fill a symbol table during the name analysis of a conventional compiler. However, by taking advantage of the reference semantics of RAGs, an explicit symbol table can be avoided. Instead, a parameterized attribute can be used to associate names with a reference to the AST node of the corresponding declaration.

The general method for the introduction of such a parameterized attribute (which we call `lookup`) is as follows:

1. Introduce a new inherited attribute `lookup` to all nodes of the AST.
2. For each language construct that defines a scope,
   a) define a new attribute `declarations` that holds a list of all of its containing declarations.
   b) define a new attribute `lookupInScope` (e.g. `lookupGlobal`, `lookupProcedure`, `lookupBlock`) which searches for the name using the attribute `declarations`.
   c) add an equation for the `lookup` attribute of its direct children in the AST to override the searching behavior.

Besides inherited and parameterized attributes, this proceeding requires broadcasting to unfold its full potential. Broadcasting enables a programmer to define the value of an inherited attribute at an appropriate location in the RAG. JastAdd

then takes care of propagating this value to the non-terminal for which the attribute was actually defined.

The language constructs that define a scope can easily be read off from the definition of fv. As shown in figure 3 there are two equations that use the *binders* function to bind free occurrences of *variables*. It is these equations which indicate language constructs that define scopes. Consequently, step 2 of the procedure above needs to be applied to procedures and programs.

Moreover, it can be read off which scope binds which of the free *variables*. For example, in SPL a procedure's local variables and parameters bind any of the corresponding free occurrences in the procedure's statement. On the contrary, the free *variables* that may occur in the types of local variable and parameter declarations are not bound by them (as it is implied by the mathemtical precedence). That is, free occurrences of procedures and types are bound on the global level, which can be observed in the equation for programs. All in all, this observation indicates the set of declarations that need to be collected in step 2a of our method.

2.3.1 An Exemplary `lookup`-Attribute for SPL

To demonstrate and further explain the presented method we will use SPL's definition of fv and $binders$ to introduce a `lookup` attribute.

Listing 2 shows the inherited `lookup`-attribute as it should be introduced by step 1 of the method. As context information is usually distributed across the syntax tree, it must be passed down to the relevant positions (e.g. a procedure call) from the nearest parent that is able to fully collect this information (e.g. the program). This is the reason why the `lookup`-attribute must be an inherited one:

```
inh Optional<Declaration> Tree.lookup(String
    name);
```

Listing 2: The inherited `lookup`-attribute

Since a programmer may use a variable that is not declared, the `lookup`-attribute returns an `Optional` in order to be total. Overall, an attribute declaration as it can be seen in listing 2 can be read in a manner similar to a Java method signature. In fact, parameterized attributes can even be translated to virtual functions [24] [23].

The second part of the method requires more effort and may vary depending on the actual source language. According to step 2a, we first need a new attribute `declarations` for each language construct that defines a scope. As previously mentioned, those language constructs can be read off from the definition of fv. In case of our simple source language SPL there are exactly two language constructs that define a scope: programs and procedures.

Following step 2a, the `declarations`-attribute should contain all declarations that are part of such a language construct. In case of a program this attribute is equal to `getDeclarationList` (see listing 3). However, in case of a procedure this attribute is the union of `getParameterList` and `getVariableList` (see listing 3). Unsurprisingly, these lists can be read off from the right-hand sides of the definition of fv for programs and procedures, as they correspond to the sets of binders that are used to capture the free variables of the respective scope.

```
syn Collection<Declaration> Program
  .declarations() {
    ArrayList result = new ArrayList<>();
    getDeclarationList().forEach(result::add);
    return result;
}

syn Collection<Declaration> Procedure
  .declarations() {
    ArrayList result = new ArrayList<>();
    getParameterList().forEach(result::add);
    getVariableList().forEach(result::add);
    return result;
  }
```

Listing 3: The `declarations`-attribute for programs and procedures

For reasons of error handling (see subsection 2.4), it is necessary that the `declarations`-attribute is not implemented using a set in the mathematical sense but instead any kind of collection which is able to hold duplicate elements.

According to the next step 2b, we need another new attribute `lookupInScope` for programs and procedures. Since those two language constructs correspond to the global and local declaration level, we named their `lookupInScope`-attributes `lookupGlobal` and `lookupLocal` in the listing 4.

```
syn Optional<Declaration> Program
  .lookupGlobal(String name) =
      declarations()
    .stream().filter(d ->
        d.getName().equals(name))
    .findFirst();

syn Optional<Declaration> Procedure
  .localLookup(String name) = declarations()
    .stream().filter(d ->
        d.getName().equals(name))
    .findFirst();
```

Listing 4: The `lookupGlobal-` and `lookupLocal-`attribute for programs

As it can be very easily seen, both equations in listing 4 look the same. Unfortunately, JastAdd does not allow to define multiple attributes by one equation. That is, unless there is a common superclass that identifies these language constructs as scopes, there is currently no way to remove this redundancy in JastAdd.

The last step 2c ensures that JastAdd consolidates the scope information such that it can be used everywhere. It completes the initial declaration of the inherited `lookup`-attribute of the first step. As mentioned previously (see section 2.3), the broadcasting feature of JastAdd plays an important role in this step. Usually, i.e. without this feature, it would be necessary to add an equation for all subclasses of `Tree` to properly complete the `lookup`-attribute. However, many of these equations would be trivial in that they would just pass the value on to the direct children. With broadcasting support, this is handled automatically by the tool. Consequently, it is sufficient to define an inherited attribute by providing a single equation for it at the root node of the AST. In JastAdd this even works if the root node is not a direct predecessor of the node for which the attribute was defined.

Back to step 2c of our method, this means that it is sufficient to add an equation for the direct descendants of procedures and programs. This overwrites the default behavior of the broadcasting feature, which would simply copy the information of the parent's scope.

As implied by the first three equations in listing 5, the direct descendants of a `Procedure` are `getStatement`, `getVariable(int index)` and `getParameter(int index)`. The latter two notations allow the programmer to define an equation for each element of `getVariableList` and `getParameterList`, respectively.

The only direct descendant of a `Program` is its list of declarations, which is why there is only one equation in this case. What is apparent when comparing the equations of `Procedure` and `Program`, is that the former does not only use the `lookupInScope`-attribute but also the `lookup`-attribute itself. This is due to the usual hierarchy of lexical scopes, as it is necessary to search for a declaration in an upper scope if it is undefined in the current one. Little attention has to be paid regarding the context of the Java expressions: The `lookup`-attribute on the right-hand side refers to that of the `Procedure` and not of its descendants, which means that it, in fact, searches in the upper scope.

```
eq Procedure.getStatement()
  .lookup(String name) = localLookup(name)
    .map(Optional::of)
    .orElseGet(() -> lookup(name));

eq Procedure.getVariable(int index)
  .lookup(String name) = localLookup(name)
    .map(Optional::of)
    .orElseGet(() -> lookup(name));

eq Procedure.getParameter(int index)
  .lookup(String name) = localLookup(name)
    .map(Optional::of)
    .orElseGet(() -> lookup(name));

eq Program.getDeclaration(int index)
  .lookup(String name) = globalLookup(name);
```

Listing 5: The inherited general look-up attribute

With the four presented listings 2, 3, 4 and 5 the implementation of the scoping rules is already complete. Of course, they can be extended freely to provide further features such as predefined standard declarations. In the context of our compiler implementation we were able to add predefined declarations using higher-order AGs (also called Non-Terminal Attributes (NTAs) by JastAdd) and by appropriately extending the `declarations`-attribute of programs.

2.4 Dealing with Naming Errors

Up to this point, we ignored the fact that a program may be erroneous in that a programmer may have declared a name twice or more, for example. In this case, the `lookup`-attribute would simply return one of those conflicting declarations, not indicating that there is something wrong. The other way around, if a name is used that was not previously declared by the programmer, the `lookup`-attribute just returns `Optional.empty()`.

Of course, it is highly desired to find all these errors during the name analysis. Following the idea of Boyland [3], collection attributes can be used to specify the respective error contributions in a precise way. However, it is not as easy as with our previous method and in some cases even not possible at all, to derive the cases of errors just by inspecting the definition of *fv*. Consequently, in addition to the formal definition of *fv* and *binders*, two more constraints are required to complete the specification for the scoping rules.

```
coll ArrayList<String> Program.nameErrors();
```

Listing 6: Collection attributes

$$\forall\ P : fv(P) \neq \emptyset \Rightarrow P\ is\ erroneous$$

Figure 4: A program may not have any free $variables$

The definition of the collection attribute which contains all errors is given in listing 6.

As it can be seen there, this notation resembles that of synthesized attributes. The result type can be chosen quite freely. However, there has to be a default constructor and a method `add` that accepts a new element. Both requirements apply to Java's standard `ArrayList`. The element type of the collection (which is `String` here) determines the type of the contributions we will see later.

2.4.1 Use of Undeclared Variables

In SPL as in many other programming languages it is not allowed to use a $variable$ which was not previously declared. During type analysis this means that looking up a name yields no result (e.g. `Optional.empty()` in our case). Formally, this restriction can be expressed by enforcing that correct programs may not have any free $variables$, which is shown in figure 4.

Given such a restriction and a way to check the premise, it is rather easy to derive an error contribution from it. Transferred to our simple source language SPL, for example, the premise can be checked by testing whether the result of the `lookup`-attribute is present or not for all *variables* in the program. This leads us to the error contribution as it is shown in listing 7.

```
Identifier contributes "undefined variable"
  when !lookup(getName()).isPresent()
  to Program.nameErrors();
```

Listing 7: Error contribution for undeclared *variables*

Given that any use of a *variable* is represented by an instance of the non-terminal `Identifier`, it is then sufficient to check the collection attribute `nameErrors` in order to verify the absence of undeclared *variables* in a program.

2.4.2 Duplicate Declarations

Similar to the restriction that undeclared variables are prohibited, it is usually also disallowed to define the same $variable$ twice in the same scope. Some simple languages may avoid conflicts of this kind, e.g. when there are only language constructs that allow to introduce one $variable$ per scope at a time. However, in SPL this is not the case, which is why the additional restriction in figure 5 is required to complete the specification of the scoping rules.

$$\forall\ program\ \overline{D} : (\bigcap_{i=1}^{n} binders(D_i)) \neq \emptyset$$
$$\Rightarrow program\ \overline{D}\ is\ erroneous \quad (1)$$
$$\forall\ proc\ x(\overline{\rho})\ \{\ \overline{V}\ S\ \} : (\bigcap_{D\ \in\ (\overline{\rho}\ \cup\ \overline{V})} binders(D)) \neq \emptyset$$
$$\Rightarrow proc\ x(\overline{\rho})\ \{\ \overline{V}\ S\ \}\ is\ erroneous \quad (2)$$

Figure 5: Programs and procedures may not have duplicate $variables$

Again, given these two restrictions and a way to check for their premises, it is easy to derive error contributions for them. As the premise verifies that there are no duplicate *variables* in the respective scope (i.e. either the global scope of the program or the local scope of the procedure), it is sufficient to check the corresponding `declarations`-attribute of `Program` and `Procedure` for any duplicates (see listing 8).

```
Program contributes "duplicate variables"
  when new HashSet<>(declarations()).size()
    != declarations().size()
  to Program.nameErrors();

Procedure contributes "duplicate variables"
  when new HashSet<>(declarations()).size()
    != declarations().size()
  to Program.nameErrors();
```

Listing 8: Error contribution for duplicate *variables*

3 Type Analysis

Type analysis, just like the name analysis, is divided into two parts: One that gathers the type information (either by synthesizing or inferring types) and one that checks if the program is well-typed. Clearly, the latter part may yield errors as a result, which again can be collected using collection attributes.

Similar to the approach for the name analysis, it is feasible to derive appropriate attribute equations from the formal specification of the type system. That is, given the set of type rules, attributes can be defined following a method.

Usually, the aspect of a type system which handles the language's expressions is the most challenging one. This is primarily due to the fact that statements or definitions do not have a type at all. For that reason, we will restrict us here to the type rules for SPL's expressions (see figure 6). Moreover, since SPL's type system is deliberately simple, types do not need to be inferred but only synthesized.

(INT)
$$\frac{}{\Gamma \vdash \ell : int}$$

(ARITHMETIC)
$$\frac{\Gamma \vdash T_1 : int \quad \Gamma \vdash T_2 : int}{\Gamma \vdash T_1 +_{bin} T_2 : int}$$

(COMPARISON)
$$\frac{\Gamma \vdash T_1 : int \quad \Gamma \vdash T_2 : int}{\Gamma \vdash T_1 \leq_{cmp} T_2 : bool}$$

(NEGATIVE)
$$\frac{\Gamma \vdash T : int}{\Gamma \vdash -T : int}$$

(VARIABLE)
$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

(ARRAY ACCESS)
$$\frac{\Gamma \vdash T_i : array[\ell] \; of \; \tau \quad \Gamma \vdash T_j : int}{\Gamma \vdash T_i[T_j] : \tau}$$

Figure 6: The type rules for SPL expressions

Besides the scoping rules and type rules, it is necessary to specify in which case two types can be considered equal (e.g., to check the well-formedness of an assignment). The two most common type equivalences are *name* and *structural equivalence*. Yet, SPL uses neither of them. Instead, it uses an equivalence which we call *referential equivalence* (as SPL introduces reference semantics for its types). Thanks to this special kind of type equivalence, Java's standard equality operator (==) can be used to test if two types are equal. In general, however, it is a much better idea to introduce a new method for types named `isEqualTo`, which implements the respective type equivalence. In the presence of subtyping it is conceivable to introduce another method `isSubTypeOf` as well.
To be able to reference the exact same type twice in a program, SPL provides type synonyms. The additional rule in figure 7 expresses the fact that a type synonym is equal to its referenced type. More technically it is necessary to resolve all type synonyms in JastAdd, as it would not be possible to compare two types using Java's equality operator otherwise. To illustrate this, consider for example comparing a named type `x` which is synonymous for another type τ. Clearly, the Java objects representing the types `x` and τ are not the exact same object and thus do not share the same reference.

(TYPE SUBSTITUTION)
$$\frac{\Gamma, x = \tau \vdash T : x}{\Gamma, x = \tau \vdash T : \tau}$$

Figure 7: The type rule for type synonyms

Similar to the circumstance that no explicit symbol table is necessary when using RAGs, no explicit data structure is required to model types. The subset of the AST that represents type expressions can be reused to represent types during the type analysis. The two methods `isEqualTo` and `isSubTypeOf` may then be implemented as parameterized attributes. Moreover, type synonyms can be implemented by rewriting the AST, which is very well supported by JastAdd or more generally rewritable RAGs.

```
rewrite NameType {
  when (lookup(getName()).orElse(null)
    instanceof TypeSynonym)
  to Type ((TypeSynonym)
      lookup(getName()).get())
    .getType();
}
```

Listing 9: Type synonyms using rewritable RAGs

Listing 9 shows the rewrite rule that implements the type rule of figure 7. In fact, this rewrite rule may be even read in a similar way to the type rule: Given a type x which is synonymous for τ (i.e. x has to be a named type), we can also use type τ instead of x. More precisely, a named type (`NameType`) is rewritten if the condition after the keyword `when` holds. This condition stands for the premise that x has to be a type synonym in the current context (which is expressed by the notation $\Gamma, x = \tau$ in the formal specification). The result of the rewriting process is the type for which the named type is synonymous.

3.1 A Method for Deriving a Type Analysis

As mentioned previously, type information only needs to be synthesized by inspecting the respective expression in SPL (as opposed to type inference). Although we are confident that it is generally conceivable to derive attribute equations from a type system that supports type inference (e.g. Hindley-Milner), we are still in progress to formalize our results to an extent where they lead to a pattern applicable to type inference.
Hence, we will restrict us to type synthesis in the rest of this paper. In general, synthesizing type rules can be translated to attribute equations using the following method:

1. Introduce a new synthesized attribute `type` for all non-terminals which formally have a type
2. Introduce a new collection attribute `typeErrors` similar to `nameErrors`
3. For each type rule that assigns a type τ to one of the non-terminals of step 1:
   a) Add an equation that assigns τ to the `type`-attribute of the respective non-terminal. In case there are multiple type rules for the same syntactic construct conflicts may arise. These conflicts have to be resolved by either evaluating the context, by removing the conflicting type rules or by further distinguishing the syntactic constructs. If τ depends on a premise (e.g. see the type rule (ARRAY ACCESS) where the result type depends on the left premise), a bottom type $\bot$ has to be returned if this premise is not fulfilled. This bottom type should be equal to any other type, which is most easily achieved by appropriately extending the parameterized attributes `isEqualTo` and `isSubTypeOf`.
   b) Add a contribution to the `typeErrors` attribute for each premise of the type rule

Clearly, there are other premises (e.g. the condition of an *if-statement* has to have a boolean type), which we have not shown here for the sake of brevity. Each of these requires a contribution to the `typeErrors` attribute similar to step 3b.

3.2 An Exemplary Implementation of (INT) and (ARRAY ACCESS)

To demonstrate the presented method, we will use it to derive the respective attributes and their equations for the type rules (INT) and (ARRAY ACCESS) presented earlier.
As shown in listing 10, two new attributes are required according to step 1 and 2. The first one is the `type`-attribute which is defined for all expressions (statements and declarations do not have a type in SPL). The collection attribute `typeErrors` is defined similarly to the `nameErrors`-attribute in subsection 2.2.

```
syn Type Expression.type();
coll ArrayList<String> Program.typeErrors();
```

Listing 10: The `type`-attribute for SPL expressions and the global `typeErrors`-attribute

For the sake of brevity, we apply step 3 in this example only to the two type rules (INT) and (ARRAY ACCESS). In general, a "type rule that assigns a type τ to one of the non-terminals of step 1" can be identified by the form of the conclusion. That is, if the conclusion depicts the form $\Gamma \vdash e : \tau$ it assigns τ to the non-terminal that represents the corresponding class of e. For example, in case of the type rule (INT), e is an integer literal ℓ and thus the type int is assigned to the non-terminal we named `IntegerLiteral`. In fact, this already corresponds to the equation of step 3a (see listing 11). Because the type rule (INT) is an axiom (i.e. it has no premise), step 3b can be skipped.

```
eq IntegerLiteral.type() = intType;
```

Listing 11: The equation for the `type`-attribute of (INT)

In contrast to (INT), the type rule (ARRAY ACCESS) has two premises. But first things first: The conclusion of the type rule assigns each array access the base type τ of the accessed array. As defined in step 3a, when the result type depends on a premise, the bottom type $\perp$ has to be returned if this premise is not fulfilled. This is the case if the expression T_i in the premise of rule (ARRAY ACCESS) is not an array (see listing 12).

```
eq Access.type() =
  getExp().type().isArrayType())
    ? ((ArrayType)
        getExp().type()).getBaseType()
    : bottomType;
```

Listing 12: The equation for the `type`-attribute of (ARRAY ACCESS)

As mentioned previously, the (ARRAY ACCESS) rule has two premises: One requiring that the accessed expression is an array and another one requiring that the index which is used to access the array is an integer. Although the implementation of the premises might be more complex depending on the actual language, the proceeding remains the same. By negating the premise, the error contribution can be written very naturally (as shown in listing 13). The premises themselves can be implemented using a custom attribute (e.g. `expHasToBeAnArray()` and `indexHasToBeAnInteger()`).

```
Access contributes "illegal access"
  when !expHasToBeAnArray()
  to Program.typeErrors();

Access contributes "illegal index"
  when !indexHasToBeAnInteger()
  to Program.typeErrors();
```

Listing 13: The respective error contributions for the premises of the type rule (ARRAY ACCESS)

Together, listing 12 and 13 implement the type rule (ARRAY ACCESS). While the synthesized attribute constitutes the conclusion, the error contributions represents the premises.

4 Summary

We have shown that a formal type system for a language can constructively and methodologically be translated into a specification for a RAG-based tool. The approach has been successfully implemented to generate a compiler (including code generation) for SPL that performs (in terms of memory and time used for compilation) as well as a hand-crafted compiler.

The method is based on the JastAdd RAG mechanisms and does not require extensions. Furthermore, because these mechanisms are not exclusive to JastAdd but are general extensions to AGs and RAGs, it is feasible to adapt our method to other tools as well. Even though SPL is a simple imperative language, further analysis has indicated that an extension of the approach is possible. For example, type inference according to Hindley-Milner should be implementable using JastAdd and the well-known Algorithm W [18]. Because of the paradigm of AGs (i.e. attributes should not have externally visible side effects) the introduction of an appropriate monad (as described in [16]) could be necessary, though.
We are confident, that the approach to start not only with formal grammars for lexical and syntax analysis, but also with a formal type system brings benefits to language designers since - as shown - this formalism can be translated into a specification for JastAdd (and potentially other tools), and thus eliminates the need for hand-written framework code in a compiler's name analysis and type analysis. An implementation of the method directly into JastAdd seems possible, but hasn't been started due to resource constraints.

Akronyme

AG Attribute Grammar
RAG Reference Attribute Grammar
CFG Context-Free Grammar
SPL Simple Programming Language
AST Abstract Syntax Tree
NTA Non-Terminal Attribute

References

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006. ISBN: 0321486811.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Jan. 12, 2004. ISBN: 0-521-58388-8.

[3] John Tang Boyland. "Descriptional Composition of Compiler Components". Available as technical report UCB//CSD-96-916. PhD thesis. University of California, Berkeley, Sept. 1996. URL: http://cs-tr.cs.berkeley.edu:80/Dienst/UI/2.0/Describe/ncstrl.ucb%2fcsd-96-916?abstract=.

[4] Luca Cardelli. “Type Systems.” In: *The Computer Science and Engineering Handbook*. Ed. by Allen B. Tucker. CRC Press, 1997, pp. 2208–2236. ISBN: 0-8493-2909-4. URL: http://dblp.uni-trier.de/db/books/collections/tucker97.html#Cardelli97.

[5] N. Chomsky. “Three models for the description of language”. In: *IEEE Trans. Information Theory* 2.3 (1956), pp. 113–124. ISSN: 0018-9448. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1056813.

[6] Charles Donnelly and Richard M. Stallman. *BISON—The YACC-compatible Parser Generator*. Tech. rep. Bison was largely written by Robert Corbett, and made yacc-compatible by Richard Stallman. See also [21]. 1988. URL: ftp://ftp.gnu.org/pub/gnu/bison/.

[7] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *OOPSLA ’07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. Montreal, Quebec, Canada: ACM, 2007, pp. 1–18. ISBN: 978-1-59593-786-5.

[8] Bryan Ford. “Parsing Expression Grammars: A Recognition-based Syntactic Foundation”. In: *SIGPLAN Not.* 39.1 (Jan. 2004), pp. 111–122. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/982962.964011.

[9] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* 1st ed., Reprint. Addison-Wesley Longman, Amsterdam, 1994. ISBN: 0201633612.

[10] Hellwig Geisse. *SPL Language Specification*.

[11] Görel Hedin. “An Object-Oriented Notation for Attribute Grammars”. In: *ECOOP*. 1989.

[12] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24 (2000).

[13] Gorel Hedin. “Reference Attributed Grammars”. In: *Second Workshop on Attribute Grammars and their Applications, WAGA’99*. Ed. by D. Parigot and M. Mernik. Amsterdam, The Netherlands: INRIA rocquencourt, Mar. 1999, pp. 153–172. URL: http://www-sop.inria.fr/members/Didier.Parigot/www/fnc2/WAGA99/accept.html.

[14] JFlex.de. *JFlex The Fast Scanner Generator*. 2018. URL: https://jflex.de/.

[15] Stephen C. Johnson. “Yacc: Yet Another Compiler-Compiler"; Computer Science Technical Report”. In: 1975.

[16] Simon L. Peyton Jones et al. “Practical type inference for arbitrary-rank types.” In: *J. Funct. Program.* 17.1 (2007), pp. 1–82. URL: http://dblp.uni-trier.de/db/journals/jfp/jfp17.html#JonesVWS07.

[17] Donald E. Knuth. “Semantics of Context-free Languages”. In: *Mathematical Systems Theory* 2.2 (June 1968). Correction: Mathematical Systems Theory 5, 1, pp. 95-96 (March 1971)., pp. 127–145.

[18] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375.

[19] Technical University of Munich. *CUP*. 2019. URL: http://www2.cs.tum.edu/projects/cup/.

[20] Terence John Parr and Russell W. Quong. “ANTLR: A Predicated- LL(k) Parser Generator.” In: *Softw., Pract. Exper.* 25.7 (1995), pp. 789–810. URL: http://www.antlr.org/article/1055550346383/antlr.pdf.

[21] Vern Paxson. *flex — fast lexical analyzer generator*. GNU software package. See also [6] for the bison parser generator. 1988. URL: http://flex.sourceforge.net/.

[22] Anthony M. Sloane. “Lightweight Language Processing in Kiama.” In: *GTTSE*. Ed. by João M. Fernandes et al. Vol. 6491. Lecture Notes in Computer Science. Springer, 2009, pp. 408–425. ISBN: 978-3-642-18022-4. URL: http://dblp.uni-trier.de/db/conf/gttse/gttse2009.html#Sloane09.

[23] Emma Söderberg and Görel Hedin. “Circular Higher-Order Reference Attribute Grammars”. In: *Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Cham: Springer International Publishing, 2013, pp. 302–321. ISBN: 978-3-319-02654-1.

[24] S. Doaitse Swierstra and Harald H. Vogt. “Higher Order Attribute Grammars”. In: *Attribute Grammars, Applications and Systems*. Ed. by Henk Alblas and Borivoj Melichar. Vol. 545. Lect. Notes in Comp. Sci. New York–Heidelberg–Berlin: Springer-Verlag, June 1991, pp. 256–296.

[25] Eric Van Wyk et al. “Silver: an Extensible Attribute Grammar System.” In: *Electr. Notes Theor. Comput. Sci.* 203.2 (June 11, 2008), pp. 103–116. URL: http://dblp.uni-trier.de/db/journals/entcs/entcs203.html#WykBGK08.

Notationsgrammatiken

Björn Lötters und Uwe Meyer
Technische Hochschule Mittelhessen
University of Applied Sciences
Wiesenstraße 14, D–35390 Gießen
{bjoern.loetters,uwe.meyer}@mni.thm.de

1 Einleitung

Die Fähigkeit einer Programmiersprache, die eigene Syntax zu erweitern, ist im Sprachentwurf und Compilerbau keine leichte Aufgabe. Das Resultat aber lohnt sich: In Anbetracht der heutigen Informationssysteme, die längst in den unterschiedlichsten Anwendungsdomänen angekommen sind, müssen Programmiersprachen flexibler und erweiterbarer sein denn je.

Dabei ist die Idee keinesfalls neu: Schon Lisp hat seine homoikonische Syntax in Kombination mit seinem Makrosystem dazu verwendet, sich selbst um neue Konstrukte zu erweitern. Interaktive Beweisassistenten wie Lean haben später den großen Vorteil der syntaktischen Erweiterbarkeit darin gefunden, dass der eigentliche Kern der Sprache sehr klein bleibt. Somit ist die Implementierung um einiges leichter und fehlerfreier möglich. Objektorientierte Sprachen wie Smalltalk oder konkatenative Sprachen wie Factor verlassen sich hingegen auf eine so schwach strukturierte Syntax, dass diese leicht mit einer neuen Bedeutung versehen werden kann.

2 Problematik mit bisherigen Ansätzen

Nichtsdestotrotz gehen alle diese Erweiterungsmechanismen einen Handel ein: Ihr zugrundeliegendes System ist entweder zu einem gewissen Grad eingeschränkt oder muss sich auf prozedurale Aspekte der definitorischen Metaebene verlassen. Um sich hiervon zu überzeugen, möge man sich die verschiedene Systeme vor Augen führen: Ein Makrosystem, wie das von Racket, trennt die Laufzeit des Programms von der eigentlichen Makroexpansion [2]. Programmiersprachen wie Scala [5] schränken Makros insofern ein, als dass diese in einer getrennten Übersetzungseinheit definiert werden müssen. Syntaxerweiterungen, wie sie beispielsweise in Lean [8] definiert werden können, verlassen sich auf die Interaktivität ihrer Sprache: Jede Definition und damit auch Erweiterung wird Zeile für Zeile wie ein Befehl ausgeführt. Es entsteht also immer auch eine topologische Sortierung auf der definitorischen Metaebene. Die Kehrseite lässt sich an der Programmiersprache Haskell [4] beobachten: Mithilfe der `infix`-Deklarationen können unabhängig von ihrer Reihenfolge eigene Binäroperatoren eingeführt werden. Diese Form der Erweiterbarkeit ist im Vergleich stark eingeschränkt: Die Definition von syntaktischen Konstrukten, die über Binäroperatoren hinaus gehen (ganz zu schweigen von beliebigen kontextfreien Erweiterungen), ist nicht möglich.

3 Ziele und bisherige Resultate

Das Ziel unserer Arbeit ist es, die Grenzen eines Systems zur deklarativen Syntaxerweiterung zu erforschen. Im Rahmen dessen entwickeln wir eine neue Klasse formaler Grammatiken, die Notationsgrammatiken. Diese sind eng mit Donald E. Knuths Attributgrammatiken [3] und Definit-Klausel Grammatiken [6] verbunden und können in die Oberklasse der deklarativen, adaptiven Grammatiken eingeordnet werden [1] [7]. Im Gegensatz zu anderen Formalismen sind unsere Notationsgrammatiken eng mit dem Lambda-Kalkül als universellem Modell

für Berechnungen verknüpft. Den Produktionsregeln einer Notationsgrammatik werden hierbei entweder vordefinierte oder benutzerdefinierte Bedeutungen in Form von Termen des Lambda-Kalküls zugeordnet. Die Besonderheit ist, dass diese Terme selbst Gebrauch von den eingeführten Notationen machen können und dass ihre konkrete Syntax nicht vordefiniert werden muss, sondern mithilfe des Formalismus' selbst definiert werden kann.

Das Listing 1 zeigt eine exemplarische Notationsgrammatik für die üblichen Elemente des Lambda-Kalküls (erweitert um `let`). Terminale sind hierbei von Anführungszeichen umschlossen. Nicht-Terminale beginnen mit einem Großbuchstaben. Wie in Definit-Klausel Grammatiken können Nicht-Terminale als Prädikate betrachtet und angewandt werden. Der grundlegendste Unterschied ist, dass Variablen – hier die kleingeschriebenen Symbole – nicht für beliebige Werte stehen, sondern immer mit einem Produkt aus bindenden Ausdrücken instanziiert werden.

```
Exp -> Let | Abs | App | Var
Let -> "let" (x: Exp) "=" (f: Exp x)
         "in" (g: Exp x) = (g (fix f))
Abs -> (x: Var) "=>" (Exp x)
App -> "(" Exp Exp ")"
```

Listing 1: Beispiel einer Notationsgrammatik

Weitere Untersuchungen widmen wir vor allem dem Zusammenspiel von Notationsgrammatiken und Typsystemen sowie der Modularisierung unseres Formalismus'. Eine Anwendung soll somit auch für interaktive Beweisassistenten und rein funktionale Programmiersprachen möglich werden.

References

[1] H. Christiansen. "A Survey of Adaptable Grammars". In: *SIGPLAN Not.* 25.11 (Nov. 1990), pp. 35–44. ISSN: 0362-1340. URL: https://doi.org/10.1145/101356.101357.

[2] Ryan Culpepper et al. "From Macros to DSLs: The Evolution of Racket". In: *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Ed. by Benjamin S. Lerner, Rastislav Bodik, and Shriram Krishnamurthi. Vol. 136. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 5:1–5:19. ISBN: 978-3-95977-113-9. URL: https://drops.dagstuhl.de/opus/volltexte/2019/10548.

[3] Donald Ervin Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2 (1968), pp. 127–145. URL: https://api.semanticscholar.org/CorpusID:5182310.

[4] Simon Marlow et al. "Haskell 2010 language report". In: *Available online http://www.haskell.org/(May 2011)* (2010).

[5] Martin Odersky et al. "An overview of the Scala programming language". In: (2004).

[6] Fernando C. N. Pereira and David H. D. Warren. "Parsing as Deduction". In: *21st Annual Meeting of the Association for Computational Linguistics*. Cambridge, Massachusetts, USA: Association for Computational Linguistics, June 1983, pp. 137–144. URL: https://aclanthology.org/P83-1021.

[7] John N. Shutt. "Recursive Adaptable Grammars". In: 1999. URL: https://api.semanticscholar.org/CorpusID:64226502.

[8] Sebastian Ullrich and Leonardo de Moura. "Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages". In: *Log. Methods Comput. Sci.* 18.2 (2022). URL: https://doi.org/10.46298/lmcs-18(2:1)2022.

Ein Schritt in Richtung Seeing Spaces: Programmiersprachunterstützung zur Visualisierung innerer Systemzustände

Ulrich Hoffmann
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
uh@fh-wedel.de

Abstract

Systeme stellen sich häufig als *black boxes* dar, bei denen nur das äußere Verhalten beobachtet und über das innere Funktionieren und die inneren Zustände des Systems nur gemutmaßt werden kann. Genaue Kenntnisse über den inneren Zustand eines Systems sind aber besonders während der Exploration und Entwicklung von Systemen insbesondere von Software–Systemen von großer Bedeutung. Klassische Debug–Methoden zeigen lediglich punktuelle Ausschnitte des Systemzustands an.
Mit seiner Idee der *Seeing Spaces* schlägt Bret Victor Entwicklungs–Räumlichkeiten vor, in denen der Raum Zustände eines Probanden–Systems vielfach über Sensoren erfasst und aufbereitet u.a. über Displays und Projektionen visualisiert. Software–Systeme müssen an dieser Stelle kooperieren und ihren inneren Zustand preisgeben, etwa durch kontinuierliche Kommunikation von Kennwerten an den Raum.
Der Vortrag präsentiert programmiersprachliche Unterstützung zur Kommunikation innerer Systemzustände (Gesamtheit der Variableninhalte und Informationen über den Kontrollfluss). Beispiele hierfür sind etwa der `delay`-Mechanismus in Golang, der Aktionen — etwa Protokollierung — beim regulären oder irregulären Verlassen einer Prozeduraktivierung erlaubt oder die Methodenkombination von CommonLisp. Objektorientierte Programmiersprachen bieten ganz allgemein durch ihr Generalisierungs-/Spezialisierungskonzept die Möglichkeit zusätzliche Funktionalitäten in Spezialisierungen auszudrücken ohne die Original-Implementierungen ändern zu müssen (vgl. einschlägige Entwurfsmuster). Metaprogrammierung erlaubt durch Programmtransformationen zur Übersetzungszeit, die nötigen Erweiterungen minimal-invasiv etwa durch die Angabe von Dekoratoren wie in Python zu notieren. Konkret stellt der Vortrag solche Dekoratoren in Python und Forth vor. Sie senden die inneren Systemzustände kontinuierlich an einen Server des Seeing Spaces, welcher diese dann über die Zeit hinweg visualisiert, aufzeichnet und nach Bedarf wieder abspielen kann. Die Architektur des Seeing Spaces und ein Prototyp für einen zugehörigen Seeing Space Server der mittels Node-Red konfiguriert werden kann, werden vorgestellt.
Der Einsatz eines Seeing Spaces ermöglicht es Entwicklern, die inneren Systemzustände permanent zu beobachten, aufzuzeichnen und online sowie offline zu inspizieren.
Die Visualisierung innerer Systemzustände führt zu verbessertem Verständnis des Systems, zu erhöhter Einsicht in dessen Funktionsweise und als Folge zu höherer Qualität und zu kürzeren Entwicklungszeiten.

Type Inference for Featherweight Java

Andreas Stadelmeier & Martin Plümicke
Duale Hochschule Baden-Württemberg (DHBW) Stuttgart Campus Horb
Department of Computer Science
Florianstraße 15, 72160 Horb Germany
andreas.stadelmeier@dhbw.de
martin.pluemicke@dhbw.de

Abstract

Featherweight Java is a reduced language in which complex features like threads and reflections, but also interfaces and assignments are dropped out. In Featherweight Java rigorous proofs are easy. Featherweight Java bears a similar relation to Java as lambda-calculus to functional programming languages like Haskell.
In this paper we add to Featherweight Java a global type inference algorithm. Featherweight Java allows to reduce the type inference algorithm such that the proofs become easy.

1 Introduction

Functional programming languages like Haskell or SML have global type inference systems since the late 1980s. In Java only local type inference [11, 12] is introduced which means that only the declaration of variables were a constant is assigned to, like

```
var n = 5;
```

or in the so-call diamond operator the parameters in a new–expression

```
ArrayList<String> = new ArrayList<>();
```

can be left out.

But in methods or in assignment of non-constant expression type annotations have to be given.

In [14] we gave a type inference algorithm for a subset of Standard Java. The subset contains all relevant features of an imperative object-oriented languages, such that real program can be given in the subset. As in the Java language specification (e.g. [6]) no formal type system is given it is difficult to prove properties of the type inference algorithm like soundnees and completeness. In contrast Featherweight Java [9] has a formal type system but it is very difficult to declare relevant programs, as no numbers, no conditions and no assignments are included.

Therefore we transfer our type inference algorithm to Featherweight Java to prove the important properties.

In this paper we consider as a first step Featherweight Java without generics and without lambda expression which corresponds to the original Java from the late 1990s. We call Featherweight Java with type inference Typeless Featherweight Java. (TFJ).

The paper is structured as follows. In the second section we define the syntax of TFJ. In the third section we give the type rules. In fourth section we give the type inference algorithm FJType which contains the type unification algorithm (Sec. 4.3). In the fifth section we prove soundness and correctness of the algorithms. We close with related work (Sec. 6), a conclusion and an outlook.

$$
\begin{aligned}
\texttt{L} &::= \texttt{class C extends C}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\\
\texttt{K} &::= \texttt{C}(\overline{\texttt{T}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{f}});\ \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\}\\
\texttt{M} &::= \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\texttt{return e};\}\\
\texttt{e} &::= \texttt{this} \mid \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new C}(\overline{\texttt{e}}) \mid (\texttt{C})\texttt{e}\\
\texttt{T} &::= \texttt{C} \mid \epsilon
\end{aligned}
$$

Figure 1: Typeless Featherweight Java Syntax

2 Typeless FJ syntax

The Typeless Featherweight Java (TFJ) syntax differs from normal Featherweight Java (FJ) by the fact, that it is possible to omit any type annotation.We declare the syntax for TFJ in figure 1. One can see that the type annotations are optional in TFJ ($T ::= C \mid \epsilon$). Another difference to the syntax of FJ is that we added the special variable `this` to the syntax. FJ treats `this` as a normal variable but our algorithm treats it as a special variable which always has a predetermined type; the type of the class it is used in.

3 Featherweight Java Type rules

We use the unchanged type rules from Featherweight Java. Our type inference algorithm tries to find a type solution for the missing types, which complies with all the typing rules given in this section. We will also show, that if there is a possible typing under these rules, our type inference algorithm will find it.

4 FJType Algorithm

At first we have to declare the two main data structures used by our type inference algorithm. The **FJTYPE** algorithm produces two kinds of constraints:

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

Or-Constraint An Or-Constraint consists out of multiple constraint sets. For example **OrConstraint**$(\{\ \{(a \lessdot b), (a \lessdot \texttt{Object})\}\ ,\ \{(a \lessdot b)\}\ \})$ is an Or-Constraint consisting of two constraint sets. The way Or-Constraints are handled is further described in Chapter 4.3.

Now we define the type inference algorithm **FJ-Type** on the TFJ syntax from chapter 2. The algorithm is split into following parts:

1. Adding type variables to input
2. Constraint generation with **FJType**
3. Unification of those constraints
4. Set in a type solution

4.1 Adding Type Variables

Before generating the constraints we add type variables to all fields, method parameters and return types which have no type ($T = \epsilon$). This process is shown in figures 2 and 3. We use distinct or so called fresh type variables for each field and method, apart from the following exceptions: If a method overrides another method both of them get the same type variables. Also the constructor parameters get the same type variables as the fields they are instantiating. See for example the field `g` in figure 3.

4.2 FJTYPE

In the next step we generate constraints by calling **TYPEClass** with each class defined in the input.

The function **TYPEClass** is given as follows:

TYPEClass: $\texttt{Class} \rightarrow \texttt{Constraints}$
TYPEClass$(\ \texttt{class}\ C\ \texttt{extends}\ D\{\overline{T}\ \overline{f};\ K\ \overline{M}\}\ ) =$
$\quad \{\ \textbf{TYPEMethod}(\mathit{Ass} \cup \{\texttt{this} : C\}, m_i) \quad \mid \quad m_i \in \overline{M}\ \}$

The **TYPEClass** function gets called for every class in the input. This function accumulates the constraints of the inherited methods. The assumptions variable Ass is used to pass the type of `this` as well as the types of the method parameters.

TYPEMethod : $\texttt{TypeAssumptions} \times \texttt{Method} \rightarrow \texttt{Constraints}$
TYPEMethod$(\ Ass, T_r\ \texttt{m}(\overline{T}\ \overline{x})\{\texttt{return}\ e;\}\ ) =$
$\quad$ **let** $Ass_m = Ass \cup \{\overline{T} : \overline{x}\}$
$\quad\quad \overline{(e : rty, ConS)} = \textbf{TYPEExpr}(\ Ass_m, e\ )$
$\quad$ **in** $\overline{(ConS \cup (rty \lessdot T_r))}$

Subtyping:

$$T <: T \qquad \text{S-REFL}$$

$$\frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}} \qquad \text{S-TRANS}$$

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}} \qquad \text{S-CLASS}$$

Expression Typing:

$$\Gamma \vdash x : \Gamma(x) \qquad \text{T-VAR}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{C}_i} \qquad \text{T-FIELD}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad \mathit{mtype}(m, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \Gamma \vdash \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{T-INVK}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \triangle; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \triangle \vdash \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{T-NEW}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 : \mathtt{C}} \qquad \text{T-UCAST}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{C} <: \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 : \mathtt{C}} \qquad \text{T-DCAST}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 : \mathtt{C}} \qquad \text{T-SCAST}$$

Method Typing:

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \qquad \mathtt{class\ C\ extends D}\{\ldots\}}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{C}})\{\ \mathtt{return\ e}_0;\ \}\ \mathtt{OK\ in\ C}} \qquad \text{T-METHOD}$$

Class Typing:

$$\frac{\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}}{\mathtt{class\ C\ extends\ D}\{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \qquad \text{T-CLASS}$$

Method type lookup:

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{B\ m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}} \qquad \text{MT-CLASS}$$

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathit{mtype}(\mathtt{m}, \mathtt{D})} \qquad \text{MT-OVERRIDE}$$

```
class Example extends Object{
  g;
  Example(g){
    super();
    this.g = g;
  }
  calculate(param1, param2){
    return this.g;
  }
}
class Example2 extends Example{
  String f;
  Example2(String f, g){
    super(g);
    this.f = f;
  }
  calculate(param1, param2){
    return this.g;
  }
}
```

Figure 2: Typeless GFJ program

```
class Example extends Object{
  G g;
  Example(G g){
    super();
    this.g = g;
  }
  C calculate(A param1, B param2){
    return this.g;
  }
}
class Example2 extends Example{
  String f;
  Example2(String f, G g){
    super(g);
    this.f = f;
  }
  C calculate(A param1, B param2){
    return this.g;
  }
}
```

Figure 3: After inserting type variables

The **TYPEMethod** function for methods just calls the **TYPEExpr** function with the return expression. It is significant to note that it adds the assumptions for the method parameters to the global assumptions before passing them to **TYPEExpr**.

In the following we define the **TYPEExpr** function for every possible expression:

TYPEExpr : $\texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{Type} \times \texttt{Constraints}$

TYPEExpr$(\ Ass, \texttt{this}\ ) = (t, \{\})$
with $(\texttt{this} : t) \in Ass$

TYPEExpr$(\ Ass, x\ ) = (t, \{\})$
with $(x : t) \in Ass$

TYPEExpr$(\ Ass, e.f\ ) =$
let $(rty, ConS) =$ **TYPEExpr**$(\ Ass, e\ ),$
$Cons_f = \{\ rty \doteq \texttt{C}, a \doteq \texttt{C}_\texttt{i}$
$\mid$ for every field $\texttt{C}_\texttt{i}\ \texttt{f}_\texttt{i} \in \overline{\texttt{f}}$
$\mid$ in each class $\texttt{C}\ \texttt{extends}\ \texttt{D}\ \{\ldots\overline{\texttt{C}}\ \overline{\texttt{f}}\ldots\}\}$
in $(a, ConS \cup \texttt{OrConstraint}(Cons_f))$
where a is a fresh type variable

TYPEExpr$(\ Ass, e_r.\texttt{m}(\overline{e})\ ) =$
let $(rty, ConS) =$ **TYPEExpr**$(\ Ass, e_r\ ),$
$\forall e_i \in \overline{e} : (pt_i, ConS_i) =$ **TYPEExpr**$(\ Ass, e_i\ ),$
$Cons_m = \{\ rty \doteq \texttt{C}, a \doteq \texttt{U}, \bigcup_{\texttt{U}_\texttt{i} \in \overline{\texttt{U}}}(\texttt{pt}_\texttt{i} \lessdot \texttt{U}_\texttt{i})\}$
$\mid$ for every method $\texttt{U}\ \texttt{m}(\overline{\texttt{U}}\ \overline{\texttt{x}})\{\ldots\} \in \overline{\texttt{M}},$
$\mid$ in each class $\texttt{C}\ \texttt{extends}\ \texttt{D}\ \{\ldots\overline{\texttt{M}}\ldots\}\}$
in $(a, ConS \cup \texttt{OrConstraint}(Cons_m)$
$\cup \bigcup_i ConS_i))$
where a is a fresh type variable

TYPEExpr$(\ Ass, \texttt{new}\ C(\overline{e})\ ) =$
let $\forall e_i \in \overline{e} : (pt_i, ConS_i) =$ **TYPEExpr**$(\ Ass, e_i\ ),$
$Cons = \{\bigcup_{\texttt{T}_\texttt{i} \in \overline{\texttt{T}}}(pt_i \lessdot \texttt{T}_\texttt{i}) \mid \texttt{class}\ \texttt{C}\{\ldots\texttt{C}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ldots\}\}$
in $(C, ConS \cup \bigcup_i ConS_i)$

The **TYPEExpr** function commits the actual work by creating the **Constraints** and **OrConstraints**. After the function **TYPEClass** is called for every class in the input, the generated constraints are accumulated and unified (see chapter 4.3).

4.3 Unify

This chapter describes the **Unify** algorithm which is used to find type solutions for the constraints generated by **FJType**.

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

output A set of type unifiers $Uni = \{\sigma_1, \ldots, \sigma_n\}$ or fail $Uni = \emptyset$

postcondition If the algorithm succeeds the resulting unifiers assign a type to every type variable in the input set $Cons_{in}$

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

$$\Omega = \text{All } \mathbf{OrConstraints} \text{ in } Cons_{in}$$
$$C = Cons_{in} \setminus \Omega$$
$$Eq_{set} = \Omega_1 \times \ldots \times \Omega_n \times C \quad \text{for all } \Omega_i \in \Omega$$

With the resulting Eq_{set} the following algorithm is called.

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

$$\text{(erase2)}\quad \frac{Eq \cup \{\, \mathtt{C} \doteq \mathtt{C} \,\}}{Eq}$$

$$\text{(swap)}\quad \frac{Eq \cup \{\, \mathtt{C} \doteq \mathtt{a} \,\}}{Eq \cup \{\, a \doteq \mathtt{C} \,\}}$$

Figure 4: Erase and swap rules

1. Repeated application of the *erase* rules and the *swap* rule (Fig. 4) to all elements of Eq_{set}. The end configuration of Eq_{set} is reached if for each constraint no rule is applicable.

2. $\forall Eq_i \in Eq_{set}$:
   $Eq'_i =$ Subset of pairs in Eq_i,
   where both type terms are type variables
   $$Eq^i_{set} = Eq'_i \times \Big(\bigotimes_{(a \lessdot \mathtt{C}) \in \mathtt{Eq}} \{\, ([a \doteq \mathtt{D}]) \mid \mathtt{D} <: \mathtt{C} \}$$
   $$\times \Big(\bigotimes_{(\mathtt{C} \lessdot \mathtt{a}) \in \mathtt{Eq}} \{\, [a \doteq \mathtt{D}] \mid \mathtt{C} <: \mathtt{D} \,\}\Big)$$
   $$\times \{\, [a \doteq \mathtt{C} \mid (\mathtt{a} \doteq \mathtt{C}) \in \mathtt{Eq}] \,\}$$

3. Application of the following *subst* rule for every $Eq' \in Eq^i_{set}$
   $$\text{(subst)}\quad \frac{Eq' \cup \{\, a \doteq \mathtt{C} \,\}}{Eq'[a \mapsto \mathtt{C}] \cup \{\, \mathtt{a} \doteq \mathtt{C} \,\}}$$

4. a) Foreach $Eq' \in Eq^i_{set}$ which has changed in the last step start again with the first step.
   b) Build the union Eq'^i_{set} of all results of (a) and all $Eq' \in Eq^i_{set}$ which has not changed in the last step.

5. *Uni* $=$ $\{\, \sigma \mid Eq'' \in Eq''_{set}, Eq''$ is in solved form,
   $\sigma = \{\, a \mapsto \mathtt{C} \mid (\mathtt{a} \doteq \mathtt{C}) \in \mathtt{Eq}'' \,\}$
   $\cup \{\, a \mapsto \mathtt{Object} \mid a \in Eq''$
   and a is an isolated type variable $\}\,\}$

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

Definition 4.2 (Solved form)**.** A set of constraints Eq has reached solved form if

$$Eq = \{\, a_1 \doteq \mathtt{C}_1, \ldots, \mathtt{a}_n \doteq \mathtt{C}_n \,\} \cup \{\, \mathtt{b}_1 \lessdot \mathtt{b}_2, \mathtt{b}_3 \doteq \mathtt{b}_4, \ldots \}$$

where a, b are type variables and $\mathtt{C}$ are types. Additionally the type variables $b_1, \ldots b_n$ must be isolated type variables.

4.4 Apply solution

If the **Unify** function does not give atleast one type solution the input program is erroneous and our algorithm will fail. In any other case we take one of the type unifiers from the **Unify** result Uni.

With any $\sigma \in Uni$ we replace every type placeholder in the input program by applying the following rules to every method, constructor and field:

$$\frac{\mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}}{\sigma(\mathtt{T})\ \mathtt{m}(\sigma(\overline{\mathtt{T}})\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}}$$

$$\frac{\mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}}{\mathtt{C}(\sigma(\overline{\mathtt{T}})\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}}$$

$$\frac{\mathtt{class\ C\ extends\ C}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}});\ \mathtt{K}\ \overline{\mathtt{M}}\}}{\mathtt{class\ C\ extends\ C}\ \{\sigma(\overline{\mathtt{T}})\ \overline{\mathtt{f}});\ \mathtt{K}\ \overline{\mathtt{M}}\}}$$

After those substitutions all type variables are replaced with types. After this step the TFJ input program is a syntactically correct FJ program.

5 Proof

In this chapter we show two things the soundness and completeness of our algorithm.

We start by showing that the constraints generated by the **FJTYPE** algorithm exactly depict the typing rules of Featherweight Java. Afterwards we proof soundness and completeness of the **Unify** algorithm.

Theorem 1. *The **FJTYPE** algorithm generates constraints which are equal to the FJ typing rules.*

Proof: We will show this by comparing the generated constraints with the FJ type rules.

FJCLASS: The **FJCLASS** function just calls the **TYPEMethod** function for every method in the input program.

Afterwards it just accumulates all the constraints generated by **TYPEMethod**. No constraint is added or deleted in this step.

TYPEMethod: This function calls the **TYPEExpr** function on the return expression of the method. It adds only one constraint to the constraint set. The rest is generated by the **TYPEExpr** function. This constraint is in line with the `T-METHOD` typing rule of FJ.

TYPEExpr:

T-FIELD Every field access `e.f` generates the following constraints:

$$\texttt{OrConstraint}(\{\{rty \doteq cl, a \doteq t_f\} \mid \text{for every field } \texttt{f}\})$$

rty is the type of the expression `e`, which is most likely a type variable, because we do not know the type of `e` yet. So for every existing field with the same name as `f` we generate one constraint set inside the **OrConstraint**. It turns out later in the **Unify** algorithm which one of those possibilities is correct. One can only be correct if the solution found by the **Unify** algorithm complies with both constraints $\{rty \doteq cl, a \doteq t_f\}$. The $(rty \doteq cl)$ makes sure the $e_0 : C_0$ *fields*$(C_0) = \overline{C}\ \overline{f}$ condition is fulfilled. The $(a \doteq t_f)$ is there to assign $e_0.f_i$ to the type C_i.

For the case that there is no known field `f` in any class then the **OrConstraint** stays empty. This causes the **Unify** algorithm to fail.

T-INVK, T-NEW For method an constructor call expressions it is the same as for the field access. Take a method call $\texttt{e.m}(\overline{\texttt{e}})$ for example. From the induction hypothesis we can derive that the types for `e` and $\overline{\texttt{e}}$ are correct. So we can say that the type of `e` has to inherit a method `m` and the types of $\overline{\texttt{e}}$ are subtypes of the parameter types of this method.

T-UCAST, T-DCAST, T-SCAST Featherweight Java allows all types of casts with the `T-SCAST` rule. Therefore we do not need to generate constraints for casts.

T-METHOD The `T-Method` rule is always fulfilled when a type solution is found by the **FJTYPE** function. During the first step of the type inference algorithm all the overriden methods are given the same types and type variables. This implies same types for each overriden method, so the `T-METHOD` rule is satisfied.

T-CLASS This rule is also satisfied if a solution is found despite the way the type variables are placed during the assumption generation step. Every field of a class that overrides a field of its super class is given the same type variable as the overriden field.

Every FJ type rule is covered by the generated constraints and there are no additional constraints generated. This shows that the constraints represent the typing rules.
□

5.1 Unify proof

Theorem 2. ***(Soundness):*** *If the* ***Unify*** *algorithm finds a solution it does not contradict any of the input constraints:* $\nexists(a \lessdot b) \in Cons_{in}$ *where* $\sigma(a) \not\leq \sigma(b)$

Proof: We show theorem 2 by going backwards over every step of the algorithm. We assume there exists a unifier $\sigma = \{\, a_1 \mapsto \texttt{C}_1, \ldots, \texttt{a}_\texttt{n} \mapsto \texttt{C}_\texttt{n} \,\}$ for the input constraints, which is the result of the **Unify** algorithm. This means for every constraint in the input set $(a \lessdot b) \in Cons_{in}$ and $(c \doteq d) \in Cons_{in}$ this unifier will substitute all variables in a way that all constraints are satisfied: $\sigma(a) <: \sigma(b), \sigma(c) = \sigma(d)$

We now look at each step of the **Unify** algorithm.

Step 5 The last step of the algorithm transforms a set of constraints Eq of the the form

$$Eq = \{\, a_1 \doteq \texttt{C}_1, \ldots, \texttt{a}_\texttt{n} \doteq \texttt{C}_\texttt{n} \,\} \cup \{\, \texttt{b}_1 \lessdot \texttt{c}_1, \texttt{b}_2 \doteq \texttt{c}_2, \ldots \}$$

to the unifier σ.

Now we have to show that the resulting unifier σ is correct for the Eq set.

For the sake of simplicity we split this step into two parts:

1. $\{\, a_1 \doteq \texttt{C}_1, \ldots, \texttt{a}_\texttt{n} \doteq \texttt{C}_\texttt{n} \,\}$ to unifiers: $\{\, a_1 \to \texttt{C}_1, \ldots, \texttt{a}_\texttt{n}\texttt{C}_\texttt{n} \,\}$
2. $\{\, b_1 \lessdot b_2, b_3 \doteq b_4, \ldots \}$ to unifiers: $\{\, b_1 \mapsto \texttt{Object}, b_2 \mapsto \texttt{Object}, \ldots \}$

We look at each part as if it would be a seperate step.

It is easy to see that the unifiers generated in the first step are correct for the part of Eq

which contains constraints of the form: $\{ a_1 \doteq \mathtt{C_1}, \ldots, \mathtt{a_n} \doteq \mathtt{C_n} \}$

Substituting all types with `Object` is a correct unifier for the set $\{ b_1 \lessdot b_2, b_3 \doteq b_4, \ldots \}$, because there are only type variables in that set which why every constraint gets replaced by `Object` for each side:
$\mathtt{Oject} \lessdot \mathtt{Oject}$, $\mathtt{Oject} \doteq \mathtt{Oject}$

These two steps also generate two distinct sets of unifiers based on definitions 4.1 and 4.2.

Step 4 Neither the constraint sets nor the unifier are altered in this step.

Step 3 An unifier σ that is correct for a constraint set $Eq[a \rightarrow \mathtt{C}] \cup (\mathtt{a} \doteq \mathtt{C})$ is also correct for the set $Eq \cup (a \doteq \mathtt{C})$. From the constraint $(a \doteq \mathtt{C})$ it follows that $\sigma(a) = \mathtt{C}$. This means that $\sigma(Eq) = \sigma(Eq[a \rightarrow \mathtt{C}])$, because every occurence of a in Eq will be raplaced by `C` anyways when using the unifier σ.

Step 2 This step transforms constraints of the form $(C \lessdot a)$ and $(a \lessdot C)$ into sets of constraints and builds the cartesian product with the remaining constraints. We can show that if there is a resulting set of constraints which has σ as its correct unifier then σ also has to be a correct unifier for the constraints before this transformation. Proof:

$(a \lessdot C)$ If σ is a correct unifier for a set containing $(a \doteq \mathtt{C})$ and $\mathtt{C} \leq \mathtt{C}$, then σ is also a correct unifier for the set containing $(a \lessdot C)$.

$(C \lessdot a)$ Same goes the other way. If $Cleq\mathtt{C}$ and σ is correct for $(C \lessdot \mathtt{C})$ then σ is also correct for $(a \doteq \mathtt{C})$

$$\frac{\mathtt{class}\ C\overline{<X \lhd N>} \lhd N\ \{\ldots\}}{\triangle \vdash C\overline{<T>} <: [\overline{T}/\overline{X}]N}$$

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

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

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

□

Theorem 3. ***(Completeness):*** *If there is a solution for the input constraints $Cons_{in}$, the **Unify** algorithm will not fail. A solution is a non-empty set of unifiers $Uni = \{ \sigma_1, \ldots \sigma_n \}$, where each unifier is a injective function which maps every type variable in the input constraints $Cons_{in}$ to a existing class type in the input.*

Proof: There only can be a solution, if for each variable $a \in Cons_{in}$ there exists a substitution $[a \rightarrow \mathtt{C}]$ which fullfills all of the input Constraints $Cons_{in}$.

We show this by structural induction. If there is a possible solution for the input set, no step of the algorithm will remove the possible solution.

The **first step** applies the three rules from figure 4:
erase-rules: The erase2 rule removes a $\{\mathtt{C} \lessdot \mathtt{C}\}$ constraint from the constraint set. The erase1 rule removes a $\{\mathtt{C} \lessdot \mathtt{D}\}$ constraint, but only if the two types `C` and `D` satisfy the constraint. Both rules do not change the set of possible solutions for the given constraint set.

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

The second step of the algorithm eliminates $\lessdot$-constraints by replacing them with $\doteq$-constraints. The algorithm considers every possible type which does not violate the eliminated $\lessdot$ -constraint itself. This step does not remove a solution from the constraint set.

In the third step the **substitution**-rule is applied. If there is a constraint $a \doteq \mathtt{C}$ then there is no other way to fulfill the constraint set than replacing a with `C`. This does not remove a possible solution.

Step 4: In the fourth step the constraints are not altered.

Step 5: In step five all constraint sets that have a unifier are in solved form. All other possibilities are eliminated in steps 1-4. There are 8 different variations of constraints:

$(a \doteq a), (a \doteq C), (C \doteq a), (C \doteq C), (a \lessdot a), (a \lessdot C), (C \lessdot a), (C \lessdot C)$

After step 1 there are no $(C \doteq C)$, $(C \lessdot C)$ and $(C \doteq a)$ constraints anymore, as long as the constraint set has a correct unifier. Because a constraint set that has a correct unifier cannot contain constraints of the form $\mathtt{C}_1 \doteq \mathtt{C}_2$ with $\mathtt{C}_1 \neq \mathtt{C}_2$ and $(\mathtt{C} \lessdot \mathtt{D})$ with $(\mathtt{C} \not<: \mathtt{D})$. By removing $(\mathtt{C} \doteq \mathtt{C})$ and $(\mathtt{C} \lessdot \mathtt{D})$ with $(\mathtt{C} \not<: \mathtt{D})$, no constraints of the form $(C \doteq C)$ and $(C \lessdot C)$ remain in a constraint set that has a correct unifier after step 1.

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

After step 3 there are no $(a \doteq C)$ anymore, because we only reach step 5 if the constraint set is not changed by the substitution (step 3).

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

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

Defintion 1. ***Type solution*** *A type solution for a TFJ program is a assignment of existing types to all omitted types ($T = \epsilon$). Applying a correct type assignment turns the TFJ program into a correct FJ program, which complies with the FJ typing rules.*

Theorem 4. ***Soundness:*** *If the **FJTYPE** algorithm finds a type assignment, this assignment is a correct type solution.*

Theorem 5. ***Completeness:*** *If there exists a correct type solution for an input, the **FJTYPE** algorithm will find it.*

Proof:

A FJ program is valid if every class satisfies the `T-CLASS` rule. A method `m` is `OK IN C` if it complies with the `T-METHOD` rule.

We showed that the **Unify** algorithm is sound (theorem 2) and complete (theorem 3) and we showed that the **FJTYPE** algorithm generates constraints which comply with the FJ typing rules.

If there is a correct solution the algorithm will find it and replace all omitting types with correct types (completeness). Also the algorithm will never generate an output that is wrong (soundness). □

6 Related Work

In some object-oriented languages like Scala, C# and also Java type inference is included. But it is only local type inference [11, 12]. Local type inference means that missing type annotations are recovered using only information from adjacent nodes in the syntax tree without long distance constraints such as unification variables. E.g. the types of variables which a non-functional expression is assigned to or a return type of a method can be inferred. But the argument types of arbitrary especially recursive methods cannot be inferred by local type inference.

The base of many global type inference algorithms is the algorithm $\mathcal{W}$ that was presented by Damas and Milner [5]. The fundamental idea of the algorithm is to determine types by many-sorted type term unification [10, 16] where the unification is called for any application in the functional program. This is an efficient way to infer the types as many-sorted unification is unitary which means that there is at most one most general result. In [15] we adopted the Milner's approach directly to Java. This means that at any method application in the Java program the finitary unification is called which means that the result sets are multiplied in many cases. In [14] we change the approach. There type constraints are collected by parsing the abstract syntax tree. After that the constraints are unified.

The type unification problem which is addressed in Section 4.3 is well-known from polymorphic order-sorted unification which is used in logic programming languages with polymorphically order-sorted types [1, 7, 8, 17].

Featherweight Java and Featherweight Generic Java are defined in [9]. In this paper many properties of the languages are proved. Theses properties are assumptions for our extension with type inference. While Featherweight Java corresponds to the original Java, Featherweight Generic Java corresponds to Generic–Java [3, 4]. In [18] Featherweight Generic Java is extended by wildcards. The calculus corresponds to Java 5.0. Finally, in [2] the lambda expressions are added to Featherweight Java. This calculus corresponds to Java 8.

7 Conclusion and Outlook

We extended the calculus of Featherweight Java by global type inference to Typeless Featherweight Java. Global type inference means that all type annotations can be left out and the compiler infers the correct types. This means the property of static typing is preserved in TFJ. The type inference algorithm collects first type informations and constructs type constraints. These constraints are solved by the type unification algorithm. We proved soundness and completeness of the type inference which are reduced to soundness and completeness of the type unification algorithm.

We plan to extend Typeless Featherweight Java by adding generics, wildcards, and lambda expressions, such that we get an extended‘ calculus of [2] with type inference. This would be a calculus that correspond to Java-TX [13].

References

[1] Christoph Beierle. “Type Inferencing for Polymorphic Order-Sorted Logic Programs”. In: *International Conference on Logic Programming*. 1995, pp. 765–779.

[2] Lorenzo Bettini et al. “Java & Lambda: A Featherweight Story”. In: *Logical Methods in Computer Science* 14(3:17) (2018), pp. 1–24.

[3] Gilad Bracha et al. *GJ Specification*. 1998.

[4] Gilad Bracha et al. *GJ: Extending the Java Programming Language with type parameters*. 1998.

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

[6] James Gosling et al. *The Java® Language Specification*. Java SE 8. The Java series. Addison-Wesley, 2014.

[7] Michael Hanus. “Parametric order-sorted types in logic programming”. In: *Proc. TAPSOFT 1991* LNCS.394 (1991), pp. 181–200.

[8] Patricia M. Hill and Rodney W. Topor. “A Semantics for Typed Logic Programs”. In: *Types in Logic Programming*. Ed. by Frank Pfenning. MIT Press, 1992, pp. 1–62.

[9] 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.

[10] A. Martelli and U. Montanari. “An Efficient Unification Algorithm”. In: *ACM Transactions on Programming Languages and Systems* 4 (1982), pp. 258–282.

[11] 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.

[12] 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.

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

[14] 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*. Lecture Notes in Computer Science 8974 (2015). Ed. by Andrei Voronkov and Irina Virbitskaite, pp. 248–256.

[15] Martin Plümicke. “Typeless Programming in Java 5.0 with Wildcards”. In: *5th International Conference on Principles and Practices of Programming in Java*. ACM International Conference Proceeding Series 272 (Sept. 2007). Ed. by Vasco Amaral et al., pp. 73–82.

[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] Gert Smolka. “Logic Programming over Polymorphically Order-Sorted Types”. PhD thesis. Kaiserslautern, Germany: Department Informatik, University of Kaiserslautern, May 1989.

[18] 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/.

Taming Wildcards
Explaining Java Wildcards with Existential Types

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

Abstract

Java introduced Wildcards as an improvement to their type system. Wildcards enable more precise type annotations and lead to more type safety. It is widely used, especially in library functions like the Java standard library. But behind the scenes of Java type checking hides alot of complexity concerning Wildcards. Things like capture conversion, free type variables, and type environments. This abstract explains Java Wildcards in depth and shows some interesting edgecases.

1 Motivation

Wildcards enable more precise type annotations and lead to more type safety. It is widely used, especially in library function like the Java standard library. But this addition to the Java type system also adds a new layer of complexity. Which is not only a problem for the Java type check, but also for our type inference algorithm. Our Java-TX project focuses on developing a global type inference algorithm for Java. A global type inference algorithm is able to determine a correct type solution even when there are no type annotations at all.

2 Introduction to Wildcards

Java Generics are invariant by default. A `List<String>` is not a subtype of `List<Object>` even though it seems intuitive with `String` being a subtype of `Object`.

In the example in Figure 1 the two variables `listOfString` and `listOfObjects` would point to the same object, a `List<String>`. The Java type system prevents this by calling a type error on the assignment: *List<String> cannot be converted to List<Object>*. Java needs to prevent this, because otherwise it would be possible to add a `Integer` to a list of `String` (last line in 1).

```
List<String> listOfStrings = ...
List<Object> listOfObjects = ...

// Type Error:
listOfObjects = listOfStrings;

listOfObjects.add(1);
```

Figure 1: Java type parameters are invariant

Java wildcards add use-site variance to type parameters [4]. They act as a placeholder for an unknown type. `List<?>` is a list with unknown contents. It is a supertype of every list. So `List<String>` and `List<Integer>` are both subtypes of `List<?>`. This is called use-site variance because there is no instantiation of the type `List<?>`. It is not possible to create an instance of a wildcard list: the statement `new List<?>()` causes an error! Java wildcards affect how a class can be used. See figure 2 for example: it is still possible get elements out of a `List<?>`. The returned type is just unknown and we have to as-

```
List<String> listOfStrings = ...
List<?> listOfObjects = ...

// Correct:
listOfObjects = listOfStrings;

Object item = listOfObjects.get();
listOfObjects.add(1); // Error!
```

Figure 2: Wildcard Example

sume `Object`, which is the parent type of every Java class. On the other hand it is not possible to add elements to that list.

In this article we try to strip Java wildcards down to their simplest core. In Java wildcard declarations come with either a upper or a lower bound by using `? extends` or `? super`. We only use the `?` wildcard, which is shorthand for `? extends Object`. Also we will use existential types to explain wildcard behavior [3]. Instead of `Box<?>` we will write $\exists$`X.Box<X>`. This notation gives wildcards a name (here `X`) and also shows to which type the wildcard is bound. Wildcards in type declarations are bound to their enclosing type. `Box<?>` means $\exists$`X.Box<X>`, and `Box<Box<?>>` is `Box<`$\exists$`X.Box<X>>`.

3 Capture Conversion

Wildcard types have to be *opened* when used in a method invocation [2]. This happens implicitly in a process called *Capture Conversion*.

The usage of wildcards as generic method parameters is restricted. Given a method with the signature<A> void cpy(Box<A> b) it is clear that this method can be invoked with any box, even with `Box<?>`. Method `cpy` takes any box disregarding its content. But the method `<A> void swap(Box<A> a, Box<A> b)` takes two boxes with the same content.

```
<A> void swap(Box<A> a, Box<A> b){
 // swaps content of boxes a and b
}

Box<String> a = ...;
Box<String> b = ...;
swap(a, b); //valid call
```

```
class List<A> {
 void add(A e){ ... }
 A get() { ... }
}
```

Figure 3: Snippet of the List class definition

Here the two Strings inside boxes `a` and `b` will be swapped. This method cannot be called with a wildcard box:

```
Box<?> a = ...;
swap(a, a); //type error
```

This is rightfully a type error, because the content of the box `a` could change during the method invocation - considering `a` is a mutable field and the program is running in a multithreaded environment. Remember that `a` is just a pointer to a `Box<?>`. It could point either to a `Box<String>` or a `Box<Integer>` at any point in time. (A wildcard box `Box<?>` does not actually exists. There is no instance `new Box<?>()` possible in Java)

This fact is utilized by Capture Conversion. The moment a value is passed to a method it won't change its type for the remainder of the method execution. Capture Conversion replaces all top-level wildcards by fresh type variables.

Lets have a look at the example in figure 6. When calling `add` on a class of type $\exists$`X.List2D<X>` the following information is given:

- `list2D` : $\exists$`X.List2D<X>`
- $\exists$`X.List2D<X>` $<:$ $\exists$`X.List<List<X>>`
- `List<A>.add` : `A` $\rightarrow$ `void`

To determine the parameter type of the call to `list2D.add` we have to apply **Capture Conversion:**
$\exists$`X.List<List<X>>` $\Longrightarrow$ `List<List<Y>>`
resulting in:

`list2D.add` : `List<Y>` $\rightarrow$ `void`

The type `Y` is a fresh type variable, with the unique name `Y`. Here it functions as a type parameter to the `List` type. There cannot exists a subtype to `List<Y>`, because Java Generics are invariant when it comes to subtyping. The fresh type variable is unique and only exists once for this specific method call and the only type equal to this variable is itself. A method call to `list2D.add` trying to add a $\exists$`X.List<X>` is not type correct.

```
class List2D<A> extends List<List<A>> {}
```

Figure 4: List2D class declaration

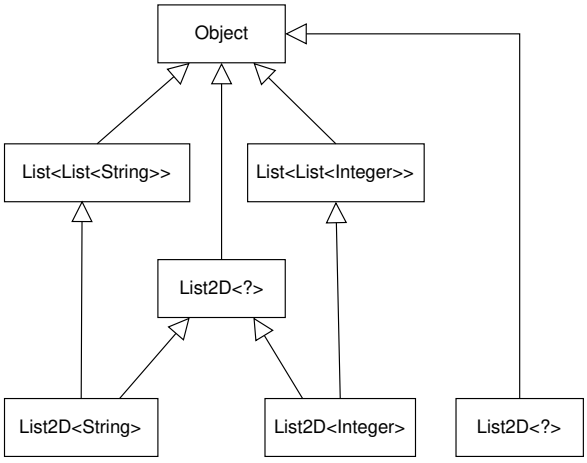


Figure 5: Subtype Graph of `List2D`

There are some special cases where it is safe to assume that two wildcards are definitely the same type.

```
class Pair<A,A> {}

<A> Pair<A,A> make(List<A> l) {...}
<A> void swap(Pair<A, A> p) {...}
```

The `swap` method is not callable with wildcards:

```
Pair<?,?> pw = ...;

swap(pw); \\ type error!
```

But the following code snippet is type correct in Java:

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

swap( make(l) ); // valid
```

The `make` call in this example produces a `Pair<?,?>` output. But under the hood this is treated as $\exists$`X.Pair<X,X>`. The variable `l` with type $\exists$`X.List<X>` is capture converted to `List<Y>` when used as a method parameter in the `make` call. Returning a `Pair<Y,Y>` type.

4 Java Wildcards

This chapter showcases interesting behaviour of Java Wildcards regarding subtyping. Imagine we define a grid of values (aka a two dimensional list `List2D`) like the one in figure 4. An immediate supertype of this type according to [1] would be: $\exists$`X.List2D<X>` $<:$ $\exists$`X.List<List<X>>`. It seems like `List<List<?>>` should be a supertype of `List2D<?>`. Unexpectetly `List2D<?>` is not a subtype of `List<List<?>>` (`List<`$\exists$`X.List<X>>`), because in the latter type the wildcard is bound to the inner `List`. $\exists$`X.List<List<X>>` $\not<:$ `List<`$\exists$`X.List<X>>` This example also shows why we use existential types to express Java Wildcards. The wildcard `X` is bound to the outer `List`. This is not expressable with the standard Java notation.

Explanation: Imagine `List2D<?>` as a grid where all the elements inside are the same type. So for example a `List2D<?>` could look like this:

Note that every List in this List of Lists has the same type - `List<String>`. Calling the following scramble method with a `List2D<?>` is possible. The method does not care about the type hold by the inner list, as long as all the lists contain the same type.

```
<A> void scramble(List<List<A>> p){
 // randomly interchange list elements
}

List2D<?> bl = ...;
scramble(bb); // valid call
```

A `List<List<?>>` on the other hand can contain:

There is no restriction about the inner lists containing all the same type This is possible because all the inner lists are a subtype of `List<?>`. Calling the `scramble` method with such a list as input would lead to type errors, because elements of different lists would be interchanged.

```
List<List<?>> b = ...;
scramble(b); // type error
```

```
List<List<?>> listWild = ...;
listWild.add(new List<String>()); //valid
listWild.add(new List<Object>()); //valid

List2D<?> list2D = ...;
list2D.add(new List<String>()); //error!

// get calls are possible:
List<?> inner = list2D.get(0);
```

Figure 6: List examples

The difference also gets clear, when we look at how adding and getting elements from those lists is handled (figure 6).

5 Conclusion

Wildcards show some unintuitive subtype behavior as explained in chapter 4. The type `List2D<?>` is not a subtype of `List<List<?>>`, whereas `List2D<String>` is a subtype of `List<List<String>>` (see figure 5).

Wildcards are opaque and it is impossible to determine during compilation what the actual type is. Capture conversion is needed to implicitly *open* wildcard types during method invocation, which enables them to be used as method parameters. Replacing wildcards with free variables also offers additional capabilities for nested method calls.

Wildcard subtyping and capture conversion can only be explained by viewing wildcards as existential types. For example a type like $\exists$`X.Pair<X,X>` can emerge inbetween method calls or $\exists$`X.List<List<X>>` as a supertype of $\exists X$`.List2D<X>`. Both types are not describable with Java wildcard syntax (using `? extends ..` notation).

References

[1] Kevin Bierhoff. “Wildcards Need Witness Protection”. In: Proc. ACM Program. Lang. 6.OOPSLA2 (Oct. 2022). URL: https://doi.org/10.1145/3563301.

[2] 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.

[3] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. “Towards an existential types model for Java wildcards”. In: Formal Techniques for Java-like Programs (2007).

[4] Mads Torgersen et al. “Adding wildcards to the Java programming language”. In: 2004 ACM Symposium on Applied Computing. SAC ’04. Nicosia, Cyprus: Association for Computing Machinery, 2004, pp. 1289–1296. ISBN: 1581138121. URL: https://doi.org/10.1145/967900.968162.

Principal set of generated generics in Java-TX

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

Abstract

Java-TX (TX standing for **T**ype e**X**tended) is an extension of Java. The predominant new features are global type inference and real function types for lambda expressions.
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

Since version 1.5 the programming language Java has been extended by incorporating many features from functional programming languages. Version 1.5 saw the introduction of generics. Generics are known as parametric polymorphism in functional programming languages. In contrast to functional programming languages such as Haskell or SML, object-oriented languages like Java allow subtyping and states of objects. Therefore, the variance of type arguments has to be considered. In PIZZA, the first approach of parametric polymorphism in Java-like languages, the arguments were declared as invariant, which means for `Vector<T>` $\leq^*$[1] `Collection<T>` and `Integer` $\leq^*$ `Object` holds

`Vector<Integer>` $\leq^*$ `Collection<Integer>`

but neither

`Vector<Integer>` $\leq^*$ `Collection<Object>`
(covariance)

nor

`Vector<Object>` $\leq^*$ `Collection<Integer>`
(contravariance)

[1] $\leq^*$ stands for the subtyping relation.

is correct. Invariance of type arguments is a strict restriction. Therefore, use-site variance by so-called wildcards was introduced in Java 5.0. In some cases, wildcards allow covariance or contravariance, respectively. In Java 8 lambda expressions were introduced, but not real function types. The types of lambda expressions are defined as target types, which are functional interfaces (essentially interfaces with one method).

Local type inference was introduced in the versions five, seven, and ten. In Java 5.0 the automatic determination of parameter instance was introduced. In Java 7 the diamond operator was introduced. In Java 10, finally, the `var` keyword for types of local variables was introduced [1].

We extended this approach by real function types in Scala-similar style [5] and a global type inference algorithm [4]. We call our extensions Java-TX. The type inference algorithm is divided in two parts. First in an recursive run over the abstract syntax tree the set of constraints for the types are collected. After that in a second step these constraints are solved by the type unification [3, 6].

The result of the type unification is given as a set of pairs of

- remaining constraints consisting only of pairs of type variables and a
- most general unifier.

In this paper we consider the remaining constraints of type variables. The constraints are transferred to bounded type parameters of classes and methods, respectively. These type parameters are called *generated generics*.

The paper is structured as follows: We start with a short overview of the type inference algorithm including the type unification (Sec. 2). Then, in Section 3 we consider the ideas how to deals with remaining constraints. After that, in Section 4 we divide the set of remaining constraints into a family where we map the types variables to the corresponding methods and the class, respectively. In Section 5 we adapt the relation of type variables to a java-conform relation. Finally, in Section 6 we present some further simplifications of the relation of type variables. We close the paper with a conclusion and an outlook.

2 Type inference algorithm

Here we would like to present a short overview of the type inference algorithm. The input is the abstract syntax tree of the corresponding Java class. The type inference algorithm consists of two steps:

Tree traversing: In a traversing of the abstract syntax tree, a type is mapped to each node of the methods' statements and expressions. If the corresponding types are left out, a type variable is mapped as type placeholder. Otherwise, the known type is mapped.

During the traversing a set of type constraints $\{\overline{ty \lessdot ty'}\}$ is generated. The constraints represent the type conditions as defined in the Java specification [2]. For more details see the function **TYPE** in [4].

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

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

The result of the type unification is a set of pairs

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

where $\{\overline{(T \lessdot T')}\}$ is a set of remaining constraints consisting of two type variables and σ is a most general unifier.

The type unification algorithm is given in [3, 6]. There we proved that the unification is indeed not unitary, but finitary, meaning that there are finitely many most general unifiers.

Let us consider as an example the application of the type inference algorithm to the faculty function.

Example 1. *First, we present the essential type variables which are mapped to nodes of the method* `getFac`*:*

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

The generated constraints are:

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

The result of type unification is given as:

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

In this example, no constraints consisting only of type variables remain. Furthermore, there is only one most general unifier.

If we instantiate the type variables by the determined types, we get:

```
class Fac {
  Integer getFac(Integer n) {
    Integer res = 1;
    Integer i = 1;
    while((
        i::Integer) <= (n::Integer)
      )::Integer  {

      (res::Integer) =
((res::Integer) * (i::Integer))::Integer;
      i++;
    }

    return(res::Integer);
  }
}
```

The unification in the above example has one solution. If there was more than one solution, there would be more than one principal typing of the method.

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

3 Generalized type variables

In a similar way as in type inference of functional programming languages, free type variables which are not instanced by other types after type inference are generalized to generics. In comparison to functional programming languages, in Java subtyping leads to a more powerful generalization mechanism.

Keeping in mind the result of the type unification (Sec. 2) given as a set of pairs of

- remaining constraints consisting only of pairs of type variables and a
- most general unifier:

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

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

Following up this idea, the remaining type variables become bounded type parameters of the class and its methods, respectively, where the left-hand side of a constraint is a type parameter and the right-hand side is its bound.

There are three possibilities for adapting this concept to Java:

- All in the constraints, connected type variables are mapped to one type parameter.
- All constraints are transferred to bounded type parameters of the class.
- The constraints are transferred to bounded type parameters of the class and its methods.

We shall see that the third possibility leads to the principal types. Additionally, due to the Java restrictions of type parameters, some type parameters have to be collected to one new type parameter. As proposed in the first possibility.

This section is structured as follows: After a motivating example, we present an apportioning of the type variables to the class and its methods. We then reduce the respective set of type variables such that the Java restrictions of type variables are fulfilled. Finally, we offer two possibilities for reducing the set of type variables without losing the principal type property.

Let us start with a motivating example.

Example 2. *On the left in Fig. 1 a Java-TX program is given. The identity function is mapped to the field* `id`*. In the method* `id2` *the identity function is called. In the method* `m` *the method* `m2` *is called.*

The application of the type inference algorithm is presented on the right, and the remaining set of constraints of the type unification is:

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

4 Family of generated generics

We divide up the set of remaining constraints cs by transferring it to a family cs' where the index set is given as the class name and its method names.

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

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

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

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

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

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

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

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

Figure 1: Class `TPHsAndGenerics` before an alter type inference

Definition 4.1 (Family of generated generics)**.** *The* family of generated generics *is defined as*

$$cs' = (cs'_{in})_{in \in CLM},$$

where

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

is the index set of the class name and its methods' names.

Let cs be a set of remaining constraints as result of the type unification. cs is transferred to cs' where, the set of generated generics of the class cs'_{cl} are given as:

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

The set generated generics cs'_m of its methods m: are given, respectively, as:

- *the type variables of the method m with its bounds, where the bounds are also type variables of the method,*
- *new constructed pairs $T_1 \lessdot T_2$ of type variable T_1, T_2 of the method m which are in the transitive closure*

$$T_1 \lessdot R_1 \underline{\leq}^* R_2 \lessdot T_2$$[2]

where $R_1 \lessdot R_2 \in cl'_{m'}$ and m' is called in m,

- *all type variable of the method m with its bounds, where the bounds are type variables of fields and*
- *all unbounded type variables of the method m and all unbounded bounds with `Object` as bound.*

We present the family of generated generics for the class `TPHsAndGenerics` from Example 2

Example 3. *The set of remaining constraints*

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

of the class `TPHsAndGenerics` results in the the family of generated generics

The set of generated generics $cs'_{\texttt{TPHsAndGenerics}}$ of the class:

- *Type variables of the fields with its bounds:*

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

- *Closure of all bounds of type variables of the fields with its bounds:*

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

- *All unbounded type variables of the fields and all unbounded bounds with `Object` as bound:*

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

The set of generated generics cs'_{m2} of the methods `m2`:

- *All unbounded type variables of the method m with `Object` as bound:*

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

[2] $\underline{\leq}^*$ stands for the reflexive and transitive closure of $\lessdot$.

The set of generated generics cs'_m ***of the methods*** `m`***:***

- *New pair* $\{\,\texttt{AD} \lessdot \texttt{AE}\,\}$ *which is in the transitive closure* $\texttt{AD} \lessdot \texttt{AI} \lessdot \texttt{AE}$, *where* $\texttt{AI} \in cl'_{\texttt{m2}}$ *and* `m2` *is called in* `m`.
- *All unbounded type variables of the method* `m` *and all un-bounded bounds with* `Object` *as bound:*

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

The set of generated generics cs'_{id2} ***of the methods*** `id2`***:***

- *All pairs where the bounds are type variables of fields:*

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

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

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

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

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

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

  <AI, AM> AI m2(AM a, AI b){
    return b;}
}
```

Figure 2: Generated generics of the class `TPHsAndGenerics`

5 Java-conform binary relation of type parameters

The set of remaining constraints as well as each element of the family of generated generics are arbitrary binary relations.

There are two conditions in Java which all members of the family of generated generics have to fulfill:

- The reflexive and transitive closure must be a partial ordering (the subtyping relation is partial ordering).
- Two different elements have no common infimum (multiple inheritance is prohibited).

Consider the following lemma:

Lemma 5.1. *The reflexive und transitive closure of any binary relation is a partial ordering if it contains no cycle.*

This means that we have to eliminate cycles and infima to get Java-conform binary relations. We will do this by a surjective mapping of connected type variables to a new type variable.

First, we shall consider two examples which result in non-conform relations.

Example 4 (Cycle)**.** *Let the class* `Cyle` *be given:*

```
class Cycle {
    m(x, y) {
        y = x;
        x = y;
    }
}
```

For the inferred method parameter `m(L x, M y)` *we get*

$$cs'_{\texttt{m}} = \{\,(L \lessdot M), (M \lessdot L)\,\}$$

But

```
<L extends M,M extends L>
  void m(L x, M y) {...}
```

is not a correct declaration.

Example 5 (Infimum)**.** *Let the class* `Infimum` *be given:*

```
class Infimum {
    m(x, y, z) {
        y = x;
        z = x;
    }
}
```

For the inferred method parameter `m(L x, M y, N x)` *we get*

$$cs'_{\texttt{m}} = \{\,(L \lessdot M), (L \lessdot N), (M \lessdot \texttt{Object}), (N \lessdot \texttt{Object})\,\}.$$

But

```
<L extends M, L extends N, M, N>
    void m(L x, M y, N z) { ... }
```

is not a correct declaration.

The general approach for eliminating cycles and infima is to equalize elements by a surjective map h that preserves the subtype relation: For $T \underline{\leq}^* T'$

$$h(T) \underline{\leq}^* h(T').$$

holds true.

We shall now present an algorithm which eliminate cycles and infima.

Algorithm 1 (Java-conform relation)**.**

Input: *A member of the family of generated generics C.*

Output: *An adapted member of the family of generated generics C and a surjective mapping h that describes the adaption of C*

Postcondition: *C is a minimal adaption such that it is Java conform.*

The algorithm:

1. ***Remove cycles:***

   For any $(T \lessdot K \lessdot G \lessdot ... \lessdot T)$ in C:

   - *Substitute all type variables of the cycle with a new type variable X in C.*
   - *Remove all constraints that built the cycle from C.*
   - *In h all removed type variables of the cycle are mapped to X.*

2. ***Eliminate Infima:***

   Apply the following steps until no infima are in C:

   For any $Constr_T = \{(T \lessdot R), (T \lessdot S), \dots\} \subseteq C$

   - *Create a new type variable X and create a new constraint $(T \lessdot X)$.*
   - *Add the new constraint $(T \lessdot X)$ to C.*
   - *Remove all constraints $Constr_T$ from C.*
   - *All right-hand sides R of all constraints of $Constr_T$ are mapped to X in h and substituted in C with X.*

In the following examples, we apply the algorithm to the classes `Cycle` (Example 4) and `Infimum` (Example 5).

Example 6. *Applying the algorithm to the class `Cycle` we get the surjective mapping h with*

$h(L) = X$
$h(M) = X$

and the adapted class:

```
class Cycle {
  <X> void  m(X x, X y) {
    y = x;
    x = y;
  }
}
```

Example 7. *Applying the algorithm to the class `Infimum` we get the surjective mapping h with*

$h(M) = X$
$h(N) = X$

```
class Infimum {
  <L extends X, L extends X, X>
  void m(L x, X y, X z) {
    y = x;
    z = x;
  }
}
```

6 Further simplifications

Let us consider again the generated generics in the class `TPHsAndGenerics` (Figure 2). There is a class type parameter `DZP` that is never used. Additionally, in the field `id`, the argument and the result type differ although the identity functions return the argument `x`. It is not clear whether this is necessary.

Eliminate inner type variables

In the class `TPHsAndGenerics` the type variable `DZP` occurs in the type `Fun1$$<DZP,DZP>` of the lambda expression `x -> x` as inner type variable which is generated during the tree traversing of the type inference (cp. Sec. 2). As type variables of inner nodes are not needed to be declared, we eliminate type variables of inner nodes. In this step, the properties of the partial ordering have to preserved.

Therefore, in the above example `UD` $\lessdot$ `ETX` follows from `UD` $\lessdot$ `DZP` $\lessdot$ `ETX`.

Equalize type variables in contravariant and covariant position

For the consideration of type variables like `UD` and `ETX`, we need another definition.

Definition 6.1 (Type variables in covariant and contravariant position)**.** *A type variable of an argument of a function/method is in* contravariant position*. A type variable of a return type of a function/method is in* covariant position*.*

In the type `Fun2$$<UD,ETX>` of the field `id` in the class `TPHsAndGenerics` the type variable `UD` is in contravariant position and `ETX` in covariant position.

Lemma 6.2. *Let T be a type variable in contravariant position and and U a type variable in covariant position. If $T \lessdot U$ is valid, then in the sense of principality of the program the two type variables T and U can be equalized.*

Proof. For the principality of a function type is demanded that the argument types have to be maximal and the return type has to be minimal. This means that T has to be maximal and U has to be minimal. Therefore, it follows from $T \lessdot U$ that for the principal type, $T = U$. □

In the above example, the type variables `UD` and `ETX` can be equalized. Fig. 3 , below, presents the complete simplified class `TPHsAndGenerics`. Here, the inner type variables are eliminated and type variables in contravariant and covariant position are equalized.

```
class TPHsAndGenerics<UD> {
  Fun1$$<UD, UD> id = x -> x;

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

  <AD extends AE, AB, AE>
  AB m(AB a, AD b){

    AE c = m2(a,b);
    return a;}

  <AI, AM> AI m2(AM a, AI b){
    return b;}
}
```

Figure 3: The complete simplified class `TPHsAndGenerics`

7 Conclusion and Outlook

We presented a concept for generalization for free type variables (generated generics) which is more powerful than in functional programming languages. The remaining type variables constraints of the type inference were distributed to the class and its method, respectively.

Therefore, we constructed a family of generated generics, where the remaining type variables are divided to the methods of the class and the class itself, respectively. Then the relations which are constructed in the sets of the family are transformed to Java-conform binary relations. Finally we simplified the relation, without loosing the principality.

In future the simplification process could be improved, such that not only variables in covariant and in contravariant position as described can be equalized, but also some variables which are both in the same position.

At the moment the type inference algorithm can only determine generics which are bounded by type variables. We plan extend the algorithm such that also type variables which are bounded by real types can be inferred.

References

[1] Brian Goetz. *JEP 286: Local-Variable Type Inference*. Updated: 2018/10/12 01:28. 2016. URL: http://openjdk.java.net/jeps/286.

[2] James Gosling et al. *The Java® Language Specification*. Java SE 8. The Java series. Addison-Wesley, 2014.

[3] 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*. Lecture Notes in Artificial Intelligence 5437 (2009). Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf, pp. 223–240.

[4] 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*. Lecture Notes in Computer Science 8974 (2015). Ed. by Andrei Voronkov and Irina Virbitskaite, pp. 248–256.

[5] 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. URL: http://doi.acm.org/10.1145/3132190.3132203.

[6] Florian Steurer and Martin Plümicke. “Erweiterung und Neuimplementierung der Java Typunifikation”. In: *Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts*. Ed. by Jens Knoop, Martin Steffen, and Baltasar Trancón y Widemann. Research Report 482. ISBN 978-82-7368-447-9, (in german). Faculty of Mathematics and Natural Sciences, UNIVERSITY OF OSLO. 2018, pp. 134–149.

Avoiding the Capture Conversion in Java-TX?

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

Abstract

In Java, wildcards are used to weaken the restrictions of invariant generic data-types. Unfortunately this implies some inconveniences. One of them are the so-called capture conversions which in some cases convert wildcards to type variables. In this paper we present an approach which is chosen in Java-TX to avoid capture conversion. We give examples where capture conversion induces type-error although the programs are sound. On the other hand there are some examples where named wildcards are necessary. We discuss the two approaches. Presumably, a mixture of both would be the best solution.

1 Introduction

In Java [**GoJoStBrBuSm23**] wildcards are used to enable covariance and contravariance for the invariant type constructors (generic classes). This is done, as the java type system would be unsound if the type constructors are not invariant. The following example shows the problem:

```
Vector<Integer> intVec =
    new Vector<Integer>();

//This is a not correct covariant
//use of the type constructor Vector
Vector<Object> objVec = intVec;

objVec.add(new Object());
```

As the variable `objVec` is a reference to a vector of `int`s in the last line an `Object` is added to a vector of `int`s, which is incorrect. Therefore a type system which would allow this would be unsound.

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 vector conversion looks like:

```
Vector<Integer> intVec =
    new Vector<Integer>();
Vector<? extends Object> objVec=intVec;
objVec.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 vectorAddAll(Vector<T> x,
                      Vector<T> y) {
  x.addAll(y);
}
```

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

```
Vector<?> v1 = ...
Vector<?> v2 = ...
vectorAddAll(v1, v2);
```

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

$$G\langle S_1, \ldots, S_n \rangle \text{ is a capture conversion of } G\langle T_1, \ldots, T_n \rangle$$

if

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

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

- $T_i = \texttt{?}\ \texttt{super}\ ty \Rightarrow S_i$ *is a fresh type variable with upper bound* $U_i[A_1 \mapsto S_1, ..., A_n \mapsto S_n]$ *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 `vectorAddAll(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 sufficient but not necessary.

For example in Fig. 1 the variable `v2` is assigned

```
class Assign {

    <X> void assign(Vector<X> v1,
                    Vector<X> v2) {
        v1 = v2;
    }

    void main() {
        Vector<?> v1 = null;
        Vector<?> v2 = null;
        v1 = v2;

        //not type correct but sound
        assign(v1, v2);
    }
}
```

Figure 1: The assign-example

to `v1`. 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 `v2` to `v1`.

2 The Java-TX approach

Following the sematics of the wildcard-types

`?` `extends` θ**:** *There is a subtype of* θ.

`?` `super` θ'**:** *There is a supertype of* θ'.

we continue the subtyping relation $\leq^*$ on wildcard-types.

Definition 2.1 (Continued subtyping relation). *For two given Java-types* θ, θ' *with* $\theta \leq^* \theta'$ *holds*

- $\theta \leq^*$ `?` `super` θ',
- `?` `extends` $\theta \leq^* \theta'$, *and*
- `?` `extends` $\theta \leq^*$ `?` `super` θ'.

Remark Especially, $\leq^*$ is not reflexive on wildcards

2.1 Type–Inference in Java-TX

In [**plue15_2**] it is introduced that for every (sub-)expression a different type variable (in the Java-TX environment type variables are called type placeholder) is assumed. All types are inferred during the compilation process. If the type inference infers a generic type variable for an expression the assumed type placeholder is not substituted. This means that no pairwise two (sub-)expressions have the same generic type. From this follows that for no generic type variable `T` a subtype relation `T` $\leq^*$ `T` arises during typecheck. Therefore it is no problem to instantiate wildcards into generic type variables.

The above examples would have the following typing:

```
<R, T extends R>
    void vectorAddAll(Vector<R> x,
                      Vector<T> y) {
        x.addAll(y);
}
```

Then

```
Vector<? extends Object> x1 =
                new Vector<Integer>();
Vector<? extends Object> x2 =
                new Vector<String>();
vectorAddAll(x1, x2);
```

leads to

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

and this results in

$$\mathtt{?\ extends\ Object} \lessdot \mathtt{?\ extends\ Object}$$

which leads to a type error as $\leq^*$ is not reflexive on wildcard-types.

In contrast the assign-example (Fig. 1) leads to

```
Vector<? extends Object> ⩽
                    Vector<? extends Object>
```

which is correct.

Hypothesis (Avoid capture conversion). If generics which are in subtype relationship are not equalized then capture conversion is avoidable.

In the following we will discuss the hypothesis.

The first problem arises if we consider cyclic dependencies. Let us consider the following example.

```
class Cycle {

    <L extends M, M extends L>
    void  m(L x, M y) {
        y = x;
        x = y;
    }
}
```

Figure 2: The cycle-example

The declaration of the generics `<L extends M, M extends L>` is not correct. Therefore, we have to equalize `L` and `M`.

```
    <L> void  m(L x, L y) {
        y = x;
        x = y;
    }
```

This means, we need capture conversion, as

```
Vector<? extends Object> v = ...;
m(v.get(0), v.get(1));
```

would result in

$$\mathtt{?\ extends\ Object} \lessdot \mathtt{?\ extends\ Object}$$

This means, the Java-TX type inference algorithm works without capture conversion. But for the use of the generated class-files capture conversion is needed as classes like `Cycle` (Fig. 2) are not correct Java-classes and therefore the corresponding class-files are not correct, too.

In the Java-TX approach a class-file generated by the Java-TX-compiler should can be used as a Java class-file with capture conversion as well as as a Java-TX class-file without capture conversion.

The problem is solved in Java-TX. For this we remember that the result of the type inference [**plue15_2**] is given as a set of pairs of (Remaining constraints, Substitution):

$$\begin{array}{l}\{\\ (\{T_1 \lessdot T_1' \ldots T_n \lessdot T_n'\}, [R_1 \mapsto ty_1 \ldots R_m \mapsto ty_m])\\ ,\ldots,\\ (\{T_1 \lessdot T_1' \ldots T_n \lessdot T_n'\}, [R_1 \mapsto \tilde{ty}_1 \ldots R_m \mapsto \tilde{ty}_m])\\ \}\end{array}$$

A first approach is to use the remaining constraints $T_i \lessdot T_i'$ as signature in the class-files. Applied to the `Cycle`–Example (Fig. 2) this results in Fig. 3. Unfortunately, the JVM crashes although

```
public <L extends M, M extends L>
  void m(L, M);
  descriptor:(Ljava/lang/Object;
              Ljava/lang/Object;)V
  flags: (0x0001) ACC_PUBLIC

  Code:
    stack=2, locals=3, args_size=3
    ...

  Signature: #13 //<L:M;M:L;>(L;M)V
```

Figure 3: Java bytecode with cycles

the signature is not used for the running of the class-file.

Bytecode attributes are extendable. To solve this problem we extend the bytecode for Java-TX by an attribute `JavaTXSignature`.

The `cycle`-example with `JavaTXSignature` is given in Fig. 4.

We will close this section with an example that implements the abstract data-type `List`.

Example 2.2. Let the Java-TX program in Fig. 5 be given. The usual typing in Java of `concat` would be:

```
void concat(List<A> l);
```

```
public <L extends java.lang.Object>
  void m(L, L);
  descriptor: (Ljava/lang/Object;
               Ljava/lang/Object;)V
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=3
    ...

  Signature:#13
     //<L:Ljava/lang/Object;>(L;L;)V
  JavaTXSignature:#14
     //<L:M;M:L;>(L;M)V>
```

Figure 4: Attribute `JavaTXSignature`

```
class List<A> {
  A elem;
  List<A> next;

  concat(l) {
    var nextLoc = next;
    while (nextLoc != null) {
      nextLoc = nextLoc.next;
    }
    nextLoc = l;
  }

  void m() {
    List<?> l = new List<Integer>();
    List<?> l2 = new List<String>();
    l.concat(l2);
  }
}
```

Figure 5: Class `List`

and in the method `m` for the call of `l.concat(l2)` two capture variable `CAP#1` and `CAP#2` are generated, such that `l` gets the type `List<CAP#1>` and `l2` gets the type `List<CAP#1>`. Therefore, this leads to an type-error, which is correct as no `String`-List is allowed to append to an `Integer`-List.

In Java-TX for `concat` the typing

```
void concat(List<? extends A>)
```

is inferred. Now the method-call `l.concat(l2)` leads to the constraint

$$\texttt{? extends Object} < \texttt{? extends Object}$$

which is wrong.

3 Drawbacks of avoiding capture conversion

In this section we will consider some drawbacks of the approach to avoid capture conversion in Java-TX.

3.1 Nested Supertypes

At first we consider nested supertypes. Let the class declaration

```
class Mat<T> extends Vec<Vec<T>> { ... }
```

be given.

`Vec<Vec<?>>` is a subtype of itself. Therefore, `Vec<Vec<?> vv2 = vv;` is correct. But

```
Mat<?> m = ...
Vec<Vec<?>> vv = m;
```

is not correct.

```
error: incompatible types: Mat<CAP#1>
       cannot be converted to
       Vec<Vec<?>>
    vv = m;
         ^
```

Let us consider an example to explain the problem.

```
Mat<?> m = new Mat<Integer>();
Vec<Vec<?>> vv = m;  //not correct
vv.add(new Vec<Object>());
```

The problem is that we add a vector of `Object`s to a matrix of `Integer`s. This is not sound. Therefore, `Mat<?>` is not a subtype of `Vec<Vec<?>>`.

We will consider the problem in a formal context. A wildcard `?` in an argument position of generic type `C<? extends B>` means, there is a type `X` with bound `B` such that `C<X>` ($\exists[\texttt{X} \leq^* \texttt{B}]\texttt{C<X>}$).

If we apply this approach to the above example we get:

$$\texttt{Mat<?>} = \exists[\texttt{X} \leq^* \texttt{Object}]\texttt{Mat<X>}$$

$$\leq^* [\exists \texttt{X} \leq^* \texttt{Object}]\texttt{Vec<Vec<X>>}$$

$$\not\leq^* \texttt{Vec}<\exists[\texttt{X} \leq^* \texttt{Object}]\texttt{Vec<X>>} = \texttt{Vec<Vec<?>>}$$

$[\exists \texttt{X} \leq^* \texttt{Object}]\texttt{Vec<Vec<X>>}$ means `X` has be instantiated be the same type in all instance of `Vec<X>` while $\texttt{Vec}<\exists[\texttt{X} \leq^* \texttt{Object}]\texttt{Vec<X>>}$ means

that X can be instantiated by different types in each instance of Vec<X>.

Therefore, in Java-TX a reduction during the type unification [**plue09_1**] as

$$\texttt{Mat<}ty\texttt{>} \lessdot \texttt{Vec<Vec<}ty'\texttt{>>}$$

to

$$\texttt{Vec<Vec<}ty\texttt{>>} \lessdot \texttt{Vec<Vec<}ty'\texttt{>>}$$

is only for non wildcards types ty and ty' allowed.

Somewhat surprising is that the next example

```
<X> void m(Vec<Vec<X>> v) {
  Matrix<?> mat = ...
  m(mat);
}
```

is correct. If we consider the example formal we will see, that it is correct.

$$\texttt{Mat<?>} = \exists[\texttt{X} \leq^* \texttt{Object}]\texttt{Mat<X>}$$

$$\leq^* [\exists\texttt{X} \leq^* \texttt{Object}]\texttt{Vec<Vec<X>>}$$

$$\leq^* [\forall\texttt{X} \leq^* \texttt{Object}]\texttt{Vec<Vec<X>>}$$

Unfortunatly, in Java-TX this would fail as the unification is treated as in the above case.

3.2 Duplication of variables

Let us consider the following example (Example 2 in [**CNDSEE08**]).

```
<X> Pair<X, X> make(List<X> l) {...}

<X> Boolean compare(Pair<X, X> p){...}

void m(Pair<?, ?> p, List<?> b)
{
  compare(p);        //1.type incorrect
  compare(make(b)); //2.OK
}
```

1. In the first case it holds

   $[\exists\texttt{X} \leq^* \texttt{Obj}][\exists\texttt{Y} \leq^* \texttt{Obj}]\texttt{Pair<X,Y>} \not\leq^* \texttt{Pair<X,X>}.$

   Therefore two capture variables are generated and the call is wrong.

2. In the second case the return type of make is given as: $[\forall\texttt{X} \leq^* \texttt{Object}]\texttt{Pair<X,X>}$.

   This means

   $$\texttt{make(b)} : [\exists\texttt{X} \leq^* \texttt{Object}]\texttt{Pair<X,X>}$$

   and this means

   $$[\exists\texttt{X} \leq^* \texttt{Object}]\texttt{Pair<X,X>} \leq^* \texttt{Pair<X,X>}.$$

   Therefore the call is type-correct.

In a second example for duplicating variables the program is not type-correct.

```
<X> void m(Vec<X> x, Vec<X> y) {
    Vec<?> l = ... ;
    ...
    m(l, l);
}
```

The question is, does

$$[\exists\texttt{X} \leq^* \texttt{Obj}](\texttt{Vec<X>}, \texttt{Vec<X>}) \leq^* (\texttt{Vec<X>}, \texttt{Vec<X>})$$

hold.

This is not corect as Java allows no tuple types. This means formally the type constraint would be:

$$([\exists\texttt{X} \leq^* \texttt{Obj}]\texttt{Vec<X>}, [\exists\texttt{Y} \leq^* \texttt{Obj}]\texttt{Vec<Y>}) \leq^* (\texttt{Vec<X>}, \texttt{Vec<X>}).$$

And this is wrong.

3.3 Summary

At the moment Java-TX allows the pros of Section 2. In contrast subtyping with wildcards in nested supertypes and in duplicated variables are not allowed:

- $\texttt{Mat<?>} = \exists[\texttt{X} \leq^* \texttt{Object}]\texttt{Mat<X>} \not\leq^* \texttt{Vec<}\exists[\texttt{X} \leq^* \texttt{Object}]\texttt{Vec<X>>} = \texttt{Vec<Vec<?>>}$
- $\texttt{Mat<?>} = \exists[\texttt{X} \leq^* \texttt{Object}]\texttt{Mat<X>} \not\leq^* [\forall\texttt{X} \leq^* \texttt{Object}]\texttt{Vec<Vec<X>>}$
- $[\exists\texttt{X} \leq^* \texttt{Object}]\texttt{Pair<X,X>} \not\leq^* \texttt{Pair<X,X>}$
- $[\exists\texttt{X} \leq^* \texttt{Object}](\texttt{Vec<X>}, \texttt{Vec<X>}) \not\leq^* (\texttt{Vec<X>}, \texttt{Vec<X>})$

While this is necessary for the soundness in the first and last case in the second and the third case this would not be necessary.

4 Related Work

Wildcards are first described in a research paper in [9]. In [8] 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 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 [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.

Note: Java-Compiler Error

We will close with a note of java-compiler error which we have detected. Later we notice that the error is known[1].

First we bring the following fact to our mind: If for a given generic class `C<T>` holds

$$\texttt{C<? extends } ty_1\texttt{>} \leq^* \texttt{C<? extends } ty_2\texttt{>}$$

then $ty_1 \leq^* ty_2$.

Let the classes `Mat` and `Vec` from Section 3.1 be given.

```
1  Vec<Mat<Integer>> vmInt
2              = new Vec<Mat<Integer>> ();
3  vmInt.add(new Mat<Integer>());
4  Vec<? extends Mat<?>> vm = vmInt;
5
6  // This should be an error
7  Vec<? extends Vec<Vec<?>>> vQMvv = vm;
8
9  Vec<Object> v0 = new Vec<Object>();
10 v0.add("String");
11 vQMvv.get().add(v0);
12 Vec<Integer> vInt_2 = vmInt.get().get();
13 int i = vInt_2.get() + 2;
```

[1] https://bugs.openjdk.org/browse/JDK-7045341

As `Mat<?>` is no subtype of `Vec<Vec<?>>` the assignment `vQMvv = vm;` in line 7 is not type correct. But the java-compiler generates no type-error.

In line 9 to 13 we activate a runtime type-error. In line 11 we add to a vector of vectors of integers an vector of the type `Object` containing a string. In line 12 we read this in line 11 added vector as a vector of integers. In line 13 we pick the element as an integer and add the number two. But the element is a string. This results in an runtime type-error:

```
java.lang.ClassCastException:
class java.lang.String cannot be cast to class
java.lang.Integer ...
```

5 Summary and Outlook

In this paper, we considered wildcards which are introduced into Java in version 5. For the use wildcards there are so-called capture conversions introduced which converts each wildcard into an own type variable.

We discussed in the paper if it would be possible to avoid 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 hypothesis that capture conversion can be avoided if no type variables that are in subtype 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 hypothesis. Furthermore we have to consider the programs that are correct in the capture conversion approach but not correct in our approach.

Perhaps, it would be possible to use named wildcards in these cases.

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). URL: https://doi.org/10.1145/3563301.

[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] 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*. Lecture Notes in Artificial Intelligence 5437 (2009). Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf, pp. 223–240.

[7] 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*. Lecture Notes in Computer Science 8974 (2015). Ed. by Andrei Voronkov and Irina Virbitskaite, pp. 248–256.

[8] 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/.

[9] Mads Torgersen et al. “Adding wildcards to the Java programming language”. In: *Journal of Object Technology* 3.11 (Dec. 2004), pp. 97–116.

Java ohne Wildcards

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

Zusammenfassung

Die Programmiersprache Java benutzt Wildcards für Subtyping von generischen Typen. Das ist notwendig, da beispielsweise `Vector<String>` in Java kein Subtyp von `Vector<Object>` ist. Sollen derartige Abhängigkeiten benutzt werden, ist der Einsatz von Wildcards notwendig. `Vector<String>` wäre hingegen ein Subtyp von `Vector<? extends Object>`. Das Konzept der Wildcards ist allerdings zum einen nicht einfach zu verstehen, zum anderen können Wildcards zu nur schwer verständlichen Typen und Typfehlern führen.
Das Ziel der Arbeit ist es, dem Java-Programmierer das Konzept der Wildcards zu ersparen ohne dabei die Eigenschaften von Wildcards zu verlieren. Während des Compiliervorgangs sollen mithilfe von Typinferenz fehlende Wildcards automatisch eingefügt werden. Dem Programmierer selbst werden im Vordergrund die Wildcardtypen nicht angezeigt, dennoch ist das Programm intern korrekt typisiert. Das soll vor allem die Lesbarkeit des Quellcodes erhöhen und dem Programmierern das Formulieren von komplexen Typen ersparen.
Der hierbei eingesetzte Typinferenzalgorithmus basiert auf dem des JavaTX Projekts. Wir ignorieren die im Programm gesetzten Generischen Typen und versuchen stattdessen mittels Typinferenz besser geeignete Typen zu finden. Tritt der Fall auf, dass das Programm in der ursprünglichen Variante nicht typkorrekt ist, so versuchen wir durch Benutzen von Wildcards eine korrekte Typisierung herzustellen. Ist es möglich, so kann diese kurz vor der Compilierung intern dementsprechend angepasst werden.

1 Einleitung

1.1 Ausgangssituation and Motivation

Ein häufig auftretendes Problem in der Java Entwicklung ist die zur Kompilierzeit inkorrekte Typisierung von Generics bedingt durch deren gegenseitiger Zuweisung. Ein einfaches Beispiel in Listing 1 zeigt, dass die zwei Typen `List<Object>` und `List<String>` nicht kompatibel sind, obwohl gilt `String` $\leq$ `Object`. Während des Kompilierens wird vom Java Compiler ein ähnlich gearteter Fehler, wie in Abbildung 1 gezeigt, ausgeworfen und die Kompilierung schlägt fehl.

Durch das Hinzufügen von Java Wildcards wird der in Listing 1 dargestellte Code in Listing 2 zu einem für den Standard Java Compiler typkorrekten Java Code.

Warum sollte ein Compiler einen Fehler produzieren, obwohl ein korrekter Typ existiert? Warum können die beschrieben Wildcards nicht optional sein?

Um die Wiederverwendbarkeit von Methoden mit Generischen Typparametern zu steigern, empfiehlt es sich, wie in Listing 3 und Listing 4 an den unterstrichenen Stellen gezeigt, Wildcards einzuführen.

```
1  List<Object> method (List<String> input) {
2    List<Object> listOfObjects = input;
3    Object test = listOfObjects.get(0);
4    String string = "Test";
5    input.add(string);
6    return listOfObjects;
7  }
```

Listing 1: Typinkorrekter Javacode

error: incompatible types in the statement:
`List<Object> listOfObjects =`
`listOfStrings:`
`List<String> cannot be converted to`
`List<Object>`

Abbildung 1: Fehlermeldung für typinkorrekten Javacode

Da dieses Konzept zu einer Verschlechterung der Lesbarkeit des Codes führt und nur zur korrekten Typisierung der Java Generics eingesetzt werden müssen, stellt sich die Frage, warum der Compiler diese nicht automatisch für den Programmierer hinzufügt?

1.2 Thematische Abgrenzung und Zielsetzung

Die oben aufgeführten Probleme, sind ein Teilaspekt der durch den Ansatz der gesamtheitlichen Typinferenz von Java bereits gelöst wird. Die Algorithmen aus [1] und [2] können zur Lösung eines Teilaspektes herangezogen werden. In dieser Arbeit soll sich vor allem auf eine geänderte Ausgangssituation konzentriert werden, welche von der aktuellen in den Algorithmen beachteten abweicht. So soll bereits ein getypter Code als Ausgangsartefakt existieren und nur das Konzept von Java ohne Wildcards eingeführt und darauf angewandt werden, um Java Generic Typprobleme zu lösen. Es soll vermieden werden, dass der Quellcode ganzheitlich ohne Typen vorliegen muss. Ziel ist somit, dass der Compiler nur fehlende Wildcards hinzufügt, damit das Programm typkorrekt wird. Als Beispiel können wir Listing 5 und Listing 6 betrachten. Mittels dieser Arbeit soll es möglich sein, dass diese als Typkorrekt erkannt werden. Es soll als Ergebnis Java typkorrekter Code, wie in Listing 3 und Listing 4 erzeugt werden.

2 Erweiterung des Java-TX Algorithmus

Die nötigen Erweiterungen des Java-TX Algorithmus zur Einbindung der Java ohne Wildcards Funktionalität lassen sich in verschiedene Schritte (Abbildung 2) aufteilen. Dadurch wird eine Einbettung in die bestehenden und zu adaptierende **TI** Funktion aus [1] ermöglicht. Dieses Kapitel definiert die nötigen Schritte des Algorithmus und deren Integration in eine Adaption **TI**′ der Funktion **TI**. Zum Abschluss des Kapitels findet sich ein Beispiel, kommend aus der Motivation dieser Arbeit, zur Verdeutlichung.

2.1 Schritte des Infer Wildcards Algorithmus

Der Infer Wildcards Algorithmus lässt sich in die folgenden Prozessschritte aufgliedern. Veranschaulicht sind diese in Abbildung 2 in Form eines Flussdiagramms.

Schritt 1: Parsen des Quellcodes in den Java-TX AST (Abstract Syntax Tree). Darauf wird in dieser Arbeit nicht gesondert eingegangen. Die dafür nötigen Funktionen werden unverändert aus der bestehenden Java-TX Implementierung genutzt.

Schritt 2: Ersetzen der Java Generic Typparameter im AST mit TPH (Typplatzhalter).

Schritt 3: Generieren der Constraints zur Definition der möglichen Wildcardausdrücke für die generierten TPHs.

Schritt 4: Zusammenführen der neuen Infer Wildcard Constraints mit den bereits aus dem Quellcode extrahierten Constraints.

Schritt 5: Finden der korrekten und allgemeinsten Typen für die TPHs mit Hilfe des Java-TX Typunifikations-Algorithmus aus [4].

Schritt 6: Erzeugen des typkorrekten Quellcodes durch Einsetzen der zuvor gefundenen Typen anstelle der im AST definierten TPH.

```
List<? extends Object> method (List<String> input) {
  List<? extends Object> listOfObjects = input;
  Object test = listOfObjects.get(0);
  String string = "Test";
  input.add(string);
  return listOfObjects;
}
```

Listing 2: Typkorrekter Javacode

```
static <K, V> Map<K, V> ofEntries(Entry<? extends K, ? extends V>... entries) {
  if (entries.length == 0) { // implicit null check of entries
    return ImmutableCollections.Map0.instance();
  } else if (entries.length == 1) {
    return new ImmutableCollections.Map1<>(entries[0].getKey(),
    entries[0].getValue());
  } else {
    Object[] kva = new Object[entries.length << 1];
    int a = 0;
    for (Entry<? extends K, ? extends V> entry : entries) {
      kva[a++] = entry.getKey();
      kva[a++] = entry.getValue();
    }
    return new ImmutableCollections.MapN<>(kva);
  }
}
```

Listing 3: Beispiel für die Nutzung von Java `? extends` Wildcards aus `java.util.Map`

```
default void replaceAll(BiFunction<? super K, ? super V, \underline{? extends V}> function)
     {
  Objects.requireNonNull(function);
  for (Map.Entry<K, V> entry : entrySet()) {
    K k;
    V v;
    try {
      k = entry.getKey();
      v = entry.getValue();
    } catch (IllegalStateException ise) {
      // this usually means the entry is no longer in the map.
      throw new ConcurrentModificationException(ise);
    }

    // ise thrown from function is not a cme.
    v = function.apply(k, v);

    try {
      entry.setValue(v);
    } catch (IllegalStateException ise) {
      // this usually means the entry is no longer in the map.
      throw new ConcurrentModificationException(ise);
    }
  }
}
```

Listing 4: Beispiel für die Nutzung von Java `? super` Wildcards aus `java.util.Map`

```
static <K, V> Map<K, V> ofEntries(Entry<\underline{K}, \underline{V}>... entries) {
  if (entries.length == 0) { // implicit null check of entries
    return ImmutableCollections.Map0.instance();
  } else if (entries.length == 1) {
    return new ImmutableCollections.Map1<>(entries[0].getKey(),
    entries[0].getValue());
  } else {
    Object[] kva = new Object[entries.length << 1];
    int a = 0;
    for (Entry<K, V> entry : entries) {
      kva[a++] = entry.getKey();
      kva[a++] = entry.getValue();
    }
    return new ImmutableCollections.MapN<>(kva);
  }
}
```

Listing 5: Beispiel für die Nutzung von Java `? extends` Wildcards aus `java.util.Map` bei Nutzung von Java-TX

```
default void replaceAll(BiFunction<\underline{K}, \underline{V}, \underline{V}> function)
      {
  Objects.requireNonNull(function);
  for (Map.Entry<K, V> entry : entrySet()) {
    K k;
    V v;
    try {
      k = entry.getKey();
      v = entry.getValue();
    } catch (IllegalStateException ise) {
      // this usually means the entry is no longer in the map.
      throw new ConcurrentModificationException(ise);
    }

    // ise thrown from function is not a cme.
    v = function.apply(k, v);

    try {
      entry.setValue(v);
    } catch (IllegalStateException ise) {
      // this usually means the entry is no longer in the map.
      throw new ConcurrentModificationException(ise);
    }
  }
}
```

Listing 6: Beispiel für die Nutzung von Java `? super` Wildcards aus `java.util.Map` bei Nutzung von Java-TX

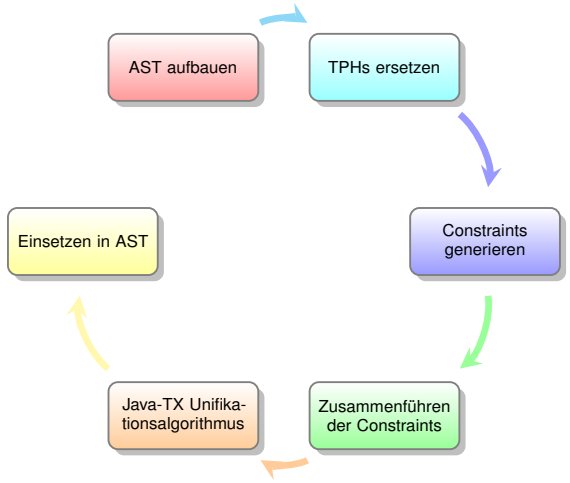


Abbildung 2: Ablauf Algorithmusschritte

2.1.1 Ersetzen der Java Generic Typparameter

Durch Iterieren über den zuvor generierten AST sollen die darin enthaltenen Java Generic Typparameter durch TPHs ersetzt werden. An jeder dieser Stellen können im Fortlauf des Algorithmus neue Typen durch Constraints bestimmt werden. Damit wird hier die Entscheidung und Stellen der Nachbesserung des Algorithmus in der Benutzereingabe getroffen. Insgesamt wird dieser Schritt durch die drei folgenden Unterschritte *Identifizieren von* `RefTypen`, *Behandlung von Java Generic Typen* und *Behandlung Java Generic Typparameter* untergliedert.

Identifizieren von `RefTypen`

Der 2.Schritt setzt den bereits erzeugten AST für den zu kompilierenden Quellcode voraus. Dieser wird durch den in Abbildung 3 beschriebenen Algorithmus auf Elemente, in denen Deklarationen von Methoden, lokalen Variablen oder Parametern vorkommen, untersucht. Diese Bereiche sind potentielle Kandidaten für zu analysierende Java Generic Typen.

Durch Iterieren über alle Elemente im AST werden jene vom Typ `Method` auf einen Rückgabetypen untersucht. Dabei gilt `void` als kein Rückgabetyp. Bei einem vorhandenen Rückgabetypen werden diese genauso wie gefundene Elemente vom Typ `FieldDecl`, `LocalVarDecl`, `ParamDecl` auf die Kindelemente im AST betrachtet. Gefundene `RefType` werden an den Algorithmus in Abbildung 4 zur weiteren Untersuchung übergeben. Anschließend wird mit dem nächsten Kindelement fortgefahren. Falls eine Bedingung nicht zutrifft, wird das nächste Element im AST bearbeitet. Die Schleife terminiert, wenn alle AST Elemente behandelt sind.

Behandlung von Java Generic Typen

Wird ein `RefType` im AST gefunden, erfolgt eine Bearbeitung nach dem in Abbildung 4 beschriebenen Algorithmus. Die weitere Verarbeitung des eingegebenen `RefType` ist an die Bedingung geknüpft, dass es ein Java Generic Ausdruck ist. Wenn die Überprüfung negativ verläuft, terminiert der Algorithmus für diesen Typen bereits. Bei bestätigten Java Generic Typen wird darauffolgende über deren Typparameter iteriert. Diese werden einzeln an die Funktion zur Behandlung von Java Generic Typparametern (siehe Abbildung 5) übergeben. Durch diese Iteration über die Typparameter erfolgt eine Behandlung mehrerer Typparameter eines Java Generic Ausdrucks. Wenn keine weiteren Typparameter zum Iterieren verbleiben, endet der Algorithmus für den `RefType`.

Behandlung Java Generic Typparameter

Zunächst muss der Algorithmus den Typ des Typparameters überprüfen. Im AST können an dieser Stelle auch Typen auftauchen, auf welche der Wildcard Inferierungsalgorithmus nicht sinnvoll anzuwenden wäre. Daher sind die Wildcards und TPHs von der weiteren Verarbeitung ausgeschlossen. Wildcards, da bereits die zu suchende Lösung durch den in dieser Arbeit beschriebenen Ansatz vorweggenommen wurde. TPHs, weil diese schon durch weitere Constraints aus den restlichen Quellcode bestimmt werden (generiert im folgenden vierten Schritt) und die Freiheitsgrade zur weiteren Typfindung schon bestehen.
Die akzeptierten Typen Generics und `RefType` werden auf verschachtelte Typen überprüft, indem diese an den in Abbildung 4 gezeigten Algorithmus übergeben werden. Es erfolgt innerhalb des selben ASTs die Manipulation aller verschachtelten Typen von innen nach außen. Folglich hat der jeweils äußere Typ am Ende nur `RefTypen` und/oder TPHs als Typparameter. Im AST tauchen die verschachtelten Typen nicht mehr auf. Nur der äußerste Typ bleibt bestehen und verweist auf einen TPH. Alle weiteren Beziehungen sind in der Menge `RepS` gespeichert. `RepS` ist eine Menge von n Tupeln. Der Algorithmus generiert je ein Tuple und speichert es in `RepS`, wenn ein TPH TPH im AST einen Typen ty ersetzt. Dabei kann ty sowohl ein TPH als auch ein `RefType` sein, was die Terminierung der Ersetzungsrekursion

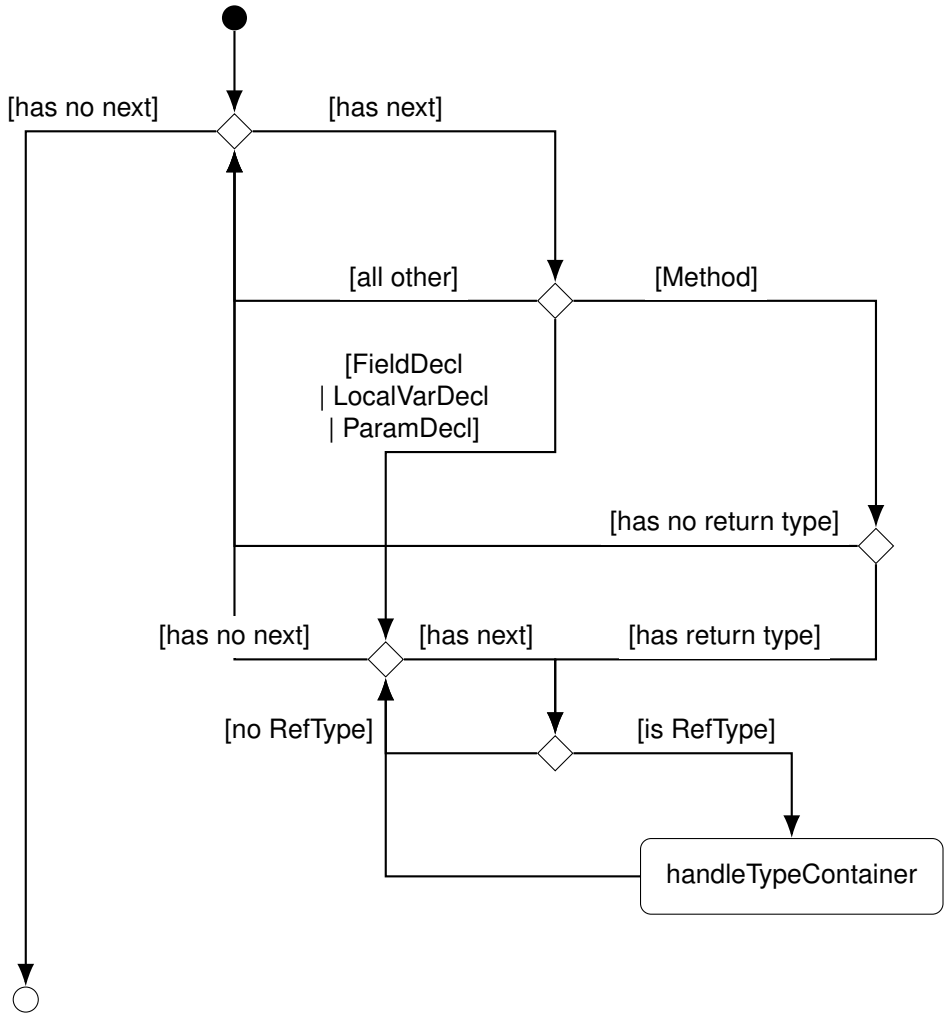


Abbildung 3: Algorithmus zum Finden von relevanten `RefType` im AST: `handleAst`

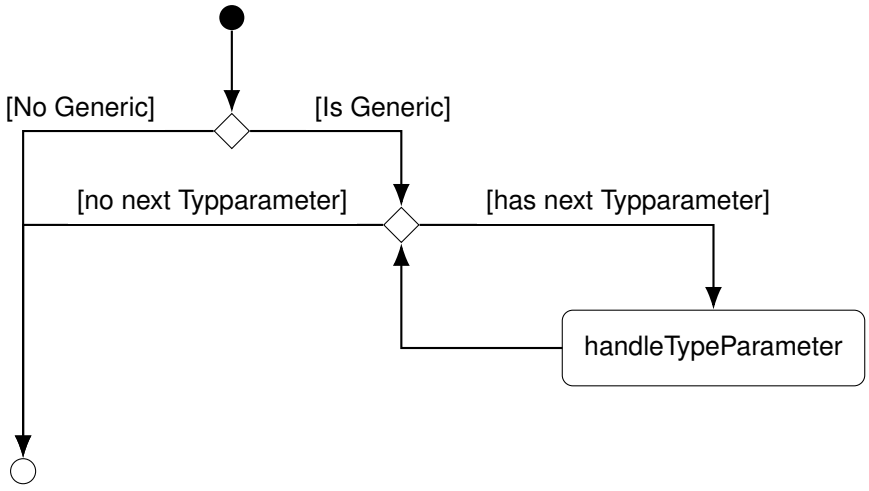


Abbildung 4: Algorithmus zum Ersetzen von Typparametern im AST, Umgang mit Java Generic Typen: `handleTypeParameterContainer`

zur Folge hat. An der Stelle im AST, an welcher der Typparameter gefunden wurde, wird TPH eingesetzt.

$$\texttt{RepS} = \{TPH_0 \doteq ty_0, ..., TPH_i \doteq ty_i | i < n\} \quad (1)$$

Diese Funktion wird beschrieben durch
replace: `Class` $\rightarrow$ `TClass` $\times$ `RepS`,
wobei `Class` eine für Java Wildcards nicht korrekt typisierte Klasse darstellt, jedoch keine TPHs enthält. `TClass` repräsentiert eine Klasse mit eingesetzten TPHs für die oben beschriebenen Bedingungen:

Der Algorithmus angewandt auf einen einfachen Java Generic Typen wie `List<String>`, wird das Ergebnis, gezeigt in Abbildung 6, generieren.

`List<String>` $\Rightarrow$ `List<A>` mit `A` $\mapsto$ (`A` $\doteq$ `String`)

Abbildung 6: Beispiel Typenersetzung für einfache Generische Typen

Verschachtelte Typen werden jeweils von innen nach außen ersetzt, so dass keine solche Typkonstrukte im AST mehr existieren. Damit sollen für jeden Typen, wie in Unterunterabschnitt 2.1.2 beschrieben, im Nachgang die Constraints unabhängig generiert werden können. Davon unberührt bleiben mehrfache Typparameter auf der gleichen Ebene. Ein Beispiel mit zwei ineinander verschachtelten Listen wird in Abbildung 7 dargestellt.

`List<List<String>>` $\Rightarrow$ `List<A>`
mit `A` $\mapsto$ (`A` $\doteq$ `List<B>`) und `B` $\mapsto$ (`B` $\doteq$ `String`)

Abbildung 7: Beispiel Typenersetzung für verschachtelte Generische Typen

2.1.2 Generierung der Constraints für die Typplatzhalter

Für jeden gefunden und durch den Algorithmus erzeugten TPH TPH aus Schritt 3 und gespeichert in der Menge `RepS`, werden die Constraints `TphConstraints` generiert. Ausschließliches Einsetzten von TPHs ermöglicht eine freie Typenfindung. Dieser Algorithmus soll sich ausschließlich auf die Findung besserer Wildcardtypen beschränken. Zusätzlich hat der eingegebene Code durch eine konkrete Typisierung auch Restriktionen, welche beachtet werden sollen. Hierfür erfolgt die Definition, dass der Typ TPH sowohl als `? extends` ty, `? super` ty oder ty evaluiert werden soll. Durch die Erzeugung solcher `TphConstraints` für jeden TPH in `RepS`, wird die in Schritt 3 aufgelöste Verschachtelung von Typen im AST durch Typbedingungen wieder hergestellt. Damit ist auch die Unifikation der geschachtelten Typparameter gewährleistet.

$$\texttt{TphConstraints} = \begin{pmatrix} TPH \doteq ty \\ \vee\, TPH \doteq \texttt{? extends}\ ty \\ \vee\, TPH \doteq \texttt{? super}\ ty \end{pmatrix}$$

Am Beispiel in Abbildung 8 ist zu sehen, wie der ersetzte Typparameter `A` durch die drei `OderConstraints`, bekannt aus [3], definiert wird. Selbiges ist auch mit dem Beispiel in Abbildung 9 für die verschachtelten Typen zu beobachten. Aufgrund der Verschachtelungstiefe x und der Generierung der drei `OderConstraints` für jede Ebene, steigt die Anzahl der möglichen Lösungen exponentiell mit 3^x. Hierbei werden Optimierungen am Unifikationsalgorithmus von Java-TX nicht mit berücksichtigt. In diesem Beispiel sind somit zur Lösung des Typplatzhalter `A` $3^2 = 9$ Möglichkeiten zu evaluieren. Die Generierung der `OderConstraints` ist ohne weitere Optimierung jedoch nötig, damit eine Subtypenrelation auf jeder Tiefe der Verschachtelung abgebildet und wenn nötig, als Lösung ausgegeben werden kann.

Die generierten n `TphConstraints` für jedes Tuple aus der Menge `RepS` werden zu den einen gemeinsamen Set `InfWildcardsConst` an Constraints für den Infer Wildcard Algorithmus zusammengefasst. Da für einen korrekten Code alle n `TphConstraints` gleichzeitig erfüllt sein müssen, werden diese mit der Und-Funktion $\wedge$ verknüpft.

$$\texttt{InfWildcardsConst} = \{\texttt{TphConstraints}_0 \wedge ... \wedge \texttt{TphConstraints}_i \mid i < n\}$$

Diese Funktion wird im Folgenden durch den Ausdruck referenziert:
genWildcards: `RepS` $\rightarrow$ `InfWildcardsConst`

2.1.3 Vereinigung der Constraints

Mit Hilfe des bereits in Java-TX implementierten Constraint Generierungsalgorithmus, werden die restlichen Constraints

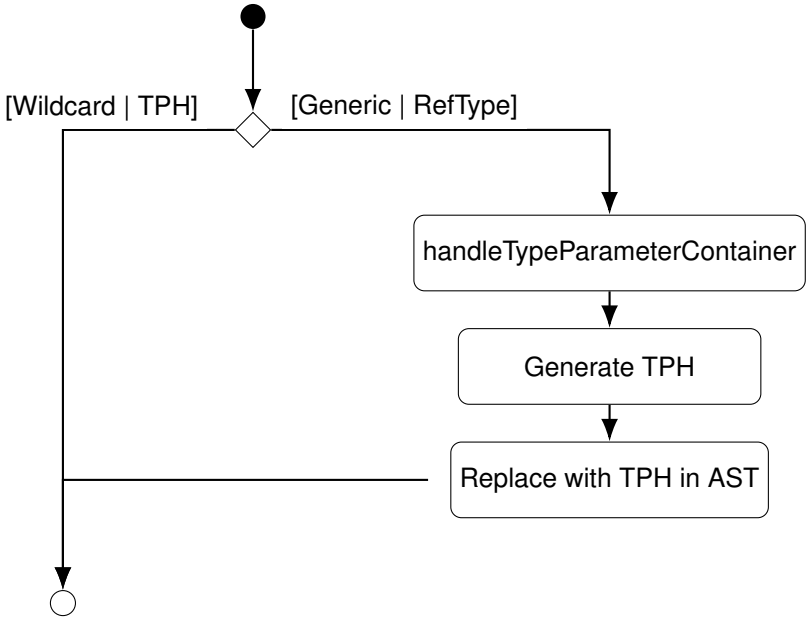


Abbildung 5: Algorithmus zum Ersetzen von Typparametern im AST, Umgang mit Typparametern: `handleTypeParameter`

`SourceCodeConstraints` für die Typunifikation generiert. `SourceCodeConstraints` sind Bedingungen von im AST befindlichen TPHs anhand des umgebenden Quellcodes, wie z.B. Variablentypen, Parametertypen von aufgerufenen Funktionen, usw. Diese Bedingungen sind nötig, da daraus erst die weiteren Typableitungen für die neu geschaffenen Typ Constraints für die Wildcard Typen getroffen werden können. Die generierten Bedingung `SourceCodeConstraints` und `InfWildcardsConst` werden nach folgender Gleichung zu `Constraints` zur weiteren Verwendung zusammengefasst.

$$\texttt{Constraints} = \texttt{InfWildcardsConst} \wedge \texttt{SourceCodeConstraints}$$

Diese Funktion wird im Folgenden durch den Ausdruck referenziert:
combine:
`InfWildcardsConst` $\times$ `AndConstraints` $\rightarrow$ `AndConstraints`

$$\texttt{List<String>} \Rightarrow \texttt{List<A>} \text{ mit } \texttt{A} \mapsto \begin{pmatrix} \texttt{A} \doteq \texttt{String} \\ \vee\ A \doteq \texttt{? extends String} \\ \vee\ A \doteq \texttt{? super String} \end{pmatrix}$$

Abbildung 8: Beispiel Constraint Generierung für einfache Generische Typen

$$\texttt{List<List<String>>} \Rightarrow \texttt{List<A>} \text{ mit } \texttt{A} \mapsto \begin{pmatrix} \texttt{A} \doteq \texttt{List<B>} \\ \vee\ \texttt{A} \doteq \texttt{? extends List<B>} \\ \vee\ \texttt{A} \doteq \texttt{? super List<B>} \end{pmatrix} \text{ und } \texttt{B} \mapsto \begin{pmatrix} \texttt{B} \doteq \texttt{String} \\ \vee\ \texttt{B} \doteq \texttt{? extends String} \\ \vee\ \texttt{B} \doteq \texttt{? super String} \end{pmatrix}$$

Abbildung 9: Beispiel Constraint Generierung für verschachtelte Generische Typen

2.1.4 Typunifikation

Die generierten Constraints `Constraints` werden in den aus vorgestellten Unifikationsalgorithmus eingesetzt. Dieser errechnet anhand der zusätzlich eingefügten Freiheitsgrade die für diese optimale Lösung $(cl'_1, \sigma'_1), ..., (cl'_n, \sigma'_n)$. Wenn nicht alle Randbedingungen mit den Freiheitsgraden erfüllt werden können, wird das Errechnen abgebrochen, da der Code nicht typkorrekt werden kann.

2.1.5 Einsetzen der Typergebnisse

Die definierten TPHs im AST durch die Funktion **replace** können durch die inferierten Typergebnisse ersetzt werden. Daraus lässt sich ein typkorrekter

AST aufstellen. Hieraus kann Quellcode, als auch Bytecode generiert werden.

2.2 Adaption von Java-TX TI

Zur Einbindung der in Unterabschnitt 2.1 aufgeführten Schritte, muss der aus [1] bekannte Java-TX TI Algorithmus angepasst werden. Durch Einfügen der zuvor definierten Funktionen **replace** **genWildcards** und **combine** an spezifischen Stellen der Funktion **TI** erhält man die Abwandlung **TI′**. Dadurch wird der zuvor erläuterte Ablauf der benötigten Schritte sichergestellt.

TI′:
$\texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \texttt{TClass} \cup \{\texttt{Error}\}$
TI′(Ass, cl) =
  let
    $(cl', RepS)$ = **replace**(cl)
    $(InfConS)$ = **genWildcards**$(RepS)$
    $(cl_t, ConS')$ = **TYPE**(Ass, cl')
    $(ConS)$ = **combine**$(InfConS, ConS')$
    $(cl'_1, \sigma'_1), ..., (cl'_n, \sigma'_n)$ = **TUnify**$(ConS)$
  in
    BuildClass$((mccs, s)_1, ..., (mccs, s)_n, cl_t)$

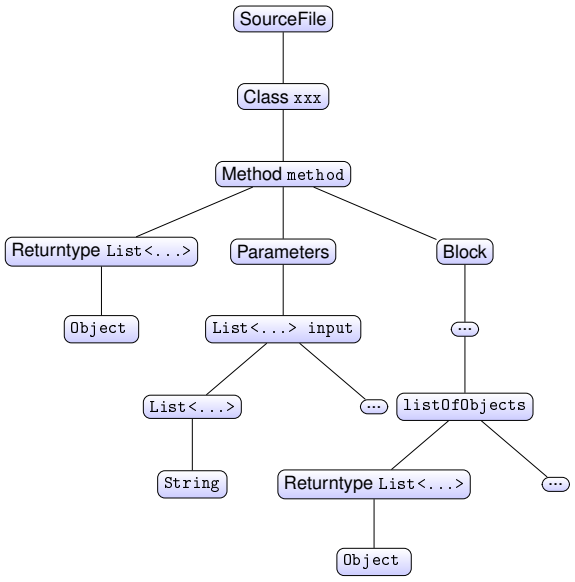


Abbildung 10: Beispiel AST geparst aus Listing 1

2.3 Algorithmus an einem Beispiel

Zum Veranschaulichen des Algorithmus soll das Listing 1 aus der Motivation aufgegriffen werden. Im Folgenden finden sich die vorher definierten Schritte mit dem Inhalt des Codebeispiels wieder:

Schritt 1: Durch das Parsen des Quellcodes von Listing 1 entsteht ein Java-TX AST wie in Abbildung 10 gezeigt

Schritt 2: Finden der Java Generic Typparameter und Ersetzen im AST mit neu generierten TPH, beschrieben durch Abbildung 11. Die geänderten Knoten sind darin farblich markiert.

$$\texttt{RepS} = \left\{ \begin{matrix} (\texttt{A} \doteq \texttt{java.lang.String}), \\ (\texttt{B} \doteq \texttt{java.lang.Object}), \\ (\texttt{C} \doteq \texttt{java.lang.Object}) \end{matrix} \right\}, n = 3$$

Schritt 3: Generieren der $3^3 = 9$ Constraints für die zuvor erzeugten $n = 3$ RepS Elemente.

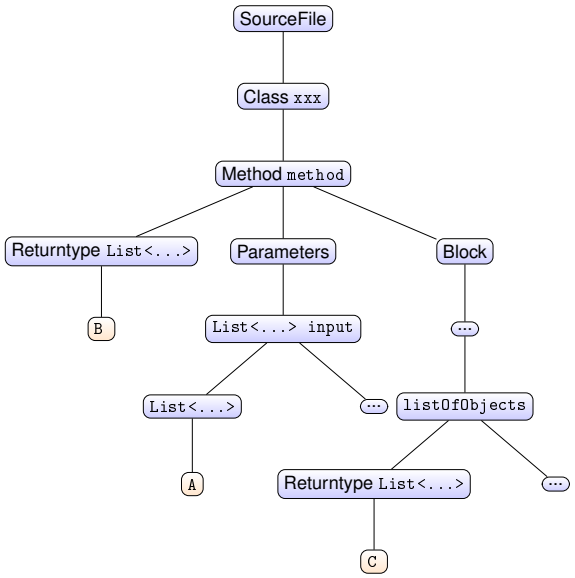


Abbildung 11: AST aus Listing 1 mit ersetzten Typen durch Typplatzhalter

$$
\begin{aligned}
&\texttt{InfWildcardsConst} = \\
&\begin{pmatrix} \texttt{A} \doteq \texttt{String} \\ \vee\ \texttt{A} \doteq \texttt{? extends String} \\ \vee\ \texttt{A} \doteq \texttt{? super String} \end{pmatrix} \\
\wedge &\begin{pmatrix} \texttt{B} \doteq \texttt{Object} \\ \vee\ \texttt{B} \doteq \texttt{? extends Object} \\ \vee\ \texttt{B} \doteq \texttt{? super Object} \end{pmatrix} \\
\wedge &\begin{pmatrix} \texttt{C} \doteq \texttt{Object} \\ \vee\ \texttt{C} \doteq \texttt{? extends Object} \\ \vee\ \texttt{C} \doteq \texttt{? super Object} \end{pmatrix}
\end{aligned}
$$

Schritt 4: Erzeugen der restlichen Quellcodeconstraints und Zusammenführen der zuvor generierten `InferWildcardsConst` zu `Constraints`.

Schritt 5: Typinferierung unter Betracht der erzeugten Constraints mit Hilfe des Java-TX Typunifikationsalgorithmuses. Daraus werden die korrekten Typen für die in Schritt 2 erzeugten TPH gewonnen, wodurch der Code typkorrekt wird.

$$
\begin{aligned}
\texttt{A} &\doteq \texttt{String} \\
\texttt{B} &\doteq \texttt{? extends Object} \\
\texttt{C} &\doteq \texttt{? extends Object}
\end{aligned}
$$

Schritt 6: Zum Überprüfen der Ergebnisse und zum Erzeugen des typkorrekten Quellcodes erfolgt ein beispielhaftes Einsetzen der Ergebnisse im AST. Abbildung 12 verdeutlicht farblich, welche Typen im AST durch das Einsetzen der Typergebnisse geändert wurden.

$$
\begin{aligned}
\texttt{Object} \rightarrow B &\rightarrow \texttt{? extends Object} \\
\texttt{String} \rightarrow A &\rightarrow \texttt{String} \\
\texttt{Object} \rightarrow C &\rightarrow \texttt{? extends Object}
\end{aligned}
$$

Die Typkorrektheit wurde bereits durch den Compiler überprüft. Dadurch kann hieraus direkt Bytecode oder auch lesbarer Quellcode erzeugt werden. Letzteres entspricht dem in der Motivation bereits aufgeführtem gewünschten Ergebnis Listing 2.

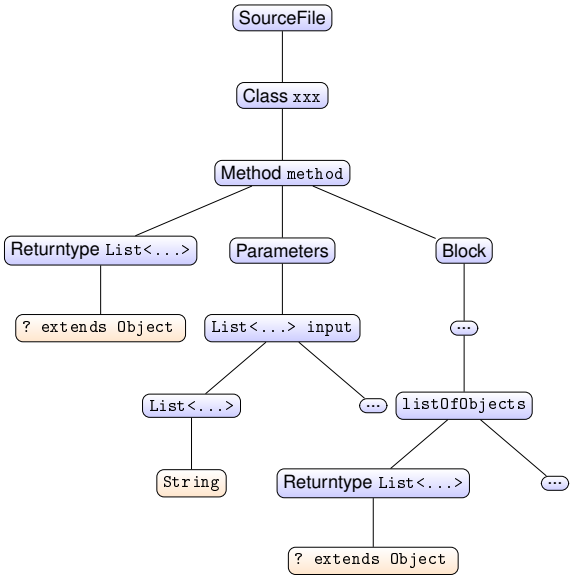


Abbildung 12: AST für Listing 1 mit eingesetzten inferierten Typen

Literatur

[1] Martin Plümicke. „More Type Inference in Java 8“. In: *Perspectives of System Informatics: 9th International Ershov Informatics Conference, PSI 2014*. Hrsg. von Andrei Voronkov und Irina Virbitskaite. St. Petersburg, Russia: Springer, 2015, S. 248–256. ISBN: 978-3-662-46823-4.

[2] Martin Plümicke. „Typeless programming in Java 5.0 with wildcards“. In: *Proceedings of the 5th international symposium on Principles and practice of programming in Java - PPPJ '07*. Hrsg. von Vasco Amaral. New York, New York, USA: ACM Press, 2007, S. 73. ISBN: 9781595936721.

[3] Andreas Stadelmeier und Martin Plümicke. „Adding overloading to Java type inference“. In: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015*. Hrsg. von Wolf Zimmermann u. a. Bonn, 2015, S. 127–132.

[4] Florian Steurer und Martin Plümicke. „Erweiterung und Neuimplementierung der Java Typunifikation“. In: *Tagungsband des 35ten Jahrestreffens der GI-Fachgruppe “Programmiersprachen und Rechenkonzepte”*. Research report. Oslo, 2018, S. 134–149. ISBN: 978-82-7368-447-9.

Bytecode-Generierer für ein typloses Java

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

Zusammenfassung

JavaTX ist eine Programmiersprache basierend auf Java 8, welche die bekannte Syntax durch eine globale Typinferenz erweitert. Damit können Typen im gesamten Programm durch den Compiler inferiert werden. Im Rahmen dieser Arbeit wurde der Codegenerierer überarbeitet. Um die Implementierung zu erleichtern wurde ein neuer AST angelegt welcher die Distanz zum Bytecode überbrücken sollte. Ziel ist es, durch neue Java Features wie Records und Sealed Interfaces eine einfachere und saubere Implementierung zu erreichen.

1 Einleitung

1.1 Java und JavaTX

Die Programmiersprache Java hat sich seit ihrer Entstehung stetig weiterentwickelt. Mit Java 5 wurden Generics hinzugefügt, ein Konzept, das in funktionalen Programmiersprachen als parametrische Polymorphie bekannt ist. Damit ist es möglich, Klassen zu erstellen, die über die verwendeten Typen abstrahieren. Seit Java 8 gibt es Lambda Ausdrücke, die auf Functional Interfaces basieren. Ein solches Interface besteht im Wesentlichen aus einer abstrakten Funktion, die dann vom Lambda Ausdruck implementiert wird. Die Vor- und Nachteile dieses Ansatzes werden in [4] diskutiert. Seit Java 9 gibt es eine limitierte Form der Typinferenz die es erlaubt, den Typ von lokalen Variablen wegzulassen. Dieser wird dann vom Compiler inferiert.

JavaTX (steht für Java Type eXtended) ist eine auf Java 8 basierende Programmiersprache, die Java um globale Typinferenz, wie sie aus funktionalen Sprachen wie Haskell bekannt ist, erweitert. Damit ist es möglich, Typen auch in anderem Kontext inferieren zu lassen, wie bei Feldern und Funktionen.

Weiterhin wird Java durch echte Funktionstypen erweitert, wie sie beispielsweise in Scala zu finden sind. Der Vorteil dabei besteht darin, dass der Typ von Lambda Ausdrücken inferiert werden kann und nicht wie bei Java aus dem Kontext oder explizit definiert wird. Als Drittes werden Generics für Klassen und Funktionen generiert, mit denen freie Typvariablen generalisiert werden [6].

1.2 Motivation

Der Codegenerierer von JavaTX in der jetzigen Form ist zwar funktional, aber die Codequalität lässt zu wünschen übrig. Es gibt viele Stellen, an denen globaler Zustand verwendet wird, was aus Gründen der geringen Lesbarkeit des Codes und die schlechte Wartbarkeit zu vermeiden ist. Auch wird gerade durch das Visitorpattern sehr viel Code benötigt, der durch die neuen Java-Features rund um Switch-Statements stark vereinfacht werden kann. Andere Features sorgen ebenfalls dafür, dass entweder weniger Code benötigt wird oder aber eine bessere Implementierung benutzt werden kann.

Des weiteren wird durch die direkte Verwendung des ASTs des Parsers eine starke Kupplung eingeführt, die durch die Verwendung eines separa-

ten ASTs verringert werden kann. Der Vorteil hierbei ist, dass Änderungen am Parser und der dazugehörigen Infrastruktur nur eine Anpassung der Übersetzungsebene nötig machen, der Codegenerierer aber bestenfalls nicht verändert werden muss.

Zuletzt soll auch eine Überladung von Funktionstypen erreicht werden. Dazu wird für sämtliche Typisierungen einer Funktion eine separate Klasse generiert, damit diese auch nach Java ein separater Typ sind.

2 Grundlagen

2.1 Typinferenz

2.1.1 Was ist Typinferenz?

In Java hat jeder Ausdruck, wie bei anderen statischen Programmiersprachen, einen konkreten Typ. Der Unterschied zu einer Sprache mit Typinferenz besteht darin, dass diese Typen aus dem Kontext ermittelt werden. Hierbei ist allerdings festzustellen, dass anders als bei dynamischen Programmiersprachen wie Python, weiterhin jeder Ausdruck einen statischen Typ hat. Es ist also nicht möglich, einer Variablen Ausdrücke mit verschiedenen Typen zuzuweisen.

In Java gibt es bereits seit Version 7 eine einfache Form der Typinferenz, den sogenannten Diamond-Operator:

```
1 // Java 6
2 ArrayList<String> myList = new
      ArrayList<String>();
3 // Java 7
4 ArrayList<String> myList = new
      ArrayList<>();
```

Bei dieser einfachen Form der Typinferenz kann man bereits die Vorteile der Typinferenz gut abschätzen. Es ist nicht mehr nötig, den Typ `String` mehrfach anzugeben. Das spart Tipparbeit und damit ist der Code auch weniger anfällig für Fehler. Allerdings muss man bei diesem Beispiel immer noch den Typ `ArrayList` mehrfach angeben.

Seit Java 9 gibt es hierfür eine Form der Typinferenz für lokale Variablen, mit dem neuen Schlüsselwort `var`:

```
1 // Java 9
2 var myList =
3     new ArrayList<String>();
```

Diese Form der Typinferenz ist trivial, da der Compiler bereits für das Typchecking den Typ des Ausdrucks auf der rechten Seite der Zuweisung berechnen muss. Für die Typinferenz wird also lediglich dieser Typ für die Variable inferiert. Ohnehin sind Typchecking und Typinferenz sehr stark miteinander verwandt. Zuerst wird die Typinferenz angewendet und dann mit dem Typchecker überprüft, ob die Typen konsistent sind.

Die Typinferenz in Java ist allerdings nur sehr eingeschränkt möglich. So muss beispielsweise für die Typinferenz einer lokalen Veriablen direkt eine Zuweisung erfolgen:

```
1 // Invalid
2 var myList;
3 myList = new ArrayList<String>();
```

Obwohl hier der Typ von `myList` eindeutig aus dem Kontext klar wird, kann Java diesen Typ nicht inferieren.

Weiterhin ist es nicht möglich, Typen in anderem Kontext zu inferieren. Beispielsweise müssen für Funktionen immer alle Parameter typisiert sein sowie der Rückgabetyp. Auch ist es nicht möglich, den Typ von Instanzvariablen zu berechnen.

In JavaTX werden diese Restriktionen aufgehoben und es ist möglich, Typen in sämtlichen Kontexten zu inferieren. Der Algorithmus hierfür wird in [5] vorgestellt. Er basiert auf dem Algorithmus W von Milner und Damas [2], der durch Subtypisierung von Java Klassen ergänzt wurde. Zusammengefasst wird ein Set von Constraints erstellt, welches durch die eingesetzten Typen erfüllt wird. Hierbei kann es mehrere Lösungen geben, wofür Java passenderweise das Overloading von Methoden erlaubt. Das heißt, eine Methode kann mehrmals angegeben werden, so lange die Parameter verschiedene Typen haben. JavaTX generiert also teils mehrere Methoden für eine Definition, was einiges an Arbeit erspart.

Gegeben sei folgendes JavaTX Programm:

```
1  import java.lang.Integer;
2  class Main {
3      x = 42;
4      f (y, z) {
5          if (y) {
6              return z + x;
7          } else {
8              return 0;
9          }
10     }
11 }
```

Das Resultat ist die folgende dekompilierte Klasse:

```
1  public class Main {
2      public Integer x = 42;
3
4      public Main() {
5      }
6
7      public Integer f(Boolean y,
           Integer z) {
8          return y ? z + this.x : 0;
9      }
10 }
```

Man sieht, dass für `y` der Typ Boolean ausgewählt wurde, da der Typ in einem If-Statement verwendet wird, welches einen Boolean akzeptiert. Für `x` wurde einfach der Typ des Integer Literals verwendet, ähnlich wie zuvor bei der Zuweisung von einer lokalen Variable. Für `z` wurde der Typ Integer ausgewählt, da bei der Addition zweier Integer ein weiterer Integer heraus kommt. Hier hätten theoretisch auch andere Typen eingesetzt werden können, aber JavaTX betrachtet aus Performancegründen nur die importierten Datentypen. Wenn nun statt `java.lang.Integer` `java.lang.Double` importiert wird, sieht die Klasse folgendermaßen aus:

```
1  public class Main {
2      public Double x = (double)42;
3
4      public Main() {
5      }
6
7      public Double f(Boolean y,
           Double z) {
8          return y ? z + this.x : (
               double)0;
9      }
10 }
```

Man sieht also, dass für x und z jeweils verschiedene Typen eingesetzt werden können.

Bei diesen einfachen Beispielen wird noch nicht ganz klar, wie mächtig Typinferenz sein kann. Gerade bei komplizierten generischen Typen mit mehreren Typparametern und Wildcard-Typen wird viel Code eingespart. Das liegt teilweise auch daran, dass Java keine Möglichkeit bietet, einen Typalias zu erstellen.

Im Stream Interface der Java Standardbibliothek werden mehrere Methoden mit komplizierten Signaturen definiert, wie zum Beispiel `reduce`:

```
1  public interface Stream<T> extends
       BaseStream<T, Stream<T>> {
2      <U> U reduce (
3          U identity,
4          BiFunction<U, ? super T, U
               > accumulator,
5          BinaryOperator<U> combiner
6      );
7      ...
8  }
```

2.1.2 Prinzipaltyp

In [7] wird die Prinzipaltyp-Eigenschaft vorgestellt:

Gegeben: Ein typisierbarer Term M
Es existiert: Eine Typisierung $A \vdash M : \sigma$
repräsentiert alle möglichen Typisierungen von M

Es ist sinnvoll, für ein Typssystem diese Eigenschaft zu besitzen, da es die Möglichkeit eröffnet für einen beliebigen Ausdruck einen Typ herzuleiten, welcher alle möglichen Typisierungen einschließt. Ein Typinferenzalgorithmus berechnet genau diesen Prinzipaltyp.

Für Java wird diese Definition nach [6] angepasst, damit die Prinzipaltypen dem maximalen Element der Subtyp-Relation entsprechen:

> „Ein Schnitttyp mit der minimalen Anzahl an Elementen für eine Deklaration entspricht dem Prinzipaltyp, wenn jedes andere Typschema für die Deklaration ein generischer Subtyp eines Elements des Schnitttyps ist.“

2.2 Sealed Classes

In Java werden typischerweise Vererbungshierarchien verwendet, um verschiedene Aktionen, basierend auf unterschiedlichen Datentypen, auszuführen:

```
1  public interface Animal {
2      public void talk();
3  }
4  public class Dog implements Animal
       {
5      @Override
6      public void talk() {
7        System.out.println("wuff");
8      }
9  }
10 public class Cat implements Animal
       {
11     @Override
12     public void talk() {
13       System.out.println("meow");
14     }
15 }
```

Hier wird beispielsweise die Methode `talk` überschrieben, was dazu führt, dass bei dem Aufruf von `talk` auf dem Interface `Animal` jeweils die korrekte Methode ausgewählt wird, je nachdem um welchen Typen es sich handelt. Ein großer Vorteil dieser Herangehensweise ist, dass `Animal` beispielsweise außerhalb einer Bibliothek vom Benutzer der Bibliothek erweitert werden kann.

Manchmal ist es jedoch nicht erwünscht, dass der Nutzer der Bibliothek die Klassen beliebig erweitern kann. Das kann etwa dann der Fall sein, wenn die Klasse nicht für Vererbung vorgesehen wurde und es Code gibt, welcher nur für bestimmte Subklassen funktioniert. In früheren Versionen von Java konnte dies nur beispielsweise durch das Schlüsselwort `final` erreicht werden, welches allerdings die Klasse komplett für Vererbung ausschließt.

Je nach Modellierung des Problems kann es sinnvoll sein, die Logik für verschiedene Fälle je nach Typ einer Instanz in einer Funktion zu implementieren, anstatt auf Vererbung zu setzen.

Ein einfaches Beispiel einer solchen Implementierung kann auf Enums basieren:

```
1  enum Animal { CAT, DOG }
2
3  static void talk(Animal animal) {
4      switch (animal) {
5      case CAT:
6        System.out.println("meow");
7        break;
8      case DOG:
9        System.out.println("wuff");
10       break;
11     }
12 }
```

Das Problem bei solch einer Implementierung ist, dass `CAT` und `DOG` jeweils ein Singleton sind und deshalb keine Instanzvariablen besitzen können. Es wäre also sinnvoll, einen Switch über den Typ einer Objektinstanz zu benutzen. In älternen Versionen von Java sähe das in etwas so aus:

```
1  public interface Animal {}
2
3  static void talk(Animal
4      if (animal instanceof Cat) {
5          Cat cat = (Cat) animal;
6          ...
7      } else if (animal instanceof
           Dog) {
8          ...
9      }
10 }
```

Das Problem hierbei ist, dass Animal von außerhalb erweitert werden kann und `talk` dann nicht das korrekte Ergebnis liefern kann. Ebenso wird nicht zur Kompilierzeit überprüft, ob alle Fälle abgedeckt werden.

In [1] wird, um dieses Problem zu beheben, ein neuer Typ von Klasse eingeführt, nämlich die sogenannten Sealed-Klassen. Mit dem modifier `sealed` können nur bestimmte Subklassen die Klasse erweitern. Die Syntax dafür sieht so aus:

```
1  public sealed interface Animal
       permits Cat, Dog  {}
2
3  public final class Cat implements
       Animal {}
4  public final class Dog implements
       Animal {}
5
6  static void talk(Animal
7      switch (animal) {
8          case Dog dog:
9              System.out.println("
                   wuff");
10             break;
11         case Cat cat:
12             System.out.println("
                   meow");
13             break;
```

```
14      }
15  }
```

Man sieht also, dass das Interface `Animal` nur die Subtypen `Cat` und `Dog` zulässt. Diese beiden Klassen müssen mit `final` markiert werden, damit sie an anderer Stelle nicht erweitert werden können und damit die Einschränkung des Interfaces `Animal` übergehen. Es ist allerdings möglich, durch ein weiteres Schlüsselwort `non-sealed` diese Einschränkung zu übergehen. Da dies allerdings nur durch Klassen möglich ist, die ohnehin in der Permits-Klausel erwähnt werden müssen, bleibt die Kontrolle darüber bei der Implementierung der Bibliothek, in der das Sealed Interface verwendet wird. Weiterhin ist es möglich, die Klassen ebenfalls mit `sealed` zu markieren und eine eigene Permits-Klausel zu erstellen.

Ebenfalls zu erwähnen, ist diese neue Form des Switch-Statements. Es wird, wie bei dem Beispiel davor, über den Typ des Ausdrucks `animal` geswitched. Diese Form des Switch-Statements wird in der Implementierung des Codegenerierers noch öfters verwendet.

2.3 Recordtypen

In Java 15 und [3] wird ein neues Schlüsselwort für die Erstellung von Klassen hinzugefügt: `record`. Java kann an manchen Stellen sehr wortreich sein, gerade wenn es um die Erstellung von POD-Typen, das heißt Klassen, die als unveränderbare Daten-Container gedacht sind, geht. Es müssen hier mehrere Methoden implementiert werden, die alle nach Schema-F aufgebaut sind. Hierzu gehören die Getter-Methoden für die unveränderbaren Datenfelder, die alle `final` sind und im Konstruktor übergeben werden. Ebenfalls implementiert werden müssen die Methoden `hashCode` und `equals`, die jeweils nötig sind, um Objekte der Klasse in einer `HashMap` zu verwenden. Des Weiteren kann auch noch die Methode `toString` implementiert werden, die eine textuelle Ausgabe des Objekts ermöglicht.

Andere Sprachen, die auf der JVM basieren, wie beispielsweise Kotlin, haben bereits ein Konzept namens Daten-Klassen, die all diese Methoden automatisch generieren.

In Java 15 werden eben diese Record-Klassen vorgestellt:

```
1  public record Employee(String
       firstName, String lastName) {}
```

Generiert werden hier die Methoden `equals`, `hasHCode`, `toString` und Accessor-Methoden für die jeweiligen Felder des Records mit dem selben Namen.

Dabei ist zu beachten, dass eine Klasse generiert wird, die von `java.lang.Record` erbt, das heißt, anders als beispielsweise in Kotlin, kann ein Record von keiner anderen Klasse erben.

Eine weitere Einschränkung besteht darin, dass die Klasse selbst implizipt `final` ist, also ebenso keine anderen Klassen von ihr erben können. Allerdings kann ein Record Interfaces implementieren.

Durch die Verbindung von Record-Klassen mit Sealed Interfaces können mit Java algebraische Datentypen implementiert werden. Die Sealed Interfaces beschreiben demnach Summentypen und die Record-Klassen sind Produkttypen.

Gerade für die Implementierung des neuen Target-ASTs sind Record-Klassen sehr sinnvoll. Das liegt daran, dass jeweils die ganze Logik zur Traversierung außerhalb des ASTs definiert ist, und die einzelnen Nodes demnach nur als Container für ihren Zustand definiert sind. Allerdings gibt es auch Nachteile. So kann beispielsweise `TargetConstructor` nicht von `TargetMethod` erben, was dazu führt, dass in diesem Fall mehrere Record-Typen implementiert werden, welche die selben Felder besitzen.

3 Implementierung

3.1 Konzeption

Um die Ziele zu erreichen gibt es mehrere Alternativen. Die sicherlich naheliegenste Möglichkeit wäre die Anpassung des vorhandenen Codegenerierers. Da es allerdings viel Zeit benötigen würde, diesen zu verstehen und die Codequalität nicht den neusten Standards entspricht, scheidet diese Möglichkeit aus. Es wird also eine komplett neue Implementierung des Codegenerierers benötigt.

Als nächste Alternative wäre zu betrachten, ob der schon vorhandene AST im neuen Codegenerierer verwendet werden kann. Ein großer Vorteil hierbei ist, dass es nicht zur Duplizierung von verschiedenen Kontrollstrukturen kommt. Ein If-Statement beispielsweise sieht im AST des Parsers und im AST des Codegenerierers nahezu gleich aus. Die meisten Kontrollstrukturen werden

eins zu eins übersetzt. In Zukunft müssen mehrere Stellen angepasst werden, wenn ein neues Konstrukt geparst werden soll.

Die Vorteile bei einer Neuschreibung des Target-ASTs sind allerdings, dass es so möglich ist, eine geringere Kupplung des Parsers und des Codegenerierers über die Schnittstelle des ASTs zu erreichen. Wenn nun also der Parser geändert wird, muss lediglich die Übersetzung des ASTs angepasst werden. Auch kann es sein, dass später der Codegenerierer Optmierungen durchführt, für die ein Zustand im AST hinterlegt werden muss. In Zukunft divergieren also AST des Parsers und des Codegenerierers.

Ein großer Vorteil der Konzeption als komplett neues Modul des Compilers ist, dass der vorhandene Code nicht geändert werden muss. Als Teil eines größeren Projektes macht es Sinn, Module möglichst unabhängig voneinander zu machen, damit bei Änderungen nur das Modul verstanden werden muss, nicht aber das ganze Projekt.

Für die Implementierung des Codegenerierers muss eine Traversierung des ASTs stattfinden. Klassisch wird so etwas mit dem Visistor-Pattern implementiert. Da es nun aber seit Java 18 eine neue Möglichkeit gibt, diese Traversierung mit Hilfe eines Typeswitches zu implementieren, gibt es eine Alternative zum Visistor-Pattern.

3.2 Target-AST

Als Eingabe für den Code-Generierer wurde ein neuer abstrakter Syntaxbaum implementiert. Das Ziel hierbei war es, einen Schritt näher in Richtung Bytecode zu gehen, um eine möglichst direkte Transformation zu erlauben. Dazu wurden die Typplatzhalter, die von der Typunifikation in den Syntaxbaum geschrieben worden sind, durch konkrete Typen ersetzt. Neu eingeführt wurden hierbei die primitiven Datentypen, die nötig sind, um Funktionen aus der Java Standardbibliothek aufrufen zu können.

Die Implementierung benutzt zum großen Teil die in Java 15 eingeführten Record-Typen [3]. Dadurch wird die Vererbung innerhalb des Target-ASTs vereinfacht, da Record-Typen nur Interfaces implementieren, aber nicht von anderen Klassen erben können. Stattdessen wurden für die Implementierung Sealed Interfaces verwendet [1]. Dies macht es möglich, bei der Bytecode-Generierung einen Typswitch anstelle des Visitor-Patterns zu verwenden.

Ein weiterer Vorteil der Neuimplementierung des Target-ASTs ist, dass eine bessere Trennung des Codegenerierers und des Parsers ermöglicht wird. Wenn zuvor eine Änderung am AST gemacht wurde, musste sowohl der Parser als auch der Codegenerierer angepasst werden, um auf die neue Änderung zu reagieren. Die Trennung des Parsers vom Codegenerierer durch die Übersetzung des ASTs führt dazu, dass lediglich diese Übersetzung geändert werden muss, anstatt den Codegenerierer zu verändern.

3.3 Einsetzen des Unify-Ergebnisses

Gegeben sei ein einfaches Beispiel aus der Testsuite des Compilers, welches iterativ die Fakultät berechnet:

```
1  import java.lang.Integer;
2
3  public class Fac {
4      getFac(n) {
5          var res = 1;
6          var i = 1;
7          while (i <= n) {
8              res = res * i;
9              i++;
10         }
11         return res;
12     }
13 }
```

Listing 1: Fac.jav

Hier müssen mehrere Typen berechnet werden. Die Typinferenz arbeitet immer von Innen nach Außen, also wird zunächst der Rumpf der Methode betrachtet. Die Typen von res und i sind einfach zu finden, da sie gleich zugewiesen werden. Da `java.lang.Integer` importiert wird, bekommen beide den Typ `Integer`. Um den Typ von Parameter `n` zu berechnen, wird der Ausdruck `i <= n` betrachtet. Der Typ des Ausdrucks ist `Boolean`, da der `<=` Operator einen Boolschen Wert zurück gibt. Da `Integer` nur mit anderen Zahlen verglichen werden kann und es durch das Import-Statement eine weitere Einschränkung gibt, wertet der Typ ebenfalls zu `Integer` aus.

Das Ergebnis der Typinferenz ist ein AST mit Typvariablen und ein Set von Constraints.

```
1  class Fac {
2      Fac(){
3          super(());
4      }::TPH AD
5
```

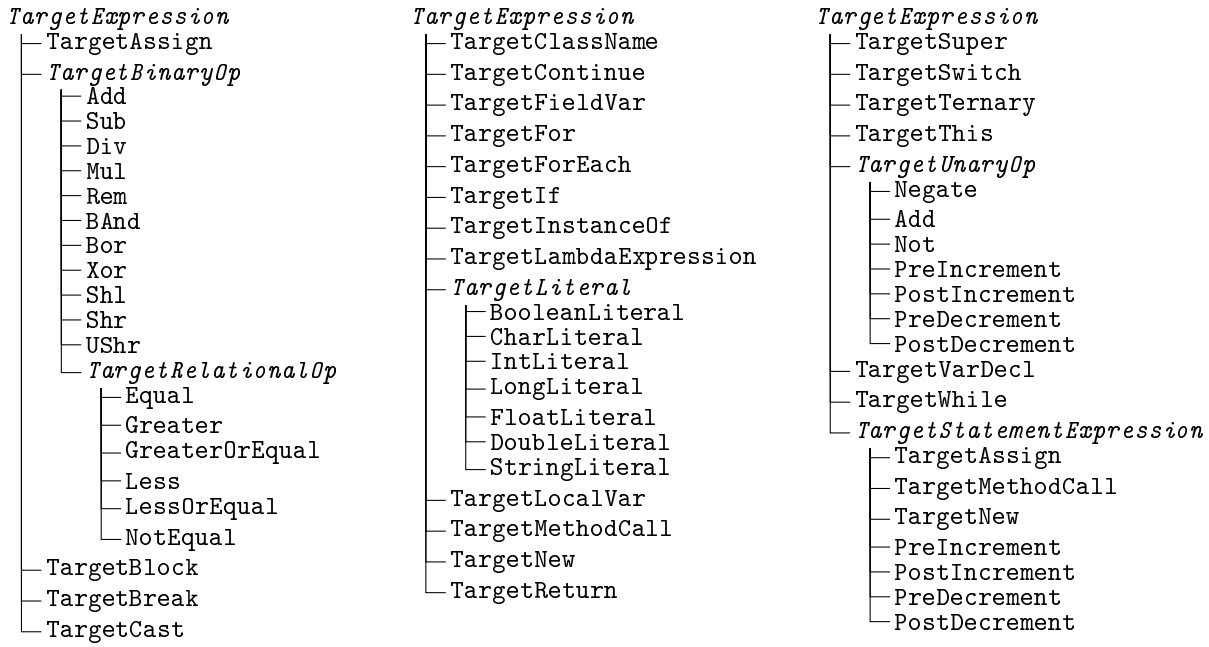


Abbildung 1: Hierarchie Target-AST (Kursiv gedruckte Elemente sind Interfaces)

```
6       TPH M getFac(TPH N n)({
7           TPH O res;
8           (res)::TPH O = 1;
9           TPH Q i;
10          (i)::TPH Q = 1;
11          while((i)::TPH Q | (n)::
                TPH N)({
12              (res)::TPH O = (res)::
                    TPH O | (i)::TPH Q;
13              (i)::TPH Q++;
14          })::TPH V;
15
16          return (res)::TPH O;
17      })::TPH X
18
19      Fac()({
20          super(());
21      })::TPH AA
22  }
```

Listing 2: AST von Fac.jav mit Typplatzhaltern

```
1  [[  (TPH T = java.lang.Integer),
2      (TPH S = java.lang.Boolean),
3      (TPH N = java.lang.Integer),
4      (TPH M = java.lang.Integer),
5      (TPH P = java.lang.Integer),
6      (TPH R = java.lang.Integer),
7      (TPH Q = java.lang.Integer),
8      (TPH O = java.lang.Integer),
9      (TPH U = java.lang.Integer)]]
```

Listing 3: Resultat der Typinferenz

In diesem Fall besteht das Resultat der Typinferenz aus einem Set. Es können aber auch mehrere Möglichkeiten existieren (Siehe 2.1.2).

Im ersten Schritt werden Constraints der Form $(T \doteq \text{RefType})$ ausgewertet. Das bedeutet, die Typvariable T entspricht dem konkreten Typ RefType. Dafür wird eine `HashMap concreteTypes` von Typplatzhalter auf Referenztyp angelegt. Ebenfalls ausgewertet werden Constraints der Form $(T \doteq \text{TypePlaceholder})$. Diese werden in eine separate `HashMap equals` eingefügt.

Wenn nun ein Typplatzhalter durch einen konkreten Typ ersetzt werden soll, wird zunächst in `equals` nach dem Typplatzhalter gesucht. Wenn dieser vorhanden sein sollte, wird er durch den Typplatzhalter aus der `HashMap` ersetzt. Im zweiten Schritt wird dann in `concreteTypes` nach dem Referenztyp gesucht. Falls dieser vorhanden sein sollte, wird der konkrete Typ in einen `TargetType` umgewandelt. Falls kein konkreter Typ existiert, handelt es sich um eine freie Typvariable und sie wird durch einen generischen Typ ersetzt, der den selben Namen hat wie der Typplatzhalter.

3.4 Zusammenfassung und Ausblick

Im Laufe dieser Arbeit wurde der Codegenerierer für JavaTX neu geschrieben. Dabei wurde ein neuer AST implementiert. Durch einen Verarbeitungsschritt worde der AST des Parsers in diesen übersetzt. Zu Hilfe genommen wurden Rekordtypen, sealed Interfaces und der in Java 18 neu eingeführte Typswitch. Dadurch wurde die Codequalität im Vergleich zu dem bereits vorhandenen Bytecodegenerierer verbessert. Die Implementierung ist modular aufgebaut und erlaubt in Zukunft ein einfaches Anpassen.

Nicht implementiert wurden Optimierungen, wie das direkte Verwenden von `if_icmp` in If-Statements oder das Löschen von nicht benutztem Code. Der AST wird sehr wörtlich übernommen. Hier könnte in Zukunft noch eine bessere Implementierung verwendet werden.

Als ein Zusatzschritt vor dem Codegenerierer wurden die Typinformationen des Typunifiers in den neuen AST eingesetzt. Wildcardtypen funktionieren momentan noch nicht, hier ist also noch eine Anpassung nötig.

Literatur

[1] Gavin Bierman. *JEP 397: Sealed Classes (Second Preview)*. 11. März 2022. URL: https://openjdk.org/jeps/397.

[2] Luis Damas und Robin Milner. „Principal Type-Schemes for Functional Programs“. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, S. 207–212. ISBN: 0897910656.

[3] Brian Goetz. *JEP 384: Records (Second Preview)*. 11. März 2022. URL: https://openjdk.org/jeps/384.

[4] Martin Plümicke und Andreas Stadelmeier. „Introducing Scala-like Function Types into Java-TX“. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ACM, 2017, S. 248–256. ISBN: 978-1-4503-5340-3.

[5] Martin Plümicke und Florian Steurer. „Erweiterung und Neuimplementierung der Java Typunifikation“. In: *Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts*. Faculty of Mathematics und Natural Sciences, University of Oslo, 2018, S. 134–149. ISBN: 978-82-7368-447-9.

[6] 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.

[7] Jim Trevor. „What are principal typings and what are they good for?“ In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*. 1996, S. 42–53.

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.: +49 711 1849 605

www.dhbw-stuttgart.de/technik/insights

Umschlaggestaltung: Kerstin Faißt
Bildnachweis: Gerd Altmann auf Pixabay
ISSN 2193-9098

ISSN 2193-9098

www.dhbw-stuttgart.de/technik/insights

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.: +49 711 1849 605

www.dhbw-stuttgart.de/technik/insights

Umschlaggestaltung: Kerstin Faißt
Bildnachweis: Gerd Altmann auf Pixabay
ISSN 2193-9098

ISSN 2193-9098

www.dhbw-stuttgart.de/technik/insights