Universität Stuttgart

# Änderungstolerante Serialisierung großer Datensätze für mehrsprachige Programmanalysen

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung


Vorgelegt von

Timm Felden

aus Mannheim



**Hauptberichter:** Prof. Dr. Erhard Plödereder

**Mitberichter:** Prof. Dr. Michael Philippsen

**Tag der mündlichen Prüfung:** 6. Dezember 2017



Institut für Softwaretechnologie der Universität Stuttgart



2017

# Inhaltsverzeichnis

# Zusammenfassung

In dieser Arbeit wird ein Serialisierungssystem für Zwischendarstellungen in Programmanalysewerkzeugketten beschrieben, welches auf großen Datensätzen skaliert, geringe Lese- und Schreibkosten hat, typsicher ist, objektorientierte Typdefinitionen enthält, einen allgemeinen Graphen speichert, die Implementierung hinter einem generierten API verbirgt, sprach- und plattformunabhängig ist und eine kompakte serialisierte Darstellung verwendet, die zudem mit Hilfe eines vollständig serialisierten Typsystems eine automatische Bearbeitung von Spezifikationsänderungen ermöglicht. Dies erlaubt bei Erweiterungen der Zwischendarstellungsspezifikation die Weiterverwendung alter Datensätze und in gewissem Umfang auch alter Werkzeuge ohne Anpassung. Inkompatible Änderungen, wie etwa das Entfernen von Supertypen oder das Umtypisieren von Feldern, werden dabei automatisch erkannt und eine Bearbeitung der Datei wird verweigert.

Es wird dabei insbesondere beschrieben, wie die Kombination aus objektorientierter, änderungstoleranter, effizienter und sprachunabhängiger Zwischendarstellung effektiv und sprachübergreifend implementiert werden kann. In einer umfangreichen Evaluation wird die Effizienz der vorgeschlagenen Lösung anhand zahlreicher realer und realitätsnaher Anwendungsfälle demonstriert. Hierbei wird ein breites Spektrum realer Programme als Datensätze verwendet. Dabei zeigt sich, dass die entwickelten Algorithmen und

Datenstrukturen den bisherigen Lösungen in puncto Skalierbarkeit und Laufzeit überlegen sind. Zudem sind erzeugte Dateien für äquivalente Graphen kompakter als die direkt vergleichbarer Lösungen, obwohl Eigenschaften wie Änderungstoleranz und ein vollständig serialisiertes Typsystem hinzugefügt wurden.

# Abstract

This thesis presents a serialization system for program analysis toolchains. The serialization system scales to large datasets, has low-cost read- and write-operations, is type-safe, offers object-oriented type declarations, handles arbitrary object graphs, hides implementation details behind a generated API, and additionally enables an automatic reaction to specification changes using a fully serialized type system. As a consequence, old datasets can be reused if the intermediate representation of a toolchain is extended. Furthermore, old tools can be reused without recompilation. If the intermediate representation is changed by an incompatible modification such as a removal of a super type or retyping of an existing field, old tools will refuse to interact with files of the new intermediate representation. Likewise, old datasets will be refused by tools expecting the new intermediate representation.

This thesis emphasizes the description of an implementation of an intermediate representation that combines the properties of object orientation, change tolerance, efficiency, and language independence. Moreover, it proves that the underlying strategy can be implemented in multiple fundamentally different programming languages. Additionally, an extensive evaluation demonstrates the efficiency of the proposed solution using several real and realistic applications. This demonstration is based on a broad range of real datasets. The evaluation shows that algorithms and data structures proposed

by this thesis are superior to prior solutions in terms of scalability and runtime. Also, files produced by the proposed serialization system are shown to be smaller than those created by competitors for equivalent datasets despite the full serialization of the type system.

# Danksagungen

Zuerst bedanke ich mich bei Erhard Plödereder, der mir mit konstruktiver Kritik bei der Realisierung von SKilL sowie bei der Verfassung dieser Doktorarbeit zur Seite stand. Ferner danke ich Michael Philippsen für die zeitnahe, offene und konstruktiver Kritik an dieser Arbeit.

Außerdem danke ich Thomas Felden und Julia Stolz für die konstruktive Kritik an diesem Dokument sowie Martin Kaistra, Dennis Przytarski und Constantin Michael Weißer für konstruktive Kritik an der Realisierung von SKilL. Ferner danke ich Andreas Lochbihler, Peter H. Schmitt und Mattias Ulbrich für einen herausragenden Einfluss auf meine Ausbildung, ohne den die Konstruktion von SKilL nicht möglich gewesen wäre.

Kapitel 1

# Einleitung

Im Bereich der Übersetzungs- und Programmanalysewerkzeuge ist es üblich, einzelne Bearbeitungsschritte in dedizierte Phasen auszulagern [LA04; Lam87; Was90; App07; GJLB00; WM92; RVP06]. Diese Phasen kommunizieren miteinander über sogenannte Zwischendarstellungen. Kann die Zwischendarstellungen zwischen diesen Phasen serialisiert werden, so können die einzelnen Phasen in separate Werkzeuge gekapselt werden. Diese Werkzeuge werden hintereinander ausgeführt, um die gewünschte Gesamttransformation zu erhalten. Man spricht daher von einer Werkzeugkette.

Generell stellt sich die Frage, warum man versuchen sollte, eine effiziente serialisierbare Zwischendarstellung für Werkzeugketten zu finden und welchen Nutzen man daraus ziehen kann. Zunächst erlaubt die Aufteilung von Berechnungen in einer Werkzeugkette auf einzelne Werkzeuge eine Entkopplung der Implementierungen. Kommunizieren Werkzeuge über ein gemeinsames Dateiformat, also über eine gemeinsame Zwischendarstellung der verarbeiteten Daten, so ist es möglich, diese in separate Programme aufzuspalten.

Zunächst resultiert aus der Aufspaltung eine gesteigerte Ausführungszeit, da hierdurch zusätzlicher Kommunikationsaufwand entsteht. Allerdings ist

es nun möglich, einen Teil der Berechnung, die in ein eigenes Werkzeug ausgelagert wurde, leicht durch eine Alternative zu ersetzen. Dieses Vorgehen ist grundsätzlich auch innerhalb eines Werkzeugs möglich. Handelt es sich hierbei aber um ein abgetrenntes Werkzeug, so kann man bei der Aufspaltung zusätzlich die Implementierungssprache variieren. Das hat zur Folge, dass man leicht auf eine Bibliothek zurückgreifen kann, die ein gegebenes Problem löst, ohne diese an die Implementierungssprache der Werkzeugkette anbinden zu müssen. Zudem kann man bei lange existierenden Werkzeugketten die primär genutzte Programmiersprache wechseln, ohne direkt alle Werkzeuge neu implementieren zu müssen. Das Abtrennen von Werkzeugen erlaubt es dem Werkzeugnutzer außerdem, Zwischenergebnisse einer Berechnung zu behalten und so die letzten Teile der Gesamtberechnung wiederholen zu können, etwa wenn sich ein Teil der Konfiguration geändert hat, um sich in diesem Fall die Ausführungskosten der vorderen Teile zu sparen.

Ist es ferner möglich, die Zwischendarstellung werkzeugabhängig anzupassen, so können entlang eines Ausführungspfads[1] neue Datenstrukturen eingeführt und neue Werkzeuge auf bereits bestehenden Datensätzen getestet werden. Verfügt man über nur schwer oder nicht wiederbeschaffbare Datensätze, so ergibt sich hieraus ein enormer Vorteil. Dabei muss es sich nicht nur um Datensätze handeln, deren Quelle nicht mehr verfügbar ist. Es kann sich dabei auch um Daten handeln, deren Erzeugung mehrere Tage dauert. In dieser Arbeit wird beispielsweise eine mit Andersen [And94] analysierte Version von `bash`[2] verwendet, bei der die Analyse alleine etwa sechseinhalb Tage Rechenzeit erfordert. Müsste man diesen Datensatz aufgrund der Einführung spezieller Änderungen in einem vollkommen unabhängigen Werkzeug ersetzen, wäre der damit verbundene Aufwand schwer zu rechtfertigen. Außerdem wäre es für ein Projekt lähmend, wenn es nicht möglich wäre, im Rahmen eines Experiments am Ende eines Ausführungspfades neue Datenstrukturen einzuführen, mit denen Ergebnisse repräsentiert

[1] Ausführungspfad bezeichnet in dieser Arbeit die Reihenfolge in der Werkzeuge ausgeführt werden.

[2] siehe `http://tiswww.case.edu/php/chet/bash/bashtop.html`

werden können.

Ferner ergibt sich aus einer sprachunabhängigen Zwischendarstellung die Möglichkeit, Prototypen in einer für Prototypisierung geeigneten Programmiersprache zu entwickeln. Ist die Algorithmik entwickelt, so kann man das Werkzeug in eine auf Performanz optimierte Form bringen und dabei gegebenenfalls auch die Implementierungssprache wechseln. Da die Zwischendarstellung dieselbe bleibt, erhält man quasi kostenfrei eine zuverlässige Testumgebung.

Damit dieses Vorgehen wirtschaftlich ist, muss eine Reihe von Voraussetzungen erfüllt sein. Zum einen darf es keinen nennenswerten Programmieraufwand bei der Anbindung einer Zwischendarstellung an ein Werkzeug geben. Daher sollte ein, zur Intermediate Representation (Zwischendarstellung) (IR)-Spezifikation passendes, Application Programming Interface (API) für jede verwendete Programmiersprache automatisch zur Verfügung stehen, was mit einer maschinenlesbaren IR-Spezifikation und Quellcode-Generatoren realisiert werden kann. Zudem entstehen an der Schnittstelle zwischen Werkzeugen Kosten, zum einen in Form der entstandenen Austauschdatei und zum anderen in Form von Laufzeit, die verwendet wird, um die Daten zu schreiben und wieder einzulesen. Um eine Berechnung in viele Werkzeuge aufspalten zu können, muss das Datenformat folglich sowohl kompakt als auch schnell les- und schreibbar sein.

## 1.1. Anforderungen

Die Analyse existierender Zwischendarstellungen und Serialisierungslösungen[1] zeigt, dass der Ansatz, einzelne Phasen in Werkzeuge auszulagern und über eine serialisierbare IR aneinander zu koppeln, an sich zwar sehr gut funktioniert, die Umsetzungen aber Spielraum für grundlegende Verbesserungen haben (siehe §4). Beim Entwurf der in dieser Arbeit vorgestellten

[1] Neben einer Repräsentation von Objektgraphen sind auch eine Spezifikationssprache und ein Konzept zur Anbindung der Serialisierung an die eigentliche Algorithmik eines Werkzeugs erforderlich.

Serialisierungslösung für Zwischendarstellungen wurden daher die folgenden Forderungen berücksichtigt:

1. Um eine IR ohne manuelle Typabbildung repräsentieren zu können, muss es eine ausreichende Auswahl vordefinierter Typen, wie Referenzen, Container, Klassen, Interfaces, Ganzzahlen, Gleitkommazahlen, Strings und definierte Erweiterungspunkte, geben (§5.1). Für die Wahl der Container kann man sich beispielsweise an C++ (siehe [ISO12] §23) orientieren. Da keine Aussage über die konkrete Implementierung des Containers getroffen wird, können die verschieden Set-, Map- und Listenimplementierungen zusammengefasst werden. Zu einem vergleichbaren Ergebnis kommt man, wenn `java.util.Collection` zugrunde gelegt wird (siehe [Ora16]). In beiden Sprachen wird deutlich zwischen Arrays fixer Größe und dynamisch wachsenden Arrays unterschieden, weshalb man diese Unterscheidung übernehmen sollte.

2. Um die Lesbarkeit einer IR-Spezifikation zu verbessern, muss die Spezifikationssprache, im Sinne einer Auslagerung von Teilspezifikationen in eigenen Dateien, modular sein (§5.1.7, [Fel17]). Hierdurch können IR-Spezifikationen mit hunderten Typdefinitionen thematisch sortiert und gruppiert werden, was die Suche innerhalb der IR-Spezifikation erleichtert.

3. Um Datensätze zwischen Werkzeugen unabhängig von Implementierungssprache und ausführender Plattform verwenden zu können, muss das Austauschformat unabhängig von Plattform und Programmiersprache sein (§5.2, §7.3.2, §7.3.4).

4. Ein Programmierer sollte nur über ein generiertes API mit der Zwischendarstellung interagieren können. Hierdurch soll einerseits das notwendige Wissen über die verwendete Serialisierungslösung minimiert werden. Andererseits soll sichergestellt werden, dass Austauschdateien der Zwischendarstellung zu deren Spezifikation konsistent sind. Zudem wird es Programmierern ermöglicht, Werkzeuge in der Programmiersprache zu schreiben, die sie am besten beherrschen oder

für die ihnen die besten Bibliotheken zur Verfügung stehen (§5.1.5, §5.3, [FP16]).

5. Eine typsichere Darstellung von Referenzen auch auf unbekannte Typen ist erforderlich, damit zum einen keine manuelle Arbeit in eine Darstellung von Referenzen investiert werden muss und zum anderen falsche Benutzung von Feldern vor dem Schreiben einer Austauschdatei erkannt und verhindert werden kann, selbst wenn die Programmiersprache, in der ein Werkzeug implementiert ist, dafür keinen Mechanismus bietet (§5.2). Ein Beispiel für eine falsche Benutzung wäre etwas das Speichern einer Gleitkommazahl in einem als Referenz typisierten Feld.
6. Um auf möglichst viele Änderungen der IR automatisch reagieren zu können, soll das Austauschformat ein Maximum an Aufwärts- und Abwärtskompatibilität gewährleisten (§5.2.6.4, §7.5.3, [Fel17] §4.3.2).
7. Eine vollständige Serialisierung des Typsystems der IR ist erforderlich, um auf Änderungen der Zwischendarstellung reagieren zu können. Ferner lässt sich so zuverlässig eine inkompatible Änderung der Typisierung einer Austauschdatei feststellen (§5.2.6, [Fel17] §7). Die Frage, welche Änderungen in-/kompatibel sind, kann an dieser Stelle offen bleiben, solange sie irgendwann eindeutig festgelegt wird.
8. Um Austausch und Archivierung von Daten zu erleichtern, sollen Daten im Austauschformat möglichst kompakt repräsentiert werden (§7.8.1).
9. Um die Kommunikation zwischen Werkzeugen zu beschleunigen, muss das Serialisierungssystem eine hohe Transferrate aufweisen. Das Design sollte im Zweifelsfall die Leserate vor der Schreibrate optimieren, um den Test von Werkzeugen zu erleichtern (§7.3.1, §7.8.3).
10. Um Analysen zu beschleunigen, die nur auf einem Teil der IR operieren, sollen serialisierte Daten, die nicht verwendet werden, auch nicht gelesen werden (§7.8.3). Dieser Ansatz wird bedarfsorientiertes Lesen genannt.

11. Um kleine Ergänzungen, z.B. durch eine Zeigeranalyse, ohne ein komplettes Neuschreiben des Graphen zu realisieren, sollen neue Objekte und Felder an existierende Austauschdateien angehängt werden können (§7.4).
12. Damit in einer Werkzeugkette kein Werkzeug existieren muss, das die gesamte Spezifikation der IR kennt, ist keine globale Spezifikation der IR erforderlich. Dies ermöglicht zudem, in gemeinsam genutzten Typen unterschiedliche Felder zu spezifizieren, ohne dass dies einen Einfluss auf die Kompatibilität hat, solange die tatsächlich zur Kommunikation benötigten Felder gleiche Namen und Typen haben.

Diese Anforderungen wurden vollständig umgesetzt. Es wurde zudem darauf geachtet, eine möglichst leicht erlern- und benutzbare Spezifikationssprache zu definieren, um den Einarbeitungsaufwand zu minimieren. Dieser Punkt ist nicht als Anforderung formuliert, da sich ein Beleg durch Experimente schwierig gestaltet und daher im Rahmen dieser Arbeit nicht erfolgt ist. Dennoch beschäftigt sich §7.5 mit diesem Thema. Der Name des hier vorgestellten Ansatzes lautet Serialization Killer Language (SKilL).

Es wurden bei der Umsetzung von SKilL die folgenden Annahmen gemacht:

1. Alle Ergebnisse einer Berechnung zu einer Eingabe werden in einer Austauschdatei, über welche die Werkzeuge miteinander kommunizieren, abgelegt.
2. Ein Werkzeug wird in einem Prozess ausgeführt und hält alle notwendigen Daten im Hauptspeicher.
3. Alle Werkzeugaufrufe erfolgen sequentiell, d.h. eine Synchronisation auf Formatebene ist nicht erforderlich.

Diese Annahmen stellen vor allem eine Abgrenzung zum Bereich der Datenbanken dar.

## 1.2. Forschungsfragen

Diese Arbeit widmet sich folgenden Fragestellungen:

- Welche existierenden Techniken sind für die Repräsentation einer Zwischendarstellung in Programmanalysewerkzeugketten geeignet? (§4)
- Welche Eigenschaften muss eine Technik haben, die die Anforderungen aus Abschnitt 1.1 erfüllt? (§3, §6)
- Wie kann eine derartige Technik implementiert werden? (§6)
- Skaliert die Implementierung für IR-Spezifikationen echter Werkzeugketten? Skaliert die Implementierung für echte Eingabedaten? (§7)
- Wie wirkt sich die Möglichkeit von bedarfsorientiertem Lesen auf die Dateigröße (§5.2.6, §7.8.1f) und den Ressourcenverbrauch (§7.4) aus?
- Wie wirkt sich die Kombination von bedarfsorientiertem Lesen und inkrementellem Schreiben auf die Ausführungszeit aus? (§7.4, §7.8.3)

Kapitel 2

# Grundlagen

Dieses Kapitel widmet sich den Grundlagen der Arbeit. In Abschnitt 2.1 wird der Werkzeugkettenbegriff und die damit zusammenhängende Architektur von Programmanalysewerkzeugen erläutert. In Abschnitt 2.2 wird auf Aspekte der Änderungstoleranz näher eingegangen. In Abschnitt 2.3 wird das der Arbeit zugrunde liegende Verständnis von Objektorientierung kurz zusammengefasst.

## 2.1. Werkzeugketten

In diesem Abschnitt wird das der Arbeit zugrunde liegende Verständnis von Werkzeugketten zusammengefasst. Hierbei wird auch begründet, warum und wann die vorgeschlagene Architektur für Analysewerkzeuge sinnvoll ist.

### 2.1.1. Aufbau von Werkzeugketten

In dieser Arbeit wird davon ausgegangen, dass eine Werkzeugkette eine Sammlung an Werkzeugen ist. Die Kette entsteht durch die Weiterverarbeitung von Ergebnissen anderer Werkzeuge und die damit verbundene

Datenabhängigkeit. Im Allgemeinen können Zwischenergebnisse auch an mehrere Werkzeuge weitergegeben oder aus den Ergebnissen mehrerer vorgelagerter Werkzeuge berechnet werden. Daher handelt es sich bei diesen Werkzeugen genau genommen zwar um einen gerichteten azyklischen Graphen, der Begriff *gerichteter azyklischer Werkzeuggraph* ist aber zu sperrig und nicht gebräuchlich, weswegen der Begriff Werkzeugkette verwendet wird.

Weiterhin zielt diese Arbeit auf eine Verbesserung der Entwicklung von Werkzeugketten zur Programmanalyse ab. Die Skalierbarkeit der Analysen ist dabei von zentraler Bedeutung. Es sollen sowohl kleine als auch große Programme effizient analysierbar sein. Dementsprechend ist davon auszugehen, dass die Laufzeit einzelner Werkzeugausführungen zwischen wenigen Millisekunden und mehreren Tagen variiert. Entwickelt man ein Werkzeug, welches auf einer Analyse basiert, deren Berechnung mehrere Tage dauert, so ist es nicht akzeptabel, diese bei jedem Testlauf zu wiederholen. Ebenso inakzeptabel erscheint es, eine Analyse wiederholen zu müssen, weil das nachfolgende Werkzeug in der Zwischendarstellung weitere Typdefinitionen einführt. Ferner ist es wünschenswert, dass ein Werkzeug, welches nur wenige Millisekunden analysiert, auch nur ebenso lange für die Serialisierung der Zwischendarstellung benötigt.

Zur Veranschaulichung ist eine vereinfachte Werkzeugkette für Forschungszwecke in Abbildung 2.1 dargestellt. Dabei folgt der vorgesehene Ausführungspfad den durchgezogenen Pfeilen. Da es sich um einen Entwicklungsprozess handelt, bei dem im Allgemeinen nicht zu erwarten ist, dass sich alle ursprünglichen Annahmen direkt bestätigen, wird man potentiell hinter jedem Schritt Debug-Ausgaben benötigen, um die tatsächlichen Ergebnisse mit den Erwartungen vergleichen zu können. Menschenlesbar sind dabei nur die Ausgaben der *Debug-Ausgabe* und der *Hübschen Publikationsdarstellungsausgabe*.

Zur Erforschung und Bewertung von Analysen ist es oft erforderlich, alternative Analysemethoden zu vergleichen. Wenn man z.B. eine Zeigeranalyse in einzelnen Werkzeugen implementiert hat, so kann man diese, die gleiche Darstellung der Ergebnisse vorausgesetzt, leicht auf den Analysepfaden

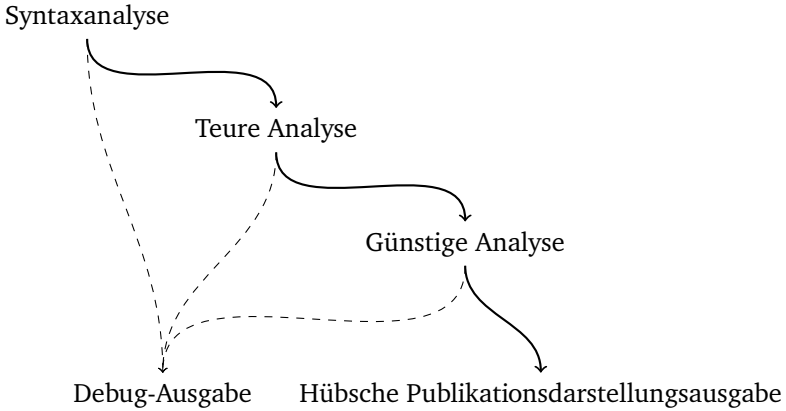



Abbildung 2.1.: Eine einfache Werkzeugkette

austauschen und so den Effekt auf eine nachgelagerte Analyse untersuchen.

Hat man diese Alternativen einmal implementiert, so wird man feststellen, dass die potentiell präziseren Ergebnisse nicht immer einen Mehrwert bieten, der den Mehraufwand rechtfertigt. Insbesondere wenn man ein System analysieren möchte, das so umfangreich ist, dass die Analyse selbst zu ressourcenintensiv wäre, ist es hilfreich, auf günstigere Alternativen zurückgreifen zu können.

Es ist dabei leichter, Analysen am unteren Ende zu ersetzen, da hier die Auswirkungen auf andere, bereits existierende, Werkzeuge geringer ist. Dies ist aber ohnehin der Punkt, der bei einer Untersuchung durch Prototypen und kleine Experimente primär von Interesse ist. Dadurch entsteht eine Situation, in der die oberen Teile automatisch stabiler und langlebiger werden, während die unteren Teile der Werkzeugkette kurzlebig und veränderbar sind.

In einer sich verändernden Werkzeugkette eine Zwischendarstellung zu konstruieren ist ein schwieriges Problem. Zunächst sollte hierzu relativ zentral spezifiziert werden, welche Datenstrukturen ausgetauscht werden. Ferner sollte die Semantik der Daten spezifiziert werden. Gäbe es nur eine einzige zentrale IR-Spezifikation, so würde das Problem entstehen, dass alle

jemals verwendeten Datenstrukturen jedem Werkzeug bekannt sind.

Tatsächlich benötigt man für jedes Werkzeug eine klare Spezifikation der vom Werkzeug bearbeiteten Zwischendarstellung. Da Werkzeuge in weiten Teilen auf gemeinsamen Datenstrukturen operieren, sollten deren IR-Spezifikation generell zentral organisiert und bei der Erzeugung der IR-Spezifikation einzelner Werkzeuge darauf zurückgegriffen werden.

Um die Arbeit auf der Zwischendarstellung zu erleichtern, wird ein zwischendarstellungsspezifisches API benötigt, welches die Interaktion mit den Daten kapselt. Dabei ist wichtig, dass das API möglichst auf die Bedürfnisse eines spezifischen Werkzeugs zugeschnitten ist. Außerdem ist es unerlässlich das API so zu gestalten, dass die Interaktion mit den Datenstrukturen der Zwischendarstellung möglichst performant ist, da sich dies maßgeblich auf die Ausführungszeit auswirkt. Ist man bereit, sich bei der Implementierung einer Werkzeugkette auf eine bestimmte Programmiersprache festzulegen, so kann man die Implementierung der Zwischendarstellung in dieser Sprache sowohl als Spezifikation als auch als API der Anbindung verwenden.

Verwendet man hingegen eine eigene Spezifikationssprache und einen Quellcode-Generator um ein API zu generieren, ist es möglich, ein Werkzeug unter Zuhilfenahme von geeigneten Bibliotheken in einer anderen Programmiersprache zu implementieren. Hierdurch kann mit geringem Aufwand geprüft werden, ob dadurch Verbesserungen gegenüber einem Werkzeug der Werkzeugkette erzielt werden können. Der Erfolg lässt sich dabei leicht messen und das bessere Werkzeug kann behalten und fortan benutzt werden.

### 2.1.2. Ausführungsmodell

Dem in dieser Arbeit verwendeten Begriff von Werkzeugketten liegt ein Ausführungsmodell zu Grunde, welches die folgenden Eigenschaften hat:

- Berechnungen werden entlang eines Pfades ausgeführt. Der Werkzeugabhängigkeitsgraph ist gerichtet und azyklisch.[1] Die Zahl der

[1] Diese Einschränkung auf azyklische Abhängigkeiten ist jedoch ultimativ aus Sicht des hier entwickelten Ansatzes nicht von Belang. Es ist allerdings zu bedenken, dass im Fall zyklischer Abhängigkeiten ein Ausführungsmodell der Werkzeuge nicht mehr naheliegend ist und aus diesem Grund auch nicht näher untersucht wurde.

Wurzelknoten ist dabei irrelevant. Jede tatsächlich durchführbare Analyse kann durch Linearisierung aller Werkzeugaufrufe realisiert werden. Konkret heißt dies, dass es einen Zusammenfluss von Ergebnissen geben kann, wie das etwa bei einem Binder (englisch Linker) der Fall ist.

- Werkzeugaufrufe sind wiederholbar, vorausgesetzt man besitzt eine Kopie der Eingabedaten.
- Werkzeuge können auf unterschiedlichen Maschinen ausgeführt werden.
- Werkzeuge können in unterschiedlichen Programmiersprachen implementiert sein.
- Werkzeuge können neue Informationen an bestehende Austauschdateien anhängen, wie etwa eine SSA-Darstellung an einen AST.

### 2.1.3. Entwicklung von Werkzeugketten

In diesem Abschnitt wird kurz ein Entwicklungsprozess neuer Werkzeuge skizziert, um die Rolle der hier entwickelten Zwischendarstellungslösung zu erläutern. Hierbei wird die in Abb. 2.1 dargestellte Werkzeugkette als Beispiel wiederverwendet.

Die Entwicklung beginnt mit dem ersten Werkzeug – in diesem Beispiel einer Syntaxanalyse. Da es sich um das erste Werkzeug handelt und die Eingabe hier nicht in Form der Zwischendarstellung erfolgt, kann die Entwicklung zunächst ohne Zwischendarstellung begonnen werden. Ist die Entwicklung soweit, dass auf eine Zwischendarstellung abgebildet werden soll, sind einige Arbeitsschritte zu erledigen. Als Erstes ist eine IR-Spezifikation zu erstellen, welche alle Datentypen der in der Zwischendarstellung gespeicherten Objekte beschreibt. Danach kann diese IR-Spezifikation genutzt werden, um sie mithilfe eines Quellcode-Generators in eine Anbindung zu überführen, welche vom Werkzeugbauer genutzt werden kann, um den Anwendungscode darauf aufzubauen. Die Werkzeugimplementierung wird dann um Aufrufe des generierten Codes so erweitert, dass dieser letztlich

zur IR-Spezifikation passende Austauschdateien erzeugen kann. An diesem Punkt ist die Entwicklung der Syntaxanalyse zunächst abgeschlossen.

Als Nächstes wird das Werkzeug *Teure Analyse* entwickelt. Das Vorgehen ist dabei weitgehend analog zum ersten Werkzeug. Falls die Analyse eine umfangreichere Zwischendarstellungsspezifikation benötigt, wird diese zunächst erweitert. Dann wird daraus wieder eine Anbindung generiert, mit Hilfe derer die teure Analyse auf der Zwischendarstellung implementiert werden kann. Bei der Entwicklung treten natürlich diverse Probleme auf. Besonders herausfordernd sind dabei Datenmengen, welche zu groß sind, als dass sie von Mensch ungefiltert verstanden werden könnten. Um hier Fehler gut beheben zu können, wird im Rahmen des Entwicklungsprozesses das Werkzeug *Debug-Ausgabe* entstehen, welches den Inhalt der Zwischendarstellung für den Werkzeugbauer übersichtlich darstellt. Die so entstandenen Debug-Ausgaben können nach Abschluss des Entwicklungsprozesses weggeworfen werden.

Nach Abschluss der Arbeiten an der teuren Analyse soll eine neue, günstige Analyse entwickelt werden, welche auf der teuren aufbauen soll und deren Resultate in irgendeiner Form verbessert. Es wird wieder die IR-Spezifikation erweitert, eine Anbindung generiert und die Analyse implementiert. Falls es sich bei der günstigen Analyse um ein neues Verfahren handelt, ist davon auszugehen, dass bei der Entwicklung festgestellt wird, dass die hier notwendige Erweiterung der Zwischendarstellung mehrfach angepasst werden muss. Diesen Entwicklungsprozess kann man durch zwei Eigenschaften erheblich beschleunigen. Einerseits ist es hilfreich, wenn Erweiterungen der Zwischendarstellung durch ein Werkzeug am Ende der Werkzeugkette keine Anpassungen der vorgelagerten Werkzeuge erfordert, und andererseits ist es ausgesprochen hilfreich, wenn vorhandene Datensätze hierdurch nicht ungültig werden, da hierdurch das Testen und damit auch die Entwicklung neuer Werkzeuge am Ende der Kette weitaus effizienter wird und auf einem breiteren Bereich von Testdaten erfolgen kann.

Hat man erfolgreich eine neue günstige Analyse entwickelt, die das Ergebnis der teuren Analyse deutlich verbessert, so wird man dies der Welt mitteilen wollen. Weil die Welt üblicherweise nicht an den Datensätzen

der hier entstandenen Zwischendarstellung interessiert ist, sondern eine übersichtliche und leicht lesbare Darstellung des Erreichten als Beweis der Effektivität bevorzugt, entwickelt man ein weiteres kleines Werkzeug, welches die Ergebnisse der günstigen Analyse auswertet und in eine publizierbare Form bringt.

Aus Sicht des Werkzeugbauers besteht die Rolle der im Rest der Arbeit vorgestellten Lösung darin, die Anbindung der Zwischendarstellung zu realisieren. Hierfür wird eine Spezifikationssprache definiert, zu der ein Quellcode-Generator existiert, welcher Implementierungen und benutzbare APIs in mehreren verbreiteten Programmiersprachen generieren kann. Die Implementierung verwendet ein im Folgenden beschriebenes Austauschformat. Daneben gibt es einige hilfreiche Werkzeuge und Eigenschaften, die den gerade beschriebenen Prozess effektiver gestalten sollen. Diese werden im Hauptteil der Arbeit beschrieben. Das Ziel dabei ist es, den Entwicklungsprozess zu beschleunigen und auf Werkzeugketten mit sehr vielen Werkzeugen skalierbar zu machen, ohne dabei wichtige Parameter wie die Ausführungskosten eines Analysepfades zu beeinträchtigen.

## 2.2. Änderungstoleranz

In diesem Abschnitt wird das der Arbeit zugrunde liegende Verständnis von Änderungstoleranz dargestellt. Hierfür ist zunächst festzulegen, was genau geändert wird. Es gibt im Rahmen der Serialisierung zwei Ebenen, die sich ändern können: Die Daten und die Typisierung der Daten. Eine Änderung der Daten würde beispielsweise einer Änderung eines analysierten Programms entsprechen. Änderungen an Daten spielen hier nur eine untergeordnete Rolle. Diese Arbeit verwendet den Begriff Änderungstoleranz für die Anpassung an Veränderungen der Typisierung. Dabei wird sowohl die Veränderung der Typisierung aus Sicht der Werkzeug-, als auch der Dateispezifikation betrachtet.

Die Änderungen der Typisierung sind zunächst interessanter, da sich diese direkt auf die Wartungskosten einer Werkzeugkette auswirken, für Erwei-

terungen einer Zwischendarstellung jedoch erforderlich sind. Das Ziel im Sinne der Wartungskosten ist es, dass bestehende Werkzeuge trotz Änderungen an der Zwischendarstellung ohne Anpassung weiter verwendet werden können. Dieses Ziel ist in dieser Allgemeinheit nicht zu erreichen, da beispielsweise rein semantische[1] Änderungen nicht automatisch erkennbar sein können. Damit es Werkzeugen überhaupt möglich sein kann, auf Änderungen tolerant zu reagieren, muss die zugrundeliegende Zwischendarstellung diese Änderungen tolerieren können. Würde beispielsweise das Hinzufügen eines weiteren Attributs schon beim Deserialisieren der Austauschdatei zu einem Fehler führen, könnte das Werkzeug selbst nicht entscheiden, ob dieses Attribut problematisch ist oder nicht. Um auf diese Änderungen automatisch reagieren zu können, müssen diese beim Lesen erkannt und von der Anbindung an die Zwischendarstellung automatisch behandelt werden. Ebenso darf es bei der Deserialiserung nicht zu Abstürzen oder undefiniertem Verhalten kommen, falls sich die Typisierung geändert hat. Eine änderungstolerante Reaktion auf die Deserialisierung eines inkompatiblen Datensatzes endet in einer definierten Fehlerbehandlung. Für diese Forderung ist nicht relevant, wie inkompatible Typisierungen definiert sind.

Da Änderungen unterschiedliche Konsequenzen haben, werden im Folgenden Gruppen von Änderungen eingeführt. Diese gehen von einer objektorientierten Zwischendarstellung mit namensäquivalenten Typen und Feldern aus, d.h. es wird über den Namen entschieden, ob sich ein Typ bzw. Feld geändert hat. Ferner basiert der hier verwendete Änderungstoleranzbegriff auf der Forderung, dass die Typisierung einer Austauschdatei, die von einem Werkzeug eingelesen wird, erhalten und gegebenenfalls ergänzt wird. Dadurch wirkt sich eine Anpassung an einem Werkzeug auf alle nachgelagerten Werkzeuge aus und wird nicht von der Ausführung des nächsten Werkzeugs absorbiert. Es wird also angenommen, dass eine Änderung an einem Werkzeug bzw. dessen Zwischendarstellung immer auch für alle nachgelagerten Werkzeuge und nicht nur für das geänderte Werkzeug gelten soll. Unter

[1] Also Änderungen, welche die Interpretation von Datenstrukturen betreffen ohne ihre Typisierung zu ändern, wie etwa eine Änderung der Zulässigkeit von Zyklen über einer Attributbeziehung.

Verwendung einer globalen IR-Spezifikation ist diese Eigenschaft immer erfüllt. Auf Beispiele für die im Folgenden aufgeführten Änderungen wird in §7.7 eingegangen.

**Veränderung des Typs eines Feldes** Diese Änderung entspricht einer Typkonversion beim Lesen und Schreiben. Würde man alle wertkompatiblen Änderungen von Feldtypen akzeptieren, so müsste der Wertebereich beim Lesen oder Schreiben für jeden Wert geprüft werden. Invarianz zu fordern, vermeidet diese Prüfungen. Im Fall von Typänderungen wie etwa zwischen Zahlen und Zeigern ist eine automatische Anpassung nicht sinnvoll, da diese von logischen Umstrukturierungen wie einer Umstellung von Indices in eine Datenstruktur auf direkte Verzeigerung hervorgerufen werden.

**Hinzufügen eines Feldes** Das Hinzufügen eines Feldes erfordert, dass ein Werkzeug mit einem unbekannten Feld arbeiten kann und dieses am Ende seiner Arbeit exportiert, damit es von nachgelagerten Werkzeugen verwendet werden kann. Problematisch ist in diesem Zusammenhang, falls das nachgelagerte Werkzeug neue Objekte anlegt, die dieses Feld haben müssten. Hierdurch gibt es Objekte, die dem Feld berechnete Werte zuordnen, während die neuen Objekte dem Feld höchstens Standardwerte zuordnen können. Hingegen ist das Hinzufügen neuer Felder mit eindeutigem Namen generell unproblematisch, falls in nachgelagerten Werkzeugen keine neuen Objekte des entsprechenden Typs erzeugt werden.

**Entfernen eines Feldes** Das Entfernen eines Feldes ist zunächst symmetrisch zum Hinzufügen, da ein Werkzeug, welches innerhalb einer Werkzeugkette ein Feld hinzufügt, eine Zwischendarstellung einliest, in der das Feld fehlt, sprich entfernt wurde. Bei dieser Änderung ist wichtig zu beachten, dass die Umbenennung eines Feldes bei Namensäquivalenz zu einer Entfernung des alten Feldes und dem Hinzufügen des logisch selben Feldes unter neuem Namen entspricht. Hier ist ein manueller Eingriff folglich erforderlich. Die Verschiebung von Feldern innerhalb der Typhierarchie verhält sich

wie eine Umbenennung, falls für diesen Fall keine Sonderregeln eingeführt werden (in dieser Arbeit gilt Verschiebung = Umbenennung).

Um eine logisch inkompatible Änderung durch das Entfernen eines Feldes ohne Anpassung eines Werkzeugs erkennen zu können, muss dieses erkennen können, dass ein Feld nicht vorhanden war.

**Veränderung einer Supertyp-Beziehung** Im Allgemeinen ist diese Art der Änderung problematisch. Verschiebt man hierdurch beispielsweise einen Typ in einer Typhierarchie so, dass sich die geerbten Felder unterscheiden, so ist nicht davon auszugehen, dass ein Werkzeug ohne Anpassungen weiter funktioniert. Das Einfügen oder Entfernen von Typen innerhalb einer Typhierarchie führt zu dem Problem, dass Instanzen dieses Typs vom Werkzeug nicht korrekt repräsentiert werden können und daher eine Anpassung des Werkzeugs erforderlich ist.

**Hinzufügen eines Typs** Das Hinzufügen eines Typs ist unproblematisch, falls ein Werkzeug in der Lage ist, den neuen Typ in die bestehende Typisierung zu integrieren. Neue Typen, deren Supertyp dem Werkzeug unbekannt sind, stellen dabei kein Problem dar. Bei Typen mit bekannten Supertypen besteht das Problem, dass im Werkzeug stattfindende Typprüfungen korrekt realisiert werden müssen. Wird ein konkreter Typ unterhalb eines, aus Sicht des Werkzeugs abstrakten, Typs eingeführt, so ist nicht zu erwarten, dass das Werkzeug weiter korrekt funktioniert. Analog zu neuen Feldern muss auch für Instanzen neuer Typen deren Weitergabe an nachgelagerte Werkzeuge sichergestellt werden.

**Entfernen eines Typs** Das Entfernen eines Typs ist analog zum Entfernen eines Feldes.

### 2.2.1. Änderung bestehender Datensätze

Die Änderung von Datensätzen ist generell unproblematisch, falls man bei der Serialisierung eine neue Austauschdatei erzeugt. Verändert man eine

bestehende Austauschdatei, so stellt man fest, dass es unproblematisch ist, neue Instanzen, Felder oder Typen hinzuzufügen, falls das Dateiformat dies zulässt. Eine Veränderung der Vererbungsbeziehung oder der Typisierung von Feldern würde dabei zu einer inkonsistenten Austauschdatei führen und ist im Sinne des hier verwendeten Änderungsbegriffs somit unzulässig.

Das Anhängen von Änderungen bzw. Löschung bestehender Instanzen, Felder und Typen an Dateien ist prinzipiell denkbar. Dieses Vorgehen hat jedoch den Nachteil, dass eine Austauschdatei dadurch tote Daten enthält, die beim Lesen interpretiert werden müssen. Da das Anhängen dieser Änderungen an Austauschdateien weder die Wartung noch die Entwicklung von Werkzeugketten beeinflusst,[1] handelt es sich hierbei ohnehin um ein Randthema.

### 2.2.2. Anwendungsfälle

In diesem Abschnitt wird kurz auf verschiedene Änderungen an Werkzeugketten und die damit verbundenen Wartungsarbeiten eingegangen. Da Anpassungen an einem bestehenden Werkzeug immer eine Prüfung der semantischen Konsistenz der Resultate aus Sicht nachgelagerter Werkzeuge erfordert, beschränken sich die interessanten Fälle auf das Hinzufügen neuer Werkzeuge.

**Neues Werkzeug am Ende einer Werkzeugkette** In diesem Fall kommt es idealerweise zu keinem Wartungsaufwand, weil ein neues Werkzeug am Ende eines Ausführungspfades die Ausführung keines anderen Werkzeugs beeinflusst. Dabei ist die dadurch vorgenommene Änderung an der Zwischendarstellung unerheblich. Bei mehrmaligem Hinzufügen am Ende einer Werkzeugkette gilt diese Betrachtung nur für das letzte Werkzeug. Für die vorangegangenen Werkzeuge gelten die direkt nachfolgenden Betrachtungen zum Hinzufügen innerhalb der Werkzeugkette.

[1] Siehe §7.4 / Tab. 7.19.

**Neues Analysewerkzeug innerhalb der Werkzeugkette** Analysewerkzeuge sind in dieser Arbeit Werkzeuge, welche die Zwischendarstellung höchstens durch das Berechnen neuer Attribute verändern. Dabei dürfen auch neue Objekte entstehen, solange der bestehende Objektgraph dadurch nur erweitert wird. Ein Beispiel für ein Analysewerkzeug ist eine Zeigeranalyse. Das reine Hinzufügen neuer Informationen ist unproblematisch, falls bestehende Informationen erhalten bleiben. Werden hierbei keine Instanzen bestehender Typen erzeugt, so führt eine Änderungstoleranz der Zwischendarstellung direkt zu einem änderungstoleranten Werkzeug. Das Einführen neuer Attribute erfordert, dass geprüft werden muss, ob nachgelagerte Werkzeuge diese invalidieren, was dort Wartungsaufwand verursachen kann. Eine solche Invalidierung kann beispielsweise erfolgen, indem ein nachgelagertes Werkzeug eine Transformation auf dem Knotentypen durchführt, der das Attribut trägt (z.B. Entfernung von totem Code).

**Neues Transformationswerkzeug innerhalb der Werkzeugkette** Transformationswerkzeuge sind in dieser Arbeit Werkzeuge, welche den bestehenden Objektgraph strukturell verändern. Ein Beispiel für ein Transformationswerkzeug ist die Entfernung toten Codes. Das Hinzufügen eines Transformationswerkzeugs erfordert, dass die Transformation Attribute vorgelagerter Analysen semantisch erhält, falls diese von nachgelagerten Werkzeugen verwendet werden.[1]

## 2.3. Definition objektorientierter Daten

In diesem Abschnitt werden die in dieser Arbeit verwendeten Begriffe und Vorstellungen von objektorientierten Daten definiert. Zur Vereinfachung wird angenommen, dass es nur objektorientierte Typen gibt. Es folgt eine Reihe von Definitionen, welche die Objektorientierung genauer charakterisieren.

[1] Die bloße Markierung toten Codes kann hingegen als Analyse realisiert werden.

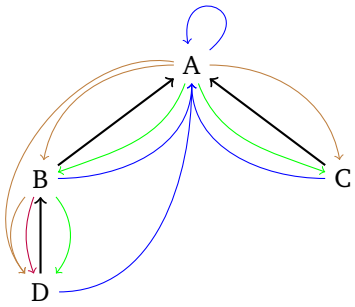



Abbildung 2.2.: Veranschaulichung von Super-, Sub- und Basistyp.

**Vererbung** Für alle Typen $t, s$, und alle Felder $f$ von $s$ gilt, falls $t$ ein Subtyp von $s$ ist, dann ist $f$ auch ein Feld von $t$.

**Vererbarkeit** Für alle Typen $t, s, u$, wobei $t$ Subtyp von $s$ ist und ein Feld $f$ von $s$ mit Typ $u$ existiert, gilt, dass die Methoden $getF$ und $setF$ so definiert werden können, dass die Verwendungen $t.getF() : u$, $t.setF(u)$, $s.getF() : u$, sowie $s.setF(u)$ legal sind. Mit ": $u$" ist hier eine Typzusicherung gemeint, d.h. das Resultat kann wie ein $u$ verwendet werden.

**Bezeichnungen für Typbeziehungen** Es werden die Begriffe *Supertyp*, *Subtyp* und *Basistyp* verwendet, um den direkten Supertyp, einen direkten Subtyp oder den entferntesten transitiven Supertyp zu bezeichnen. Transitive Supertypen sind die positive Hülle der Supertyprelation. Der Basistyp eines Typs ohne Supertyp ist der Typ selbst. Haben zwei Typen denselben Supertyp, so haben sie auch denselben Basistyp. Auf der Menge der Typen sind daher Supertyp eine partielle Funktion, Subtyp eine binäre Relation und Basistyp eine totale Funktion. Diese Beziehung ist in Abb. 2.2 veranschaulicht.

**Instanzen** *Statische Instanzen* eines Typs T sind alle Objekte, die als T allokiert wurden. *Dynamische Instanzen* eines Typs T sind alle Objekte, die als T benutzt werden können. Durch die Historie von SKilL bedingt wird der Begriff *statische Instanzen* für alle direkten Instanzen eines Typs und *dynamische Instanzen* für statische Instanzen und dynamische Instanzen von Subtypen verwendet. Dieser Begriff ist aus der Implementierung der Speicherverwaltung in SKilL hervorgegangen. Hier entsprechen die statischen Instanzen exakt jenen Instanzen, deren statischer und dynamischer Typ identisch ist.

#### Repräsentation von Eigenschaften durch abstrakte Klassen

Es stellt sich die Frage, ob es nicht ausreichend ist, in einer Vererbungshierarchie nur die Blattknoten zu serialisieren und somit auf Vererbung im Datenformat zu verzichten. Obgleich es vielleicht guter Stil wäre, in einer Vererbungshierarchie nur konkrete Klassen in den Blättern zu verwenden, muss ein objektorientiertes Datenformat mit allen gängigen Verwendungen von Objektorientierung umgehen können. Da es durchaus IRs gibt, die konkrete und instantiierte Klassen im Inneren der Vererbungshierarchie aufweisen, muss damit umgegangen werden können.

## 2.4. Verwendung von Begriffen

In diesem Abschnitt werden weitere Begriffe definiert, deren Verwendung möglicherweise nicht unmittelbar klar ist oder von den gängigen Konventionen leicht abweicht. So ist man beispielsweise mit dem Problem konfrontiert, dass es sich bei einem Quellcode-Generator, welcher eine IR-Spezifikation in eine IR-Implementierung für ein Werkzeug übersetzt, um einen Compiler handelt. Der Compiler, der das Werkzeug in ein ausführbares Programm übersetzt ist natürlich ein anderer Compiler. Das übersetzte Programm selbst kann natürlich auch ein Compiler sein. Daher werden in dieser Arbeit Begriffe verwendet, die hier Klarheit schaffen sollen. In Abbildung 2.3 wird der Zusammenhang zwischen häufig verwendeten Begriffen visuell dargestellt.

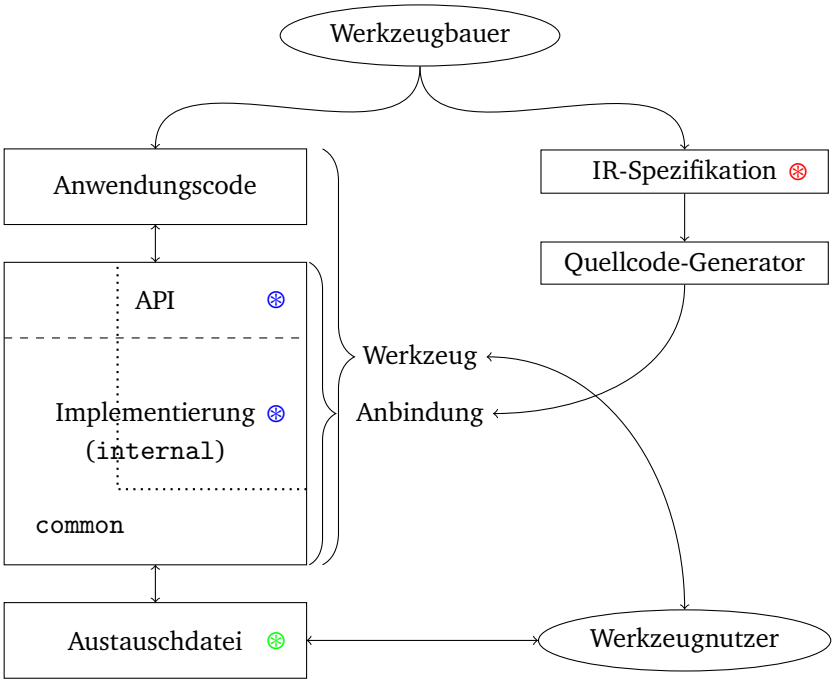



Abbildung 2.3.: Landkarte verwendeter Begriffe. Die gestrichelte Linie ist die Abgrenzung zwischen API und Implementierung. Die gepunktete Linie trennt die generierten von den gemeinsam genutzten Code-Teilen einer Anbindung. Mit ⊛ werden verschiedene Ausprägungen der IR-Spezifikation markiert: `.skill`-Spezifikation, Werkzeugspezifikation, Dateispezifikation

**Quellcode-Generator** Der Übersetzer der IR-Spezifikation in eine sprachspezifische Implementierung (Anbindung) wird im Folgenden Quellcode-Generator genannt. Ist dieser nicht näher spezifiziert, so handelt es sich dabei um den SKilL-Quellcode-Generator.

**Werkzeug** und **Werkzeugkette** Ein Programm, welches auf einer IR arbeitet wird Werkzeug genannt. Im Rahmen dieser Arbeit wird davon ausgegangen, dass ein Werkzeug eine oder mehrere Analysen oder Transformationen auf der IR implementiert. Eine Sammlung von Werkzeugketten mit gemeinsamer IR wird Werkzeugkette genannt. Handelt es sich bei dieser Werkzeugkette um einen Compiler, so ist dies in dieser Arbeit nicht von Belang.

**Anbindung** Als Anbindung wird die Abstraktionsschicht zwischen Analyseimplementierung (Anwendungscode) und dem Binärformat bezeichnet. Die Anbindung besteht sowohl aus dem API als auch der dahinter stehenden Implementierung (`internal`). Anbindungen haben als Parameter ein Serialisierungssystem und eine Programmiersprache. Eine SKilL-Java-Anbindung wird SKilL/Java genannt, um sie in der Gegenwart anderer Serialisierungssysteme abzugrenzen. Der Quelltext einer Anbindung wird von einem Quellcode-Generator-Back-End erzeugt.

**API** Mit API wird in dieser Arbeit die Schnittstelle zwischen Werkzeug und IR-Implementierung bezeichnet. Sind andere APIs gemeint, wird dies ausdrücklich zum Ausdruck gebracht.

**Common** Als `common` wird der spezifikationsunabhängige Teil einer Anbindung bezeichnet.

**Werkzeug**- und **Dateispezifikation** In dieser Arbeit werden sich ändernde IR-Spezifikationen betrachtet. Dies führt dazu, dass sich die Spezifikationen,

mit denen Datensätze erzeugt und Werkzeuge übersetzt wurden, unterscheiden können. Um die entsprechenden potentiell unterschiedlichen Spezifikationen zu identifizieren, werden für die Spezifikation der verarbeiteten Austauschdatei der Begriff Dateispezifikation und für die Spezifikation, aus der die Anbindung des Werkzeugs generiert wurde, der Begriff Werkzeugspezifikation verwendet.

**Unbekannter Typ** Typen werden unbekannt genannt, wenn sie nicht in der Werkzeugspezifikation enthalten sind. Das bedeutet, dass der Typ der Implementierung der Anbindung statisch nicht bekannt ist, weil er nur in der Dateispezifikation enthalten ist. Im Rahmen dieser Arbeit können unbekannte Typen nur dadurch entstehen, dass die Werkzeugspezifikation eines vorgelagerten Werkzeugs eine Typdefinition enthält, welche einem nachgelagerten Werkzeug fehlt. Aus Sicht des Dateitypsystems verhalten sich diese Typen also wie alle anderen in einer IR spezifizierten Typen.

**Bedarfsorientiertes Lesen** Hinter diesem Begriff verbirgt sich das Lesen von Felddaten bei der ersten Benutzung. Da in dieser Arbeit die Verwendung von Feldern, wie etwa im Rahmen der Durchführung eines Passes in einer Attributgrammatik, zugrunde gelegt wird, werden immer alle Daten eines Feldes gelesen. D.h. der Zugriff `x.f` für ein `x` eines Typs *T* bedeutet, dass für alle Instanzen von *T* die Werte von `f` aus der Datei gelesen werden. Dieses Vorgehen entspricht einer Übertragung der faulen Auswertung von Ausdrücken in funktionalen Programmiersprachen auf die Verwendung von Attributen in Zwischendarstellungen. Es wird nicht der Begriff faules Lesen verwendet, um sich von einer faulen Auswertung einzelner Datenpunkte abzugrenzen.

**ID** und **Offset** Der Begriff ID wird für Namen in `.sf`-Dateien verwendet. Es handelt sich hierbei um natürliche Zahlen, die so vergeben werden, dass sie möglichst klein sind. IDs eignen sich aufgrund ihrer Konstruktion oft als Index in eine Array-artige Verwaltungsstruktur. Bei der Verwendung als

Index ist teilweise eine kleine konstante Verschiebung wie etwa 1, 31 oder 32 erforderlich. Offsets bezeichnen Verschiebungen relativ zu einem Array, wie etwa dem Byte-Strom (Array aus Bytes) oder dem Zeiger-Array das zur Verwaltung von gelesenen Objekten verwendet wird. Offsets sind oft relativ zu einer Position im entsprechenden Array ausgedrückt, um ihre Werte kleiner zu halten.

Kapitel 3

# Implikationen der Anforderungen

Dieses Kapitel beschäftigt sich mit den Konsequenzen der gestellten Anforderungen an potentielle Zwischendarstellungsformate. Dabei werden einzelne Anforderungen in Eigenschaften des resultierenden Systems umgeformt. Dadurch lässt sich nicht nur eine Lösung des Problems ableiten. Es können auch bereits existierende Verfahren leicht bewertet werden (siehe §4.6).

## 3.1. Leicht erlernbare und modulare Spezifikationssprache

Um eine leicht erlernbare Spezifikationssprache zu erhalten, sollte sie zum einen funktionsarm und zum anderen stark an dem orientiert sein, was der Werkzeugbauer bereits kennt und erwartet. Ferner sollte man den Werkzeugbauer zwingen, seine Definitionen übersichtlich und gut dokumentiert darzustellen. Daher ist es hilfreich, Kommentare semantisch an die betroffene Spezifikationseinheit zu koppeln und sie in den generierten Code zu exportieren.

Die Forderung nach Modularität hat offensichtliche Konsequenzen für die Spezifikationssprache und wäre durch einen Vorverarbeitungsschritt[1] für bestehende Spezifikationssprachen nachrüstbar. Die Spezifikationssprache so zu entwerfen, dass die Deklarations- und Importreihenfolge unerheblich ist, erhöht die Erlernbarkeit und Nutzbarkeit, da man diese Eigenschaft von anderen Spezifikationssprachen, insbesondere von Java, bereits gewohnt ist.

## 3.2. Grundtypen inklusive Referenzen und Container

Bei den rudimentären Typen wie Integer, Boolean, Float und String muss man die Kodierung und den Wertebereich so wählen, dass einerseits ausreichend viele Typen vorhanden sind, um IR-Spezifikationen zufriedenstellend darzustellen, es andererseits aber nicht zu viele werden, um den Implementierungsaufwand im Griff zu behalten. Dabei sollte man sich auf Standardkodierungen, wie z.B. Zweierkomplement und Unicode, beschränken.

Referenzen erfordern, dass Objekte irgendwie identifizierbar sind. In Kombination mit der bereits bestehenden Forderung nach einer kompakten Repräsentation ergibt sich eine, möglichst implizite, dichte Nummerierung von Objekten. Diese Nummern können dann in Referenzen übersetzt werden. Mithilfe des serialisierten Typsystems können Referenzen typsicher gemacht werden. Ist die Implementierungssprache eines Werkzeugs nicht statisch typisiert, so sind beim Schreiben automatische Typprüfungen durch die Anbindung erforderlich.

Container stellen per se kein Problem dar, zumal ihre Serialisierung immer durch ein Array erfolgen kann und damit sehr effizient ist. Es ist hier wichtig lediglich zu spezifizieren, was ein Container mit Elementen macht und nicht, wie er algorithmisch realisiert ist. Das bringt zwar Unwägbarkeiten in Bezug auf die tatsächliche Laufzeitkomplexität mit sich, führt aber letztlich zu benutzbaren Implementierungen in mehr als einer Programmiersprache.

[1] Beispielsweise an C-`include`-Makros angelehnt.

## 3.3. Objektorientierung

Zunächst erfordert die Unterstützung von Objektorientierung nur eine objektorientierte Spezifikationssprache und ein, falls möglich, objektorientiertes API. Dabei wäre es nicht einmal erforderlich, die spezifizierten Typen zu generieren. Für das Datenformat gelten jedoch aufgrund der Sprachunabhängigkeit die bereits in Abschnitt 2.3 dargestellten Einschränkungen. Ferner ergibt sich aus den beiden Forderungen, Typfehler erkennen und Objekte mit polymorphen Zeigern serialisieren zu können, die Möglichkeit, die Objektorientierung des Datenformats zu nutzen, um auftretende Fehler dem Werkzeugbauer[1] gegenüber in einer Form zu präsentieren, die zu seiner IR-Spezifikation passt und damit unmittelbar verständlich ist.

## 3.4. Referenzen auf statisch unbekannte Typen

In Kombination mit einem serialisierten Typsystem und den Referenzen ist hier lediglich an jede einzelne Referenz ihr Typ anzufügen. Referenzen auf Objekte statisch bekannter Typen benötigen hingegen, zumindest serialisiert, keine individuellen Typinformationen, da eine dynamische Information, die schon aus statischem Wissen folgt, keinen Mehrwert darstellt und somit nur unnötiger Ballast ist. Dadurch ergeben sich im Binärformat zwei strukturell unterschiedliche Zeigertypen.

## 3.5. Implementierung versteckt sich vollständig hinter generiertem spezifikationsspezifischem API

Aus dieser Forderung ergibt sich direkt, dass es einen Quellcode-Generator gibt. Dieser liest die IR-Spezifikation und generiert für die gewünschte Zielsprache eine Implementierung zur Manipulation entsprechender Daten. Dabei ist es erforderlich, dass Kommentare aus der IR-Spezifikation in den generierten Code exportiert werden, da diese das Wissen über die

[1] Wird die Fehlermeldung nicht vom Anwendungscode abgefangen, so wird diese an den Werkzeugnutzer weitergegeben. Daher sollte sie auch für diesen verständlich sein.

IR-Spezifikation zum Zeitpunkt der Generierung für den Programmierer dokumentieren.

## 3.6. Plattform- und Sprachunabhängigkeit

Im Bereich der Plattformunabhängigkeit muss über gemeinsame Eigenschaften und die Eigenheiten der drei großen Betriebssysteme, sowie von Hardware-Architekturen, nachgedacht werden. Bei Hardware-Architekturen ergibt sich als einzige Einschränkung eine Spezifikation der Endianness, auf die in §5.2.6.2 noch einmal eingegangen wird.

Um eine Unabhängigkeit von Programmiersprachen zu erreichen, muss man beim Entwurf eines Serialisierungssystems umsichtig sein. Insbesondere, wenn man sich beim Entwurf nur auf C++ und Java beschränken würde, wodurch man allerdings schon einen erheblichen Teil der Werkzeugbauer abdeckt, kann man leicht Funktionen spezifizieren, die sich nicht mit vertretbarem Aufwand und in für Werkzeugbauer verständlicher Form umsetzen lassen. Hier wäre insbesondere z.B. forall-Polymorphie[1] in Form von Generics oder Templates zu nennen. Wenn man sich hingegen mit C beschäftigt, wird man sehen, dass der Integrationsaufwand für zu polymorphe Spezifikationssprachen unverhältnismäßig hoch ist. Betrachtet man weitere Programmiersprachen, ergeben sich weitere Einschränkungen. Zum einen erkennt man, als Konsequenz generierter APIs, die Notwendigkeit von Bereinigung für Schlüsselwörter. Wenn man sich verbreitete Programmiersprachen wie Ada, C#, Haskell, Lua, OCaml, Perl, Python und Scala ansieht, sieht man, dass die Liste der Standardtypen und Schlüsselwörter zu lang wäre, als dass man sie einfach verbieten könnte. Konflikte mit Schlüsselwörtern und Standardtypnamen, wie etwa `void` oder `Unit`, werden sich folglich nicht vermeiden lassen und müssen bei der API-Generierung behandelt werden.

Würde man die Schnittmenge der Typsysteme bilden, so blieben nur unspezifizierte Zahlen und Zeichenketten, was kaum einen Mehrwert hätte. Daher ist eine individuelle Anpassungen bei jeder Anbindung unerlässlich,

[1] Also Polymorphie, die durch allquantifizierte Typparameter entsteht.

unabhängig davon, wie das Typsystem der Spezifikationssprache aussieht.

Um Ada anbinden zu können, sollten case-insensitive Bezeichner in die Spezifikationssprache übernommen werden. Dadurch vereinfacht sich die Behandlung von Namenskonflikten bei der Code-Generierung. Außerdem wird die Integration von Casing-Vorlieben oder -vorgaben einer Programmiersprache in den generierten Code erleichtert, was zu ansprechenderen APIs führt.

Ebenso wäre es kontraproduktiv, die konkrete Implementierung von Datenstrukturen oder das Layout von Objekten im Hauptspeicher zu spezifizieren. Auch eine Spezifikation des Objektmanagements im Hauptspeicher würde keinen Mehrwert darstellen, wenn man dadurch den erforderlichen Freiraum bei der Implementierung einer Anbindung an eine Programmiersprache verliert.

Außerdem wäre es ein fataler Fehler, wenn der Werkzeugbauer die Befugnis hätte, selbst in den Serialisierungsmechanismus im generierten Code einzugreifen. Hierdurch ergäbe sich ein Austauschformat, das im Zweifelsfall nur noch in der Implementierungssprache des Werkzeugs benutzbar wäre. Das Argument, der Eingriff ließe sich auf andere Programmiersprachen übertragen, ist rein theoretischer Natur, da der damit verbundene Arbeitsaufwand in der Praxis insbesondere dann nicht zu rechtfertigen ist, wenn der Werkzeugbauer nur eine Bibliothek testen möchte, die in einer anderen Programmiersprache implementiert ist.

Die wichtigste Konsequenz aber ist, dass man sich im Funktionsumfang sehr zurückhalten muss, da man jede spezifizierte Fähigkeit in jeder unterstützten Programmiersprache implementieren muss. Außerdem darf man sich bei der Spezifikation des Formats nicht von einer vermeintlich einfachen Implementierbarkeit in einer Programmiersprache verleiten lassen, sondern muss möglichst die ungeeignetste Programmiersprache im Blick haben. Daraus ergibt sich auch, dass alle Freiheitsgrade dahingehend genutzt werden sollten, die tatsächliche Implementierung möglichst einfach zu gestalten.[1]

[1] Hierdurch haben sich Formatänderungen zwischen der ursprünglichen SKilL-Sprachspezifikation und der heutigen Version ergeben.

Am Ende ist es zwingend erforderlich, eine möglichst formale[1] Spezifikation der Konvertierung zwischen Objekten und Bytestrom zu haben, weil sonst schwer zu entscheiden ist, welche Architektur und welche Anbindung nun die richtige Abbildung darstellt.[2]

## 3.7. Deserialisierung bei erster Benutzung von Daten

In §1.1.4 wurde entschieden, die tatsächliche Repräsentation vor dem Werkzeugbauer zu verstecken. Außerdem wird in §3.14 entschieden, ein Laufzeittypsystem zu verwenden, welches vom Werkzeugbauer verwendet werden kann, um auf die Typ- bzw. Feldrepräsentation zur Laufzeit zuzugreifen. Diesen Zugriffsfunktionen kann im API eine Prüfung hinzugefügt werden, die bei jeder Benutzung prüft, ob die Daten schon gelesen wurden, um diese bei Bedarf nachzulesen. Ferner ergibt sich für das Binärformat, dass es eine günstige Methode geben muss, über weite Teile ungenutzter Daten zu springen. Dadurch ergibt sich, dass Daten ihre Längen vorangestellt sein müssen.[3]

Würde man dem Werkzeugbauer gestatten, Daten parallel mit mehreren Werkzeugen zu modifizieren, so wäre es zwingend erforderlich, alle Modifikationen zu synchronisieren und die logische Konsistenz zu gewährleisten. In Verbindung mit der Forderung nach einfacher Benutzbarkeit und Implementierbarkeit ergibt sich die Forderung, dass sich Austauschdateien nicht verändern, während man auf ihnen arbeitet.

[1] Teile, in denen kein geeigneter Formalismus zur Verfügung steht, müssen natürlichsprachlich spezifiziert werden.

[2] Für den in dieser Arbeit vorgestellten Ansatz findet sich die Formalisierung in [Fel17] §B.

[3] Es ist dabei unerheblich, ob die Länge direkt etwa als Zahl der Bytes oder indirekt über die Zahl der Elemente und deren Typ erfolgt. In letzterem Fall ist es erforderlich, dass aus dem Typ die Breite der Repräsentation folgt, da die Zahl der zu überspringenden Bytes sonst nicht berechnet werden kann.

## 3.8. Anhängen neuer Daten und Instanzen an existierende Austauschdateien ohne komplettes Neuschreiben

Aus dieser Forderung ergibt sich eine sich wiederholende Blockstruktur.[1] Jede Anhängeoperation erzeugt dabei mindestens einen neuen Block. Die Interpretation der Daten muss dabei so gestaltet werden, dass die Aufteilung auf mehrere Blöcke kein Problem darstellt. Außerdem muss durch das Serialisierungssystem definiert werden, was in weiteren Blöcken angehängt werden kann und was nicht. Eine dedizierte Produktion[2] am Ende der Austauschdatei würde das Anhängen unmöglich machen. Der terminale Zustand eines Parsers kann also nur darin bestehen, dass er gleichzeitig am Ende eines Blocks und am Ende des Bytestroms ankommt. Formate, die mit einer schließenden Klammer oder einem Tag enden müssen und dabei nur einen Wurzelknoten haben, können diese Forderung nicht umsetzen.

In Kombination mit der Änderungstoleranz ergibt sich aber auch, dass es möglich sein muss, in angehängten Blöcken neue Typen zu definieren, oder existierende Typen zu erweitern. Ebenso muss es möglich sein, Typen ohne Modifikation um weitere Instanzen zu erweitern.

Ob es möglich sein sollte, Daten oder Typen aus einer Austauschdatei zu löschen, ohne dafür die Austauschdatei neu zu schreiben, ist offen. Diese Anforderung erfordert einen Mechanismus, der einzelne Objekte als gelöscht markiert. Hierbei stellt sich die Frage, ob die resultierende Austauschdatei nach wie vor kompakte Indexbereiche für die Identifikation von Objekten verwendet, d.h. manche Daten von gelöschten Objekten demnach übersprungen werden müssen, oder ob Indices neu vergeben werden müssen. Eine Neuvergabe der Indices würde jedoch bedeuten, dass über den Index serialisierte Referenzen beim Lesen uminterpretiert werden müssen. Beide Alternativen verkomplizieren den Lesevorgang durch notwendige Indirektionen, was die Leserate verringert und die Implementierbarkeit reduziert. Indexbereiche in

[1] Als Grammatik formuliert bedeutet das, dass die Produktion des Startsymbols mit einem $Block^+$ oder $Block^*$ enden muss. Aus *Block* muss sich alles ableiten lassen, was an eine Datei angehängt werden kann. Diese Eigenschaft wird beispielsweise von Java erfüllt (siehe [Ora10] §6.4.1), nicht aber von XML (siehe [BPS+06] §2.1) und JSON ([ECM13] §4 und §5).

[2] Wie etwa das Schließen aller offenen Klammern oder ein Ende-des-Datenstroms-Token.

SKilL werden in §5.2.1 erklärt; ihre Vorteile werden insbesondere in §6.3.3 deutlich.

Ferner muss es eine günstige Methode geben zu entscheiden, welche Objekte angehängt werden und welche bereits in der Austauschdatei enthalten sind. Falls das Lesen einer Datei nicht erheblich günstiger ist, als das Schreiben bzw. Anhängen, muss dies in Form einer Verwaltungsstruktur im Hauptspeicher geschehen. Ob diese Verwaltungsstruktur als vollständige Objektverwaltung oder als Markierung gelesener Objekte erfolgt, ist hier offen. Die Objektverwaltung hat gegenüber der reinen Markierung den Vorteil, dass sie genutzt werden kann, um das generierte API mit Iteratoren über verwaltete Objekte zu erweitern, was die Implementierung mancher Algorithmen deutlich[1] vereinfacht (siehe §6.3.3).

## 3.9. Abwärts- und Aufwärtskompatibilität

Ein Werkzeug kann, selbst bei gemeinsam genutzter IR-Spezifikation für alle Werkzeuge einer Werkzeugkette, keine Annahmen über den Inhalt einer eingelesenen Austauschdatei machen. Ursächlich hierfür ist, dass sich in einer weiterentwickelnden Werkzeugkette selbstverständlich auch die IR-Spezifikation weiterentwickelt. Wenn man fordert, dass es möglich sein soll, dass ein Werkzeug, ohne es neu zu übersetzen, auf einer veränderten Zwischendarstellung arbeiten kann, solange sich keine für das Werkzeug relevanten Teile verändert haben, dann bedeutet das, dass ein Werkzeug nur die Typkorrektheit des Inhalts einer eingelesenen Austauschdatei gegenüber der IR-Spezifikation zum Übersetzungszeitpunkt annehmen darf.[2]

Dabei sind die Kompatibilitätsrichtungen vollkommen symmetrisch. Auf der einen Seite gibt es das Werkzeug, welches unverändert weiter arbeitet.

[1] Insbesondere flussinsensitive Analysen können hierdurch von einem rekursiven Problem mit expliziten Typprüfungen und dispatchenden Aufrufen auf ein iteratives Problem ohne dispatchende Aufrufe und explizite Typprüfungen umgestellt werden. Eine Typprüfung muss hinter dem API stattfinden, falls die Implementierungssprache dies erfordert.

[2] Man wird im Allgemeinen auch feststellen, dass manche Änderungen eine Anpassung des Werkzeugs erfordern. Daher darf man nicht annehmen, dass die eingelesenen Daten dem genutzten Zwischendarstellungsfragment entsprechen, was jedoch von einer typsicheren Lösung beim Lesen automatisch erkannt wird.

Aus Sicht dieses Werkzeugs handelt es sich um Aufwärtskompatibilität, da es in der Lage ist, ein neues Format zu produzieren. Auf der anderen Seite steht das geänderte oder neu hinzugefügte Werkzeug. Aus Sicht dieses Werkzeugs handelt es sich um Abwärtskompatibilität, da es ein altes Format als Eingabe akzeptiert.

Um dies zu erreichen, ist es zwingend erforderlich, dass die Objektdarstellung von der konkreten Ausprägung der IR-Spezifikation unabhängig ist. Dies kann erreicht werden, indem die IR-Spezifikation in die Austauschdatei geschrieben wird. Diese Strategie ist nicht unbedingt erforderlich, stellt aber eine mögliche Lösung dar. Man könnte stattdessen auch mit Prüfziffern arbeiten, wie es z.B. in Java der Fall ist. Dagegen ist der Abgleich des spezifizierten Typsystems mit einem serialisierten Typsystem eine präzisere Methode.

Eine weitere unabdingbare Konsequenz ist, dass es klare und einfache Regeln für die Kompatibilität von Änderungen geben muss. Hier sollte man sich natürlich möglichst auf etwas festlegen, das unabhängig z.B. von der Reihenfolge der Typen in der Spezifikationsdatei ist. Zwingend erforderlich ist letzteres im Allgemeinen jedoch nicht.

## 3.10. Geringe Dateigröße

Um eine möglichst geringe Dateigröße zu erhalten, muss das Format so kompakt wie möglich gewählt werden, und die Daten sollten vorzugsweise ohne Redundanz abgelegt sein. Hat man die Wahl zwischen einem Text und einem Binärformat, sollte daher das Binärformat gewählt werden. Bei der Wahl von Kompressionsverfahren muss die Laufzeit der Kompression bzw. Dekompression beachtet werden.

## 3.11. Hohe Leserate

Um Leseraten überhaupt vergleichen zu können, ist diese zunächst auf Objekte pro Sekunde festzulegen. Man ist leicht geneigt, Leseraten in Byte pro Sekunde darzustellen, da dies vermeintlich vergleichbar und leicht messbar

ist. Hier muss man jedoch bedenken, dass der Werkzeugnutzer am Ende einen gegebenen Graphen bearbeitet und nicht eine Austauschdatei gegebener Größe. Generell gilt es natürlich, die Gesamtlaufzeit zu minimieren.

Dadurch ergibt sich eine teilweise widersprechende Zielsetzung zur geringen Dateigröße. So haben Experimente zu Beginn des Entwurfs von SKilL mit diversen Kompressionsformaten, u.a. `zip`[Inf08], `rar`[Ros15] und `7zip`[Pav10], ergeben, dass sich die Laufzeit beim Lesen und Schreiben zu sehr erhöhen würde.[1] Ein vergleichbares Experiment findet sich in §7.8.2.

Um dieses Problem zu lösen, wurde auf sehr effiziente Kompression mittels VBR-Kodierung[2] zurückgegriffen. Bei dieser Art der Kompression ist rein rechnerisch offensichtlich, dass sie für diesen Einsatzzweck eine deutlich bessere und effizientere Kompression bietet ohne fragwürdige numerische Grenzen einzuführen. Dies wäre z.B. der Fall, wenn man statt 64bit breiten Referenzen auf 32bit zurückgegriffen hätte – man erinnere sich an das Jahr 2000 Problem [Ber98]. Ferner kann man ein Binärformat als sehr günstige Kompression eines Textformats auffassen.

An dieser Stelle sollte noch erwähnt werden, dass das generierte sprachspezifische API den enormen Vorteil bietet, dass die konkrete Repräsentation der Datenstrukturen zwar vor dem Werkzeugbauer verborgen ist, nicht jedoch vor dem Compiler der das Werkzeug übersetzt. Dadurch kann der Compiler weitreichende Optimierungen vornehmen, ohne Probleme bei Änderungen zu verursachen. Obwohl die Optimierungen nach einer Änderung der IR-Spezifikation gegebenenfalls divergent ausfallen können, muss der Werkzeugbauer hierbei jedoch nicht manuell eingreifen. Diese Optimierungen haben einen erheblichen positiven Einfluss auf die Serialisierungsperformanz. Bei Ansätzen, die die Umkodierung in den Graph dem Programmierer selbst überlassen, ist davon auszugehen, dass insbesondere schwache Programmierer von einem an sich performanten Ansatz nicht profitieren können.

[1] Ist eine geringe Dateigröße, wie etwa bei der Archivierung, entscheidend, so können die Austauschdateien auch manuell komprimiert werden.

[2] Variable Byte Rate – es werden je nach Wert unterschiedlich viele Bytes verwendet, so dass kleine Werte durch weniger Bytes repräsentiert werden. Siehe [Fel17] Anhang C.

## 3.12. Serialisiertes Typsystem

Die Serialisierung der IR-Spezifikation in der Austauschdatei ist, wenn man das Typsystem und eine im mathematischen Sinne invertierbare Serialisierungsfunktion bereits spezifiziert hat, unproblematisch. Die Designentscheidungen an dieser Stelle ergeben sich vor allem in Kombination mit anderen Punkten, wie etwa geringer Dateigröße und hoher Leserate. Hierdurch muss man eine kompakte Darstellung finden anhand derer möglichst einfach entschieden werden kann, wie Typen aussehen und ob sie mit der Werkzeugspezifikation kompatibel sind.

Es wäre zwar prinzipiell denkbar, auf eine separate Spezifikationsdatei zu verweisen. Dadurch kann aber die Dateispezifikation unabhängig von den eigentlichen Daten verändert werden, wodurch die Beschreibung der Daten fehlerhaft werden kann. Folglich sollte die Dateispezifikation neben den Daten in derselben Austauschdatei abgelegt werden. Dabei sollte die Dateispezifikation vor den Nutzdaten angeordnet sein, um diese linear lesend effizient prüfen und interpretieren zu können.

## 3.13. Erkennung von Typfehlern

Um Typfehler erkennen zu können, muss man zunächst spezifizieren was Typfehler sind. An dieser Stelle wird man sich als Werkzeugbauer eventuell mit dem Problem konfrontiert sehen, dass sich die eigene Vorstellung von Typfehlern und die des Entwicklers einer gegebenen Serialisierungslösung unterscheiden.

Aus der Kompatibilitätsforderung ergibt sich z.B. dass es kein Typfehler sein kann, wenn ein Typ um Felder ergänzt wurde oder Felder fehlen.[1]

[1] Würde man das Fehlen eines Feldes als Typfehler verstehen, so könnten neuere Werkzeuge alte Daten nicht um neue Felder ergänzen. Es wäre ebenso wenig möglich z.B. Felder mit Verweisen auf Analyseergebnisse in die bestehende Zwischendarstellung einzubauen. Es ist natürlich denkbar, Felder in der Werkzeugspezifikation so zu markieren, dass deren Fehlen beim Einlesen einer Austauschdatei einen Fehler verursacht, um das Vorhandensein der Eingabedaten zu gewährleisten.

Daraus ergibt sich auch, dass es definierte Standardwerte[1] geben muss, da sonst auf ein derartiges Verhalten kaum reagiert werden kann. Es wäre zwar prinzipiell denkbar, vom Werkzeugbauer zu verlangen, diese per Callback zur Laufzeit zu erfragen, allerdings wäre dieses Vorgehen in der Praxis kaum praktikabel.

Um offensichtliche Typfehler handelt es sich dagegen, wenn man z.B. eine Zahl erwartet und einen Container vorfindet. Diese sind nach der Deserialisierung der Typen der Felder leicht zu diagnostizieren. Weniger problematisch ist es, wenn sich beispielsweise der Wertebereich von Zahlen verändert, weil unterschiedlich breite Integertypen verwendet werden. Hier kann man zwischen Invarianz oder einer Laufzeittypprüfung wählen.[2]

## 3.14. Partielle Sicht einzelner Werkzeuge auf Gesamtspezifikation

Aus der Kompatibilität ergibt sich direkt, dass es ausreichen muss, nur einen Teil der Spezifikation zu kennen. Dies kann[3] mit einem Laufzeittypsystem erreicht werden, welches von einer Anbindung beim Lesen erzeugt wird. Dieses Laufzeittypsystem entsteht durch Vereinigung der Typdefinitionen aus Datei- und Werkzeugspezifikation. Alternativ kann man auch unbekannte Daten auf den bekannten Teil der Spezifikation abbilden und nicht abbildbare Daten ignorieren.[4] Dies ist jedoch in Hinblick auf Benutzbarkeit und Änderungstoleranz problematisch, da hierdurch alle unbekannten Daten aus der Zwischendarstellung gelöscht werden und folglich für nachgelagerte Werkzeuge nicht mehr zur Verfügung stehen.

Da unbekannte Typen über Reflection dargestellt werden können, kann man nun auch zulassen, dass es keine Gesamtspezifikation mehr geben muss.

[1] Es empfiehlt sich die von C++ (siehe [ISO12] §8.5) und Java (siehe [GJS+14] §4.12.5) verwendeten Null-Äquivalente zu verwenden. Bei der Wahl der Standardwerte in SKilL ist zu beachten, dass Container nicht wie Referenzen auf Container behandelt werden.

[2] SKilL ist invariant, sodass die Frage, welcher Typ beim Schreiben verwendet wird, einfach beantwortet werden kann.

[3] Das aktuelle SKilL verfolgt diesen Weg.

[4] Ur-SKilL verfolgte diesen Weg.

In so einem Fall ergibt jeder Ausführungspfad durch eine Werkzeugkette eine implizite Gesamtspezifikation, die sich am Ende des Pfades dem serialisierten Typsystem entnehmen ließe. Außerdem kann man mit Reflection spezifikationsunabhängige Konverter mit konstantem Programmieraufwand implementieren. Das ist insbesondere für Vergleiche und Forschungskooperationen von sehr hoher Bedeutung.

## 3.15. Zusammenfassung

Zur Übersicht werden alle Eigenschaften noch einmal kurz zusammengefasst. Für die Architektur ergibt sich:

1. Generierter und kommentierter Code (§3.5)
2. Prüfung der Einhaltung des Typsystems durch das API (§3.13, §3.2)
3. Laufzeitdarstellung des Typsystems (§3.9, §3.14)
4. Zielsprachspezifische Bereinigung gegen Namenskonflikte (§3.6)
5. Unterspezifizierte in-Memory-Repräsentation (§3.6)
6. Austauschdateien sind implizit exklusiv von einem Werkzeug genutzt, sodass keine Locks und keine Konsistenzprüfung für bedarfsorientiertes Lesen/Anhängen erforderlich sind (§3.7)

Für die Spezifikationssprache ergibt sich:

1. Syntax und Semantik möglichst an Java und C++ angelehnt (§3.1)
2. Ausreichend viele Grundtypen (§3.6, §3.3)
3. Möglichst geringer Funktionsumfang (§3.6)
4. Eingebaute Objektorientierung (§3.3)
5. Einfaches Typsystem ohne Templates, Generics oder Typvariablen (§3.6)

Für das Serialisierungsformat ergibt sich:

1. Binäres Format (§3.10)
2. Nur sehr schnelle Dateikompression (§3.10, §3.11, §7.8.2)
3. Spezifizierte Endianness und Zeichenkodierung (§3.6)
4. Spezifikationsunabhängige Objektdarstellung (§3.9)
5. Implizit nummerierte Objekte (§3.2)
6. Referenzen mit und ohne individuelle Typinformationen (§3.4)
7. Überspringbare Datenblöcke (§3.7)
8. Serialisierte Typinformationen inklusive Anhängen neuer Typen (§3.12, §3.8)
9. Eine beliebige Anzahl gleichwertiger Blöcke als Toplevelstruktur (§3.8)

An dieser Stelle sei bemerkt, dass SKilL einige Eigenschaften hat, die sich nicht aus diesen Forderungen ergeben. Die hier bleibenden Freiheitsgrade sind immer noch ausreichend groß, um eine Vielzahl echt unterschiedlicher Serialisierungssysteme konstruieren zu können. So ist die Implementierung von Interfaces[1] reine Kür und der Katalog an Hints und Restrictions[2] lässt sich bestenfalls historisch begründen.

[1] siehe §5.4.1
[2] siehe §5.1.6 bzw. für den Katalog [Fel17] §5

Kapitel 4

# Stand der Technik

Dieser Abschnitt befasst sich mit den bereits existierenden Ansätzen, die ebenfalls das Problem der Zwischendarstellungen adressieren. Dabei werden die im letzten Kapitel erarbeiteten Eigenschaften benutzt, um eine schnelle Klassifikation zu erreichen.

## 4.1. Textbasierte Datenformate

Datenformate wie Extensible Markup Language (XML)[BPS+06], JavaScript Object Notation (JSON)[ECM13] oder YAML[BEdN09] beschäftigen sich zwar mit der Darstellung einfacher Daten, allerdings werden in dieser Arbeit gestellte Anforderungen an Architektur und Typsystem nur unzureichend erfüllt.

Da es eine Reihe von Ansätzen gibt, die auf XML aufbauen, und die allesamt durch eine wenig kompakte serialisierte Form auffallen, sei hier bemerkt, dass in der Sprachspezifikation [BPS+06] in §1.1 die Designziele von XML dargelegt werden. Für das beobachtete Verhalten verantwortlich ist dabei Punkt 10:

*Terseness in XML markup is of minimal importance.*

(Frei übersetzt: *Die Kompaktheit von XML-Daten ist nebensächlich*.)

Es gibt verschiedene Kompressionsmethoden welche die Dateigröße reduzieren. Ein guter Vergleich solcher Werkzeuge findet sich in [Sak09]. Verwendet man, wie es im Vergleich gemacht wird, ein externes Werkzeug für die De-/Kompression, so wird jedem Werkzeugaufruf ein Weiterer vor- und nachgelagert, was Ausführungszeit und Benutzbarkeit beeinträchtigt. Eine ähnliche Untersuchung findet sich für SKilL und zwei weitere Formate in §7.8.2. Diese legt nahe, dass sich zumindest eine allgemeine Dateikompression für die direkte Kommunikation innerhalb der Werkzeugkette nicht rechnet.[1]

### 4.1.1. XML Schema

Für XML gibt es, neben anderen Spezifikationssprachen, die Spezifikationssprache XML Schema Definition Language (XSD)[GST+08]. Verwendet man zudem Quellcode-Generatoren wie JAXB[Sun06] oder xmlbeanscxx[2] [Apa15], so erhält man zusätzlich auch ein generiertes API für Java und C++. Nichtsdestotrotz führt der XML-Unterbau dazu, dass die in §3.15 geforderten Eigenschaften an das Austauschformat 1, 7 und 9 nicht erfüllt werden. Betrachtet man die Eigenschaften von XSD, so erkennt man schnell, dass in dieser Arbeit andere Ziele verfolgt werden. Die Darstellung ist nicht Java-ähnlich und es gibt sehr viele und z.T. sehr elaborate Typen und Wege, um Typen zu beeinflussen, was dem Punkt *geringer Funktionsumfang* widerspricht und dadurch die Kosten einer Implementierung erhöht.

## 4.2. Sprachspezifische Serialisierung

Bisherige sprachspezifische Lösungen erfüllen unsere Anforderungen an Sprachunabhängigkeit zunächst einmal nicht. Jedoch lohnt es sich trotzdem, diese zu untersuchen, um von ihnen zu lernen.

[1] Im Sinne von Forderung §1.1.9 ist die erhöhte Laufzeit unverhältnismäßig.

[2] Das Projekt wird nicht mehr gepflegt.

### 4.2.1. Java

Java verwendet als Spezifikation der Daten die ohnehin vorhandenen Typdefinitionen in Java. Dabei gibt es in der Programmiersprache selbst das Schlüsselwort `transient` (siehe [GJS+14] §8.3.1.3), welches Felddefinitionen von der Serialisierung ausschließt.[1] Bei der tatsächlichen Implementierung handelt es sich um regulären Java-Code, der mit der Standardbibliothek ausgeliefert wird. Die Verwendung von *serialVersionUID*s zur Identifikation von Typen anstelle eines vollständig serialisierten Typsystems und die Möglichkeit, in die Serialisierung von Objekten durch das Überschreiben von Methoden einzugreifen, sind maßgeblich für praktische Probleme im Bereich der Änderungstoleranz verantwortlich (siehe [Blo08] Kapitel 11), da im Allgemeinen nicht entschieden werden kann, wie das Typsystem der Austauschdatei aus Sicht des schreibenden Werkzeugs ausgesehen hat, ohne zu wissen, wie das schreibende Werkzeug zum Zeitpunkt des Schreibens der Austauschdatei implementiert war. Ferner ist das Format im Allgemeinen[2] nicht bedarfsorientiert auswertbar. Dagegen ist das Anhängen neuer Daten mit geringem Aufwand implementierbar.

### 4.2.2. Ada

Es gibt einen sehr rudimentären Vergleich zwischen Ada und SKilL in [Prz14] §7. Der Vergleich zwischen Ada-Serialisierung und SKilL wurde nicht weiter verfolgt, da die Experimente von Herrn Przytarski den Schluss zulassen, dass es, bei insgesamt vergleichbarer Performance, bei einer Bewertung darauf ankommt, dass SKilL eine sprachunabhängige Lösung ist. Ferner wird die in §2.2 beschriebene Änderungstoleranz nicht automatisch[3] umgesetzt und das Binärformat ist nicht von der konkreten Implementierung des schreibenden Werkzeugs abhängig. Aus Sicht der Ziele dieser Arbeit ist die Ada-Sprachspezifikation (siehe [TDB+13] §13.13.2) zu freizügig.

[1] Ein vergleichbares Konzept wird in §5.4.4 beschrieben.

[2] Wenn man davon ausgeht, dass der Serialisierungsmechanismus nicht überschrieben wurde.

[3] Man kann in die Serialisierung nahezu beliebig eingreifen.

## 4.3. Problemspezifische Lösungen

In diesem Abschnitt werden werkzeugkettenspezifische Lösungen genauer betrachtet.

### 4.3.1. Compiler und Virtuelle Maschinen

Aus dem Bereich Compiler-Zwischendarstellungen sind besonders Java Bytecode[LY99], LLVM Bitcode[LLV15], GNU RTL[SG17] und CIL/CLI[ECM12] hervorzuheben. Diese werden derzeit aktiv gewartet und haben einen signifikanten Marktanteil. Sie sind vollkommen auf ihre Aufgabe zugeschnitten, wodurch sie sich nicht als allgemeine Lösung eignen.[1] Daher haben diese auch eine Sprach-Spezifikation in Form eines langen[2] Textes, der jedes Detail erläutert. In einem lösungsunabhängigen Serialisierungssystem wie SKilL obliegt die Beschreibung der Details den darauf aufgebauten IRs. Dadurch ist es diesen möglich, Details nach ihren Bedürfnissen frei festzulegen.

Insbesondere der Konstantenpool sowie die prinzipiell überspringbaren Informationen und die erweiterbare Codierung der .class-Files wurde in SKilL übernommen. Dabei werden in SKilL jedoch nur Strings in einen Pool aufgenommen. Andere im Bytecode vorhandene Konstanten, wie etwa Verweise auf Methoden, sind nur im problemspezifischen Kontext sinnvoll, könnten aber in SKilL mit Referenzen nachgebaut werden. Java Bytecode und LLVM Bitcode beginnen mit einem magischen Wert, der die Eingabe falscher Datenformate leicht erkennbar macht. Das Konzept magischer Werte wurde abstrahiert und in Form von Konstanten in SKilL übernommen.

[1] Das von der Zwischendarstellung vorgegebene Grundgerüst kann nicht verändert werden. Folglich können diese Lösungen nur dann zum Einsatz kommen, wenn das Grundgerüst verwendet werden kann bzw. ohnehin verwendet werden soll.

[2] Die JVM-Spezifikation ist über 400 Seiten lang, die ursprüngliche SKilL-Spezifikation umfasste nur 43 Seiten.

### 4.3.2. Bauhaus IML

Der vorhandenen Lösung für Bauhaus[1] fehlen die Fähigkeiten, Daten anzuhängen und bedarfsorientiert auszuwerten, sowie ein Umgang mit Veränderungen der Spezifikation. Außerdem führt die Möglichkeit, eigene Serialisierungsprozeduren zu definieren, zu den in Abschnitt 3.6 beschriebenen Problemen. Als Folge dessen wurden Bauhaus Intermediate Language (IML)-Anbindungen an andere Programmiersprachen durch Sprach-Interfaces realisiert, die immer auf Aufrufe der Prozeduren der IML-Ada-Anbindung abgebildet werden. Dadurch ist zwar die Performanz der Serialisierungsfunktionen nicht wesentlich beeinträchtigt, dafür aber jeder einzelne Datenzugriff. Zu IML als Serialisierungsformat ist keine veröffentlichte Dokumentation bekannt.

## 4.4. Interprozesskommunikation

Im Bereich der Interprozesskommunikation findet man üblicherweise keine Objektorientierung. Insbesondere das Anhängen an Daten oder das bedarfsorientierte Auslesen fehlt hier meist gänzlich, da üblicherweise kleine Datensätze an einen anderen Prozess versandt werden, um Arbeitspakete an diesen zu verteilen. Interessant ist dieser Bereich dennoch, da er sehr kompakte Formate hervorgebracht hat, von denen man lernen kann.

### 4.4.1. Protocol Buffer

Protocol Buffer[Goo13] bieten eine Spezifikationssprache, um ein kompaktes Format zu beschreiben und Anbindungen dafür in mehreren Programmiersprachen zu generieren. Hier sind besonders die variabel langen Zahlentypen hervorzuheben, da diese mit geringem Rechenaufwand eine gute Datenkompression erreichen können. Datenstrukturen bestehen aus Feldern, die über IDs identifiziert werden. Hierdurch ist eine Änderungstoleranz erreichbar,

[1] Es gibt hierzu keine zitierbare Beschreibung.

welche auf struktureller Äquivalenz anstelle von Namensäquivalenz beruht. Objektorientierung fehlt hier allerdings gänzlich.

### 4.4.2. Thrift

Thrift[Apa13] ist ein mit Protocol Buffer vergleichbarer Ansatz, der das Typsystem einer Nachricht in der Nachricht serialisiert. Objektorientierung wird hier explizit abgelehnt.[1]

### 4.4.3. Weitere

Aus dieser Richtung kommen noch weitere Formate wie SBE[Tho15] und CapNProto[Var15], die aus Sicht dieser Arbeit keine relevanten Verbesserungen gegenüber Protocol Buffer oder Thrift enthalten.

## 4.5. Zwischendarstellungsformate für Analysewerkzeuge

Es existieren neben IML und SKilL noch weitere Ansätze, um die Kommunikation in Werkzeugketten zu organisieren.

### 4.5.1. IDL

IDL (Interface Description Language)[Lam87] beschäftigt sich, wie der Name vermuten lässt, hauptsächlich mit der Frage, wie Werkzeuge miteinander interagieren. Die Frage einer effizienten Darstellung der Daten ist hier aber eher nachgelagert. In [NWL81]§4 wird die Datenrepräsentation kurz erläutert. Die gewählte Lösung kann nicht bedarfsorientiert lesend implementiert werden. Vergleicht man die heutzutage verwendeten Datenmengen mit denen von vor drei Jahrzehnten, so ist es wenig verwunderlich, dass man damals noch keine Maßnahmen ergriffen hat, auch auf heute üblichen Datenmengen noch skalierbar zu sein. Die ehemals getroffenen Entscheidungen sind in diesem Kontext durchaus auch heute noch nachvollziehbar, würden

[1] Siehe `https://thrift.apache.org/docs/features` (vom 20.7.2016)

jetzt aber anders ausfallen. So würde man beispielsweise etwa Unicode statt ASCII wählen. Die erste Version von IEEE-754 ist aus dem Jahre 1985, sodass hier nicht auf standardisierte Gleitkommazahlen zurückgegriffen werden konnte. Auch die Entwicklung eines Datenformats, das parallel de- und enkodiert werden kann, wäre zu dieser Zeit kein naheliegendes Ziel gewesen.

### 4.5.2. EMF

Das Eclipse Modeling Framework (EMF) verfolgt einen Ansatz, der vom Standpunkt der Architektur die größten Gemeinsamkeiten mit der in dieser Arbeit vorgestellten Lösung hat. So gibt es z.B. Quellcode-Generatoren, die zu einer gegebenen Spezifikation eine Anbindung generieren. Der Ansatz ist zwar UML-zentrisch, in erster Linie aber unabhängig von konkreten Programmiersprachen. Die verfügbaren Datenstrukturen erscheinen für eine Zwischendarstellung jedoch unzureichend, da keine serialisierbare Repräsentation für Arrays oder Mengen zur Verfügung steht. Eine Wortsuche in dem im FAQ als "the EMF book" bezeichneten Werk [MDG+04] liefert weder für *array* noch für *set* eine Antwort auf dieses Problem. Zudem gibt es keine naheliegende Repräsentation von *Maps*. Diese werden über einen Sonderfall im EMF-Quellcode-Generator realisiert.[1]

EMF setzt zunächst auf ein XML-basiertes Serialisierungsformat, dieses kann jedoch ausgetauscht werden. Ein auf JSON basierendes alternatives Austauschformat ist *emfjson* [Hil16]. Weil die Architektur von EMF den Austausch des Serialisierungsformats zulässt, ist EMF teilweise orthogonal zu SKilL. Es wäre durchaus möglich, eine SKilL-basierte Serialisierung für EMF zu konstruieren.

EMF selbst bietet einen Quellcode-Generator mit Zielsprache Java. Es gibt Bestrebungen, Quellcode-Generatoren für die Zielsprachen C++ und C# zu

[1] Siehe `http://wiki.eclipse.org/EMF/FAQ#How_do_I_create_a_Map_in_EMF.3F` (am 20.7.2016). Die dort dargestellte Lösung erfüllt den Anspruch an einfache Bedienbarkeit in keinster Weise.

implementieren. Auf letzteren wird vom EMF-Projekt selbst[1] verwiesen, eine Implementierung war aber nicht auffindbar.[2] Der EMF-Quellcode-Generator für die Zielsprache C++[3] wird nicht mehr erkennbar gewartet. Die Anleitung bezieht sich auf eine Eclipse Version von 2010 und lässt sich mit Eclipse Neon (von 2016) nicht nutzen, um C++-Code zu generieren, da das entsprechende Menü, entgegen der Beschreibung in der Dokumentation, nicht existiert. Daher kann man schlussendlich sagen, dass es sich bei EMF defakto um eine reine Java-Lösung handelt.

### 4.5.3. GXL

GXL [HSSW00] ist eine auf XML basierende Spezifikation für den Austausch von Graphen. Dadurch erbt es die Eigenschaften der Kombination von XML und XSD. Ein Vergleich mit SKilL findet sich in der Studienarbeit von Herrn Kaistra [Kai15]. Laut Kaistra sind unkomprimierte XML-Austauschdateien etwa einen Faktor 10 größer als äquivalente SKilL-Austauschdateien. Komprimiert man Austauschdateien mit der besten für das Format verfügbaren Kompression, so erhält man vergleichbare Dateigrößen zum Preis gestiegener Laufzeiten. Er konnte ferner beobachten, dass die von ihm verwendete SKilL/Ada-Implementierung mehr Hauptspeicher aber weniger Laufzeit beanspruchte. Die von Kaistra vermessene Anbindung wird in dieser Form jedoch nicht mehr verwendet, da sie durch eine neue Variante mit verbesserter Architektur ersetzt wurde.

### 4.5.4. TGraphen

TGraph [ERW08] ist eine Alternative zu SKilL mit dem Vorteil einer frei verfügbaren Implementierung.[4] Im Kontext dieser Arbeit ist es wichtig, zwischen der theoretischen Betrachtung von Daten als typisierte Graphen und

[1] Siehe http://www.eclipse.org/modeling/emft/?project=emf4net

[2] Am 7.7.2016 und am 24.4.2017 enthielt der Punkt Downloads an Stelle eines Links den Text "coming soon!".

[3] Siehe https://github.com/catedrasaes-umu/emf4cpp

[4] Siehe https://github.com/jgralab/jgralab

der tatsächlichen Implementierung zu unterscheiden. Man könnte natürlich, ähnlich wie es Jonathan Roth in seiner Masterarbeit [Rot15] für SKilL vorschlägt, einen Wrapper um SKilL-Zustände bauen, der das TGraph API exportiert. Dadurch könnte man die TGraph-Theorie auf SKilL anwenden. Dieser Teil ist also orthogonal zum zugrunde liegenden Serialisierungssystem, sodass wir uns im Folgenden nur noch mit diesem befassen.

Ein interessanter Aspekt ist, dass bei TGraphen sowohl Knoten, als auch Kanten explizit existieren und Typen erhalten. In SKilL-Spezifikationen hingegen existieren Kanten implizit als Referenzen. In SKilL-Austauschdateien sind Knoten implizit, um bedarfsorientiertes Lesen umzusetzen und Platz zu sparen. Effiziente SKilL-Implementierungen benötigen in etwa 30% der Ausführungszeit beim Lesen für Objektallokation. Daher ist es schwer vorstellbar, dass man eine vergleichbar effiziente Implementierung erreichen kann, wenn Kanten ebenfalls durch Objekte repräsentiert werden. Wie in §7.8.3 zu sehen ist, stoßen hier schon IML und SKilL an ihre Grenzen, obwohl sie Kanten durch einfache Zeiger repräsentieren. Die in [ERW08] in Fig. 3 dargestellte Graphgröße deckt sich, wenn man die Knotenzahl betrachtet, in etwa mit einer vergleichbaren Darstellung in der Bauhaus-IR. Bei den Kanten ergeben sich in Bauhaus bei manchen Analysen erheblich größere Zahlen,[1] sodass eine entsprechend schlechtere Performance zu erwarten ist. Da der Vorsprung von SKilL gegenüber IML ohnehin eher knapp ist,[2] würde man mit einer entsprechenden Änderung im direkten Vergleich mit IML verlieren und hätte es bedeutend[3] schwerer zu argumentieren, warum man IML ersetzt. Zudem wäre zu erwarten, dass der Mehrbedarf an Hauptspeicher nicht zu vernachlässigen ist.

Ein Nachteil ist, dass das Format kein bedarfsorientiertes Lesen unterstützt. Um sich der Wichtigkeit dieser Eigenschaft zu vergewissern, betrachte man entweder, wie sehr diese Eigenschaft die Datenerhebung für diese Arbeit

[1] Siehe Tabelle 7.9. Hier hat `bash43` nach der CFG Analyse ein Verhältnis von etwa 117 Kanten pro Knoten, bei einer an sich erheblichen Knotenzahl. Wie später noch gezeigt wird, stellt dieses Problem für SKilL nur eine sehr mäßige Herausforderung dar. Wenn Kanten eigene Objekte wären, ist nicht erkennbar, wie hier eine brauchbare Darstellung aussehen könnte.

[2] Siehe §7.8

[3] Es ist schwer gegen schneller **und** besser zu argumentieren.

erleichtert hat (siehe §7.5), oder man betrachte die Ausführungszeiten in Abschnitt 7.8.3.

Alle übrigen Eigenschaften hätten sich vermutlich mit einem für die Schaffung von SKilL vergleichbaren Aufwand realisieren lassen. So wären beispielsweise die Anbindung an Programmiersprachen neben Java oder an eine SKilL-ähnliche Spezifikationssprache keine Herausforderung gewesen. Betrachtungen zu Änderungstoleranz sind nicht öffentlich zugänglich. Da das Austauschformat ein Schema spezifizieren kann, ist ein änderungstolerantes Verhalten prinzipiell umsetzbar.

## 4.6. Zusammenfassung

Als übersichtliche Zusammenfassung soll eine Matrix dienen, welche die 8 wichtigsten Features für die 8 wichtigsten in diesem Kapitel vorgestellten Lösungen darstellt (siehe Tab. 4.1). Das in dieser Arbeit zentrale Feature der Änderungstoleranz wird in die Aspekte Erkennung und Toleranz der Änderung aufgeteilt, weil viele Mechanismen Änderungen erkennen können. Die Features sind nach Abhängigkeiten und Wichtigkeit sortiert, sodass das Fehlen eines Features die Evaluation weiterer Features weiter rechts nicht erfordert. Die Bewertung erfolgt in Form der Symbole ✓/× für eindeutig klassifizierbare Beziehungen. Ein Strich (–) bedeutet, dass die Betrachtung aufgrund eines fehlenden Features nicht sinnvoll ist. Zahlen beziehen sich auf die folgenden Anmerkungen:

1. Stark implementierungsabhängig mit deutlicher Tendenz zu ×.
2. Werkzeugbauer kann Serialisierungsfunktionen überschreiben.
3. Die Lösungen sind offensichtlich plattformunabhängig. Der Sprachabhängigkeit kann man den Implementierungsaufwand entgegenhalten, d.h. Tendenz zu ✓.
4. Die Spezifikationssprache enthält weder Arrays noch Maps.
5. Die Spezifikationssprache enthält weder Listen noch Sets. Ebenso fehlen Zeiger, mit denen allgemeine Graphstrukturen aufgebaut werden

| | GT | OO | Spez. | IR | API | ÄE | ÄT | S & P | Perf. |
|---|---|---|---|---|---|---|---|---|---|
| XML | × | × | × | – | – | – | – | ✓ | 1 |
| JSON | × | × | × | – | – | – | – | ✓ | 1 |
| Java | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × | 2 (×) | ✓ |
| Java Bytecode | × | × | × | – | – | – | – | 3 | ✓ |
| LLVM Bitcode | ✓ | × | × | – | – | – | – | 3 | ✓ |
| IML | 4 (✓) | ✓ | ✓ | ✓ | ✓ | ✓ | × | × | ✓ |
| ProtoBuf | 5 (×) | × | ✓ | ✓ | ✓ | ✓ | 6 (✓) | ✓ | ✓ |
| EMF (XMI) | 7 | ✓ | ✓ | ✓ | 8 (✓) | ✓ | × | 9 | 10 (×) |

Tabelle 4.1.: Feature-Matrix für die wichtigsten verwandten Arbeiten. Features sind: ausreichend **G**rund**T**ypen, **O**bjekt-**O**rientierung, **Spez**ifikationssprache, frei spezifizierbare **IR**, spezifikationsspezifisches **API**, **Ä**nderungs**E**rkennung/-**T**oleranz, **S**prach- **&** **P**lattformunabhängig, **Perf**ormante De-/Serialisierung

können.

6. Versionsabhängig.[1] In der neusten Version ist das Verhalten explizit unspezifiziert. Es ist hierbei zu beachten, dass sich diese Betrachtung auf Felder in Nachrichten beschränkt und diese über in der Spezifikation vergebene IDs identifiziert werden.
7. Die Spezifikationssprache enthält weder Sets noch Arrays (siehe §4.5.2).
8. Der Quellcode-Generator erzeugt bei Listen und Maps über Zahlentypen nicht übersetzbaren Quelltext.
9. Die Lösung ist plattformunabhängig. Auf ein XMI-Austauschformat und ecore-Spezifikationen beschränkt ist nicht einzusehen, was einer Implementierung in anderen Sprachen prinzipiell im Wege steht. Bei dieser Beurteilung muss aber bedacht werden, dass ProtoBuf [Goo13] und SKilL [FP16] tatsächlich Anbindungen an mehrere Programmiersprachen anbieten.
10. Superlineare Laufzeit (siehe Anhang D).

[1] Siehe
https://developers.google.com/protocol-buffers/docs/proto3#unknowns

Kapitel 5

# Design von SKilL

SKilL ist ein Serialisierungskonzept, welches primär für den Einsatz als Zwischendarstellung im Bereich der Programmanalysen entworfen wurde. Eine wichtige Eigenschaft ist dabei die Fähigkeit, sprachunabhängig Instanzen objektorientierter Typen in einer Austauschdatei zu speichern. Die effiziente Verarbeitung der sich hieraus ergebenden Graphen durch ein Werkzeug ist eine besondere Herausforderung, weil die Strukturierung der Graphen projektabhängig ist und daher augenscheinlich keinerlei struktureller Einschränkungen unterliegt.

Um dieser Herausforderung zu begegnen, verwendet SKilL eine Spezifikationssprache, die mithilfe von Quellcode-Generatoren in ein anwendungsspezifisches API übersetzt wird.[1] Hierdurch kann vor dem Werkzeugbauer weitgehend verborgen werden, wie das für SKilL entwickelte, hoch spezialisierte Serialisierungsformat im Detail funktioniert.[2]

[1] Dieses Vorgehen ist unter modernen Serialisierungskonzepten verbreitet – siehe etwa EMF[SBPM09] oder Protocol Buffers[Goo13].

[2] Der Aufbau, sowie De-/Serialisierung von Austauschdateien wird vollständig versteckt.

## 5.1. Die Spezifikationssprache

In diesem Abschnitt wird anhand von Beispielen die Spezifikationssprache erläutert.[1] Es wird eine vereinfachte Sprachdefinition verwendet, die dann in späteren Abschnitten um für diese Arbeit relevante Teile erweitert wird. Auf eine formale und vollständige Spezifikation wird hier aus Gründen der Relevanz für die weitere Arbeit verzichtet. Diese findet sich in [Fel17]. Dort finden sich auch viele Beispiele, die die Verwendung der Spezifikationssprache und ihre Randfälle illustrieren. Da es sich um eine Datenbeschreibungssprache handelt, ist diese ohnehin leicht zu verstehen. Per Konvention wird Spezifikationsdateien die Dateiendung `.skill` gegeben. Diese werden als `.skill`-Spezifikationen bezeichnet. Es handelt sich hierbei um eine SKilL-basierte IR-Spezifikation. Diese ist menschenlesbar und wird vom Quellcode-Generator verwendet, um eine Anbindung zu generieren. Die Anbindung enthält die Werkzeugspezifikation in einer für die Implementierung der Anbindung geeigneten Form.[2] Die Dateispezifikation, also die in der Austauschdatei mitgeführte IR-Spezifikation, ist nicht menschenlesbar, hat aber, auf Typen und Felder beschränkt, bei unterschiedlicher Syntax dieselbe Semantik wie eine entsprechende `.skill`-Spezifikation.[3]

Zu Beginn soll ein einfaches Beispiel die Spezifikationssprache illustrieren. Ein Typ `T` wird mit einem Feld `f` definiert. Dabei habe `f` als Typ das Äquivalent von `int` in Java, also ein 32bit Zweierkomplement-Integer. Die entsprechende Definition wird in Listing 5.1 dargestellt.

Um der Objektorientierung Rechnung zu tragen, wird das obige Beispiel als nächstes um einen Supertyp `B` und einen Subtyp `U` erweitert. Dabei haben alle Instanzen von `U` implizit auch ein Feld `f`. Instanzen von `B` haben jedoch nur dann ein Feld `f`, wenn sie auch Instanz des Typs `T` sind. Das Ergebnis ist in Listing 5.2 dargestellt. An dieser Stelle ist anzumerken,

[1] Weitere Beispiele finden sich in [Fel17], insbesondere §7.7, §8.1.

[2] Die hier vorgestellte Implementierung enthält diese implizit in der Typstruktur des generierten Codes.

[3] Beide Spezifikationen führen noch zusätzliche Informationen mit, die nur für ihre Repräsentation von Bedeutung sind. Für `.skill`-Spezifikationen sind das beispielsweise Kommentare, sowie die Groß-/Kleinschreibung von Bezeichnern. Bei Dateispezifikationen ist das beispielsweise die Zahl der Instanzen in der Datei.

```
T {
  i32 f;
}
```

Listing 5.1: Ein einfaches Beispiel

```
B {}

T extends B {
  i32 f;
}

U extends T {}
```

Listing 5.2: Einfache Veerbung

dass es Vererbung nur zwischen benutzerdefinierten Typen gibt.[1] Ferner sei erwähnt, dass die Deklarationsreihenfolge in der `.skill`-Spezifikation vollkommen ignoriert wird. Auch kann anstelle von `extends` ein `with` oder ein Doppelpunkt verwendet werden. Hier werden verschiedene Notationen verbreiteter Programmiersprachen realisiert.[2]

Wenn es um die Implementierung von SKilL geht, dann spielen sogenannte Basistypen eine zentrale Rolle. Ein Basistyp ist die Wurzel der jeweiligen Typhierarchie. Es handelt sich beim Basistyp um eine totale Funktion auf den vom Werkzeugbauer definierten Typen. Für alle Typdefinitionen des oben verwendeten Beispiels ist `B` der Basistyp.

[1] Hierdurch soll eine einfache Implementierbarkeit in unterschiedlichen Programmiersprachen erreicht werden.

[2] Für die Syntaxanalyse ist das vollkommen unerheblich. Man könnte sogar `extends : with {}` korrekt analysieren, was praktisch aber nicht empfehlenswert ist.

### 5.1.1. Grundlegender Aufbau von `.skill`-Spezifikationen

Die Spezifikationssprache ist äußerlich stark von Java, C und C++ beeinflusst, da es sich hierbei um weit verbreitete Programmiersprachen handelt. Es wird hier vereinfachend angenommen,[1] dass die Regeln für Literale und Identifier diesen Programmiersprachen folgen. Das Grundgerüst der Grammatik ist in Abb. 5.1 dargestellt. Die hier verwendeten Punkte ($\cdots$) stehen für Regeln, die der Übersichtlichkeit halber weggelassen wurden und im Laufe des Kapitels definiert werden.[2] Eine vollständige Grammatik findet sich in [Fel17] §A.

Zunächst wird eine einfache Dateistruktur verwendet, welche aus einer beliebigen Zahl an benannten Typdefinitionen besteht, welche wiederum aus beliebig vielen Felddefinitionen bestehen. Für Felder gibt es neben den selbstdefinierten Typen auch eine Reihe an vordefinierten Typen, wie Arrays, Listen, Sets und Maps.

### 5.1.2. Verfügbare Feldtypen

Eine der zentralen Fragen besteht darin, aus was eigene Typdefinitionen zusammengesetzt sein können. Hierfür wird zunächst eine Auswahl grundlegender Java-Typen abgebildet.[3] Es gibt also Ganzzahlen und Gleitkommazahlen in verschiedenen Längen. Ferner gibt es Strings, Referenzen und Standardcontainer. Außerdem gibt es einen Typ `bool`, welcher Wahrheitswerte repräsentiert.

Die verfügbaren Ganzzahlen haben stets ein Vorzeichen, werden im Zweierkomplement gespeichert und stehen in den Längen 8, 16, 32 und 64 Bit zur Verfügung. Inspiriert von LLVM/IR[LLV15] lauten ihre Namen `i8`, `i16`, usw.. Diese entsprechen in Java den Typen `byte`, `short`, `int` und `long`. Für die Repräsentation kleinen positiven Zahlen, deren maximale Größe unbekannt ist, gibt es einen Variable bitrate (VBR) kodierten 64bit Integer,

[1] Im Detail sind gerade die Regeln für Identifier für die oben genannten Programmiersprachen unterschiedlich. Dies ist jedoch für unsere Betrachtung zunächst unerheblich.

[2] Um eine Grammatik zu erhalten, kann man sich eine Produktion $\cdots ::= \varepsilon$ denken.

[3] Die Auswahl ist eine Reaktion auf die in §3.2 dargestellte Beobachtung.

⟨*file*⟩ ::= ⟨*header*⟩ ⟨*declaration*⟩*

⟨*header*⟩ ::= ···

⟨*declaration*⟩ ::= ⟨*user-type*⟩
| ···

⟨*user-type*⟩ ::= ⟨*description*⟩ ⟨*ID*⟩ (':'|`with`|`extends`) ⟨*ID*⟩)* '{' ⟨*field*⟩* '}'

⟨*field*⟩ ::= ⟨*description*⟩ (⟨*data*⟩| ··· ) ';'

⟨*description*⟩ ::= ···

⟨*data*⟩ ::= ⟨*type*⟩ ⟨*ID*⟩

⟨*type*⟩ ::= `map` ⟨*type-multi*⟩
| `set` ⟨*type-single*⟩
| `list` ⟨*type-single*⟩
| ⟨*array-type*⟩

⟨*type-multi*⟩ ::= '<' ⟨*ground-type*⟩ (',' ⟨*ground-type*⟩)+ '>'

⟨*type-single*⟩ ::= '<' ⟨*ground-type*⟩ '>'

⟨*array-type*⟩ ::= ⟨*ground-type*⟩ ('[' ⟨*INT*⟩? ']')?

⟨*ground-type*⟩ ::= ⟨*ID*⟩

Abbildung 5.1.: Grundgerüst der SKilL-Spezifikationssprache

der für kleine natürliche Zahlen kompakter[1] serialisert wird als für große oder negative Zahlen.[2] Dieser Typ wird daher `v64` genannt. Er sollte sich im API nicht von `i64` unterscheiden.

Es gibt IEEE-754-Gleitkommazahl [IEE08] in den Breiten 32bit und 64bit.

[1] Für Details siehe [Fel17] §C. Die hier dargestellte Eigenschaft folgt offensichtlich aus der Konstruktion.

[2] Der Typ wird für die Serialisierung von Referenzen verwendet. Eine Implementierung ist also ohnehin vorhanden und kann folglich dem Werkzeugbauer zur Verfügung gestellt werden.

Sie werden konsequenterweise `f32` und `f64` genannt. Diese entsprechen in Java den Typen `float` und `double`.

Strings sind Sequenzen aus beliebigen Unicode-Zeichen. Ihre Länge wird neben den Zeichen gespeichert, wie das beispielsweise in Java üblich ist. Es ist folglich einerseits möglich, null-Zeichen in der Mitte des Strings zu haben und andererseits enden Strings im Allgemeinen nicht mit einem null-Zeichen. Diese Eigenschaft ist vor allem in der Interaktion mit der Programmiersprache C [ISO11] relevant, da dort null-terminierte Strings zum Einsatz kommen. Diese Darstellung erfordert in C also einen gewissen Anpassungsaufwand, entspricht dafür aber dem üblichen Vorgehen in Ada, C++ und Java. Strings werden, ähnlich wie in Java, per Referenz in Objekten gespeichert. Daher können sie den Wert `null` annehmen. Ferner werden gleiche Strings beim Serialisieren automatisch unifiziert, da sie ohnehin nicht unterscheidbar wären.

Man kann per Referenz auf in der Austauschdatei gespeicherte Objekte verweisen. Referenzen sind polymorph, stark typisiert und können den Wert `null` annehmen. Dies entspricht dem Referenzbegriff in Java. Ferner ist es möglich, über den Typ `annotation` eine Referenz auf ein beliebiges Objekt zu speichern.[1] Hierdurch können beispielsweise Erweiterungspunkte[2] eingefügt werden, wenn die Art der Erweiterung noch unklar ist. Ein Beispiel hierfür wäre etwa eine Referenz auf das Ergebnis einer Werkzeugausführung. Dieser Typ entspricht in Java einer Wildcard, also `<?>`. Ihn mit dem Java-Typ `java.lang.Object` zu vergleichen trägt insofern nicht, als es nicht möglich ist, diesen direkt zu instantiieren.

[1] Würde man `annotation` als Supertyp aller nutzerdefinierter Typen betrachten, müsste man entweder auf alle Optimierungen verzichten, welche auf Basistypen basieren, oder die Typtheorien vor Werkzeug- und Dateispezifikation würden sich grundlegend unterscheiden. Zu den, auf Basistypen basierenden, Optimierungen zählen auch Datenparallele Ausführung von Operationen auf Basistypen, wie etwa die Vorbereitung von Instanzen für die Serialisierung.

[2] Damit sind Felder gemeint, welche Daten referenzieren, die noch nicht spezifiziert sind. Ein Beispiel hierfür wäre etwa ein Typ, welcher eine Werkzeugausführung in der Austauschdatei dokumentiert. Soll in diesem auf die Wurzel seines Resultats verwiesen werden, so steht dieser Typ für alle Werkzeuge, die ein zukünftiger Werkzeugbauer entwickeln wird, nicht fest. Eine Werkzeugkette, deren IR-Design derartige Situationen von Beginn an antizipiert hat, benötigt den Typ `annotation` nicht, da das Problem in diesem Fall über Vererbung gelöst werden kann.

```
BasicBlock {
  list<Instruction> instructions;
}
```

Listing 5.3: BasicBlock besteht aus Instruktionen

```
BasicBlock {
  InstructionList instructions;
}
InstructionList {
  list<Instruction> content;
}
```

Listing 5.4: BasicBlock verwendet Instruktionen

In SKilL verwendete Container sind Arrays konstanter und variabler Länge, sowie Listen, Sets und mehrstellige Maps.[1] Container werden stets flach in Objekte eingebettet. Dieser Entscheidung liegt der Wunsch zugrunde, unterscheiden zu können, ob ein Objekt beispielsweise eine Instruktionsliste besitzt oder ob es diese lediglich verwendet (siehe Listing 5.3 bzw. 5.4). Lagert man die Liste in eine neue Typdefinition aus, so wird diese dadurch per Referenz gespeichert. Es ist ferner nicht möglich, Containertypen ineinander zu verschachteln, was bedeutet, dass eine Liste aus Arrays aus Referenzen beispielsweise nicht erlaubt ist.[2] Um beim letzten Beispiel zu bleiben, wäre es aber selbstverständlich zulässig, ein Array aus `InstructionList`-Referenzen zu erzeugen. Mehrstellige Maps sind rechts-assoziativ.

[1] Aus Sicht der SKilL-Typtheorie sind Container keine Typen, sondern, im mathematischen Sinn, Funktionen auf Typen. Die Interpretation der SKilL-Typtheorie entspricht der Herbrand-Interpretation. Es gibt keine Möglichkeit weitere Funktionen auf Typen zu definieren.

[2] Hierdurch wird versucht das versehentliche Kopieren großer Containerstrukturen zu verhindern.

```
A {
A f; -> ok
}
B {
B f; -> illegal (doppelter Name)
A f; -> illegal (doppelter Name)
}
C : A {
A f; -> illegal (existiert in Supertyp)
}
D {
A f; -> legal
}
```

Listing 5.5: Beispiel: Sichtbare Felder

### 5.1.3. Vererbung, Sichtbarkeit und Typäquivalenz

SKilL verwendet zunächst[1] nur Einfachvererbung. Grund hierfür ist primär die einfache Abbildbarkeit auf gängige Programmiersprachen. Wie von Programmiersprachen bekannt, erben Subtypen alle Felder ihrer Supertypen. Es ist nicht zulässig, von vordefinierten Typen, wie etwa `string`, zu erben.[2]

Da es sich um ein reines Datenbeschreibungsformat handelt, wäre eine eingeschränkte Sichtbarkeit im Sinne privater Felder nicht sinnvoll. Ohne zugreifbare Methoden, die den internen Zustand eines Objekts manipulieren, wäre dieser für den Werkzeugbauer wertloser Ballast. Infolge der daraus resultierenden allgemeinen Sichtbarkeit aller Felder ist es nicht zulässig, ein Feld zu überschreiben. Im Kontext der Spezifikationssprache gelten Felder als äquivalent, wenn ihre Namen äquivalent sind und sie im selben

[1] Die Grammatik lässt mehrere Supertypdefinitionen zu. Diese werden in §5.4.1 verwendet, um Interfaces als Erweiterung der Spezifikationssprache einzuführen.

[2] Würde man dies erlauben, so wäre man in Programmiersprachen, in denen dies ebenfalls verboten ist, mit dem Problem konfrontiert, dass man die entstehenden Typen abbilden muss. Dies hätte zur Folge, dass man einen Stellvertreter für eingebaute Typen verwenden müsste, was die Nutzbarkeit des entstehenden APIs beeinträchtigt.

```
A {
 B f; -> ok
}
B {
 A f2; -> ok
 C f3; -> illegal
}
```

Listing 5.6: Beispiel: Sichtbare Typen

```
Type {} -> wie "type"
type {} -> wie "type"
TYP3 {} -> wie "typ3"
typE {} -> wie "type"
typ_e {} -> wie "type"
```

Listing 5.7: Beispiel: Typ- bzw. Namensäquivalenz

Sichtbarkeitsbereich liegen (siehe Listing 5.5). Sichtbar sind alle Felder des Typs und alle sichtbaren Felder aller Supertypen.

Weiterhin gibt es keine Deklarationsreihenfolge, d.h. alle deklarierten Typen können auch uneingeschränkt verwendet werden. Nicht deklarierte Typen dürfen nicht verwendet werden, d.h. sie werden nicht implizit eingeführt. Es wäre zwar immer möglich, eine korrekte und zur tatsächlichen Lösung kompatible Definition eines unbekannten Typs einzuführen. Allerdings wäre die `.skill`-Spezifikation hierdurch anfällig für Tippfehler (siehe Listing 5.6).

Typen gelten als äquivalent, falls sie unter demselben Namen definiert wurden. Dabei wird die Groß-/Kleinschreibung ignoriert (siehe Listing 5.7). Ebenso werden einzelne Unterstriche ignoriert. Dadurch werden beispielsweise die Worttrennungskonventionen der Programmiersprachen Java, C und Ada abgebildet, sodass ungeachtet der in der `.skill`-Spezifikation verwendeten Konvention die Konvention der Zielsprache im generierten API

⟨*field*⟩ ::= ⟨*description*⟩ (⟨*constant*⟩ | ⟨*data*⟩ | ⋯ ) ';'

⟨*constant*⟩ ::= `const` ⟨*type*⟩ ⟨*ID*⟩ '=' ⟨*int*⟩

Abbildung 5.2.: Produktion für Konstanten

verwendet werden kann (siehe Listing 5.9 und 5.10, bzw. §6.1).

### 5.1.4. Konstanten

Um beispielsweise Dateiversionen zu markieren, wurden Konstanten in SKilL eingebaut. Diese haben einen fixen Wert und verbrauchen unabhängig von der Zahl der Instanzen konstant viel Speicher. Es können nur Integer-Werte verwendet werden. Um diese in der `.skill`-Spezifikation zu repräsentieren, wird die Grammatik um eine Produktion für konstante Felder erweitert. Diese Produktion ist eine weitere Alternative in der Produktion *field* (siehe Abb. 5.2).

### 5.1.5. Dokumentation in `.skill`-Spezifikationen

Da aus der `.skill`-Spezifikation ein API generiert wird, ist in SKilL, im Gegensatz zu verbreiteten Programmiersprachen, ein Kommentar ein echtes Token und Bestandteil der Grammatik. Dies hat den Effekt, dass vor jeder Typ- und Felddeklaration maximal ein Kommentar stehen darf, welcher der nachfolgenden Deklaration zugeordnet wird. Dadurch können die Kommentare vom Front-End des Quellcode-Generators verarbeitet und an die Quellcode-Generator-Back-Ends weitergereicht werden, sodass das API anschließend mit den Kommentaren aus der `.skill`-Spezifikation versehen ist. In Listing 5.8 ist ein leicht gekürztes Beispiel aus der SKilL-Testsuite dargestellt. In Listing 5.9 und 5.10 sehen wir das gekürzte Ergebnis für Java und Ada.

Um diesen Aspekt zu realisieren, wurde in der Grammatik die Regel `description` um Kommentare erweitert. Da der Quellcode-Generator auch

```
/**
 * The age of a person.
 * @author Timm Felden
 */
Age {
  /**
   * People have a small positive age, but maybe they
   * will start to live longer in the future, who knows
   */
  v64 age;
}
```

Listing 5.8: Kommentare in der Spezifikation

andere Anweisungen erhalten soll, welche die generierte Anbindung beeinflussen, werden an diesem Punkt noch Hints und Restrictions (siehe folgender Abschnitt) in die Grammatik eingeführt. Dies führt zu den in Abb. 5.3 dargestellten Produktionen. An dieser Stelle sei bemerkt, dass die Implementierung einfacher ist, wenn Kommentare zunächst als eigenes Token eingelesen und danach mithilfe eines zweiten Parsers zerlegt werden. Hierdurch lässt sich der Inhalt der Kommentare praktikabel auf Worte aufteilen, welche genutzt werden können, um unterschiedlich breite Kommentare zu erzeugen.

### 5.1.6. Hints und Restrictions

Der generierte Code lässt sich durch die `.skill`-Spezifikation für jedes Feld und jeden Typ mittels sogenannter Hints und Restrictions beeinflussen. Konzeptionell wird zwischen im serialisierten Typsystem persistierten Modifikationen (*Restrictions*) und nicht serialisierten Modifikationen (*Hints*) unterschieden. Aus Sicht der Spezifikationssprache erschließen sich die beiden Konstrukte direkt aus der Grammatik (Abb. 5.3).

In dieser Arbeit werden die Restrictions sind `@singleton`, `@abstract`, `@default`, `@nonnull`, `@range` und `@coding` verwendet. `@singleton` mo-

```java
/**
 *  The age of a person.
 *  @author  Timm Felden
 */
public class Age extends SkillObject {
  /**
   * People have a small positive age, but maybe they
   * will start to live longer in the future, who knows
   */
  final public long getAge() { [...] }

  [...]

}
```

Listing 5.9: Kommentare in der Spezifikation (Java)

```ada
type Age_T is new Skill.Types.Skill_Object with private;

--  The age of a person.
--  @author  Timm Felden
type Age is access all Age_T;

--  People have a small positive age, but maybe they will
--  start to live longer in the future, who knows
function Get_Age (This : not null access Age_T'Class)
    return Standard.Skill.Types.V64;

[...]
```

Listing 5.10: Kommentare in der Spezifikation (Ada)

⟨*description*⟩ ::= ⟨*comment*⟩? (⟨*restriction*⟩ | ⟨*hint*⟩)*

⟨*comment*⟩ ::= '/' '*'+ ⟨*comment-text*⟩ (⟨*comment-TAG*⟩ ':'? ⟨*comment-text*⟩)* '*/'

⟨*comment-text*⟩ ::= (⟨*comment-prefix*⟩? (~['*/' | ⟨*comment-TAG*⟩ | '\n'])*)*

⟨*comment-prefix*⟩ ::= '\n' ⟨*whitespace*⟩* ('*' ⟨*whitespace*⟩+)?

⟨*comment-TAG*⟩ ::= '@' [a-z]+

⟨*restriction*⟩ ::= '@' ⟨*ID*⟩ ('(' (⟨*argument*⟩ (',' ⟨*argument*⟩)*)?')')?

⟨*hint*⟩ ::= '!' ⟨*ID*⟩ ('(' (⟨*argument*⟩ (',' ⟨*argument*⟩)*)? ')')?

⟨*argument*⟩ ::= ⟨*FLOAT*⟩ | ⟨*INT*⟩ | ⟨*STRING*⟩ | (⟨*ID*⟩ (('.'|'::') ⟨*ID*⟩)*)

Abbildung 5.3.: Beschreibungsmöglichkeiten in `.skill`-Spezifikationen

difiziert einen Typ so, dass es genau eine Instanz des Typs gibt. `@abstract` modifiziert einen Typ so, dass es genau keine statische Instanz des Typs gibt. `@default` modifiziert für Referenzen sowie Ganz- und Gleitkommazahlen ein Feld so, dass dessen Standardwert dem Argument entspricht. `@nonnull` modifiziert für Referenzen ein Feld so, dass dessen Wertebereich keinen null-Zeiger mehr enthält. `@range` modifiziert für Ganz- und Gleitkommazahlen ein Feld so, dass dessen Wertebereich dem Argument entspricht. `@coding` modifiziert ein Feld so, dass für die Felddaten eine andere Serialisierungsfunktion entsprechend dem Argument verwendet wird. Weitere Restrictionen sind in [Fel17] §5.1 beschrieben.

In dieser Arbeit werden die Hints `!Distributed` und `!OnDemand` verwendet. `!Distributed` instruiert den Quellcode-Generator, bei der Implementierung eines Feldes anstelle eines Feldes im dazugehörigen Objekt eine `HashMap` zu verwenden, welche dem Objekt den entsprechenden Wert zuordnet. `!OnDemand` instruiert den Quellcode-Generator, bei der Implementierung eines Feldes anstelle des regulären direkt lesenden Codes einen

bei Bedarf lesenden Code zu verwenden (siehe §6.3.5 bzw. §7.6). Beide Hints verändern das generierte API nicht, d. h. sie nachträglich hinzuzufügen, erfordert keine Anpassung am Anwendungscode. Weitere Hints sind in [Fel17] §5.2 beschrieben.

### 5.1.7. Spezifikations-Header

In der Spezifikationssprache existiert bewusst keine Möglichkeit, Kommentare zu verwenden, die sich an beliebiger Stelle unterbringen lassen und die vom Parser ignoriert werden. Um dennoch beispielsweise Lizenzinformationen in einer `.skill`-Spezifikationsdatei einfügen zu können, wurden Kopfkommentare eingeführt, welche mit einer Raute ('#') beginnen und sich bis zum Ende der Zeile erstrecken.

Dieser Mechanismus kann auch für Metainformationen genutzt werden. So wird beispielsweise vom Testgenerator der SKilL-Implementierung die erste Zeile der `.skill`-Spezifikation als Kommando interpretiert, um die generierten Testimplementierungen anpassen zu können.

Ferner ist es hilfreich, `.skill`-Spezifikationen in logische Teile zu untergliedern. Hierfür gibt es in `.skill`-Spezifikationen einen Importmechanismus, welcher Typdefinitionen aus einer anderen `.skill`-Datei einbindet. Dieser arbeitet ähnlich wie das `include`-Makro in C, d.h. alle Importe werden vor der semantischen Analyse durchgeführt und die Importe werden transitiv ausgewertet. Es werden jedoch im Gegensatz zu C automatisch keine `.skill`-Dateien doppelt importiert. Die Bearbeitung der Importe vor der semantischen Analyse ermöglicht die Verwendung von `.skill`-Spezifikationsdateien, die für sich genommen nicht zulässig wären. Dieser Umstand wäre durch das Einfügen weiterer Importe in diese Spezifikationsdateien zu beheben, erspart in der praktischen Anwendung aber unnötigen Arbeitsaufwand.

Die Anpassung der Grammatik ist relativ geradlinig (Abb. 5.4). Es ist zu beachten, dass für die korrekte Verarbeitung der Spezifikation zunächst alle über Imports erreichbaren Dateien eingelesen und in ASTs überführt werden müssen. Die Verarbeitung der daraus resultierenden ASTs muss dann in

⟨*header*⟩ ::= ⟨*head-comment*⟩* ⟨*include*⟩*

⟨*head-comment*⟩ ::= '#' (~[\n])* '\n'

⟨*include*⟩ ::= (`include`|`with`) ⟨*string*⟩+

Abbildung 5.4.: Der Kopf einer Spezifikation

einem weiteren Schritt gemeinsam erfolgen. Dies ist insofern unproblematisch, als die IR-Spezifikationen aus Sicht von Compilerbau-Technologie mit höchstens einigen tausend Zeilen nicht umfangreich sind. Insgesamt zeigt sich, dass die Laufzeit des gesamten Quellcode-Generators deutlich kürzer ist als die der danach aufzurufenden Zielsprachen-Compiler.

### 5.1.8. Bezeichner und Schlüsselwörter

Im Gebiet der Bezeichner und Schlüsselworte ergibt sich der Zielkonflikt, dass man einerseits dem Werkzeugbauer möglichst jede Zeichenkette als Bezeichner gestatten möchte, die er für sinnvoll erachtet. Andererseits sollte man diese Bezeichner dann möglichst unverändert in die generierten APIs übernehmen. Betrachtet man verbreitete Programmiersprachen, so fällt schnell auf, dass die Liste aller Bezeichner, die in einer dieser Programmiersprachen ein Schlüsselwort oder ein Standardtyp sind, unbrauchbar lang wäre.

Konkretes Beispiel sei der Name `Unit`, welcher sich scheinbar für die Bezeichnung einer Übersetzungseinheit in der Zwischendarstellung eignet. Dieser Name wird aber unter anderem in Scala verwendet, um das Äquivalent von `void` in Java zu bezeichnen. Um diesem Problem zu begegnen, sind die Quellcode-Generator-Back-Ends so implementiert worden, dass sie mit derartigen Konflikten umgehen können.[1] Dies geschieht einerseits durch Bereinigung von Bezeichnern oder unzulässiger Zeichen in Bezeichnern.

[1] Hierfür muss für jedes Back-End lediglich die Liste der Schlüsselwörter der Zielsprache und das Verdecken von Standardtypen der Zielsprache berücksichtigt werden.

Andererseits kann das Problem teilweise durch vollständige Benennung[1] von Typen gelöst werden.

Da es mit Ada eine Programmiersprache gibt, deren Bezeichner von der Groß-/Kleinschreibung unabhängig sind, wurde dieses Verhalten in SKilL übernommen. Ein positiver Seiteneffekt hiervon ist, dass man Typnamen, welche mit Schlüsselwörtern der Spezifikationssprache kollidieren, trotzdem verwenden kann, indem man sie mit einem Großbuchstaben beginnen lässt.

Da unzulässige Zeichen ohnehin durch eine Bereinigungssequenz abgebildet werden können, ist aus Sicht der Spezifikationssprache jedes Unicode-Zeichen, welches kein ASCII-Zeichen ist, neben den üblichen Regeln für Bezeichnern zulässig. Es wird vom Werkzeugbauer erwartet, dass er sich selbst sinnvolle Regeln auferlegt. Die derzeit implementierte Generatorinfrastruktur erlaubt es für einzelne Bezeichner zu prüfen, ob sie für eine Zielsprache direkt nutzbar wären, um dem Werkzeugbauer zu erlauben, die Wahl von Bezeichnern zu beurteilen, ohne eine Anbindung zu erzeugen. Die Definition einer Funktion für die Bereinigung von Bezeichnern erfolgt durch das Back-End des Quellcode-Generators und ist für dieses spezifisch.[2]

## 5.2. Das Binärformat

Das Binärformat von SKilL ist ein auf IO-Performance optimiertes Format. Dabei ist nicht vorgesehen, dass es direkt von Menschen interpretiert oder gar bearbeitet wird. Für binäre Austauschdateien wird per Konvention die Dateiendung `.sf` verwendet. Diese können durch kontextabhängigen rekursiven Abstieg in einem Pass und linearer[3] Zeit gelesen werden. Der Kontext des Parsers besteht dabei aus den aus der Werkzeugspezifikation bekannten Typinformationen sowie den Typinformationen und den Daten aus der bereits gelesenen Austauschdatei. Das Format ist so entworfen, dass man einen Fehler möglichst früh und mit geringem Aufwand erkennen kann.

[1] Also beispielsweise `_root_.scala.Unit` und `_root_.myPackage.Unit`.

[2] SKilL spezifiziert hierzu nur die Existenz und auch diese nur als Feature.

[3] Ist man an Felddaten nicht interessiert, kann es im konkreten Fall auch sublinear sein, da sich Daten einfach überspringen lassen.

Ferner ist es beim Lesen nicht notwendig, Daten in einem uninterpretierten Zwischenzustand[1] aufzubewahren.

Das Format wurde vor allem mit dem Ziel entwickelt, leicht implementierbar zu sein und sehr hohe Leseraten zu bieten.[2] Die Annahme, dass sich aus einer schnellen Leseoperation und einer dazu symmetrischen Schreiboperation auch eine schnelle Schreiboperation ergeben würde, hat sich bestätigt (siehe §7.3.1).

Entwirft man ein ohnehin kontextsensitives Binärformat, so ist es ratsam die Informationen darin so zu organisieren, dass zu jedem Zeitpunkt alle zur Konsistenzprüfung erforderlichen Daten vorliegen. Das bedeutet beispielsweise, dass man erst alle Typen definiert und dann zu jedem Typ die Felder definiert, da die dann auftretenden Feldtypen zwingend auf bereits bekannte Typen verweisen müssen, oder fehlerhaft sind. Dadurch wird zwar die Beschreibung des Binärformats deutlich schwerer zu verstehen, dafür erhält man eine Implementierung, die Informationen in wachsenden Arrays aufsammelt und auf diese über Indices effizient zugreifen kann. Zusätzlich ist die Fehlererkennung erheblich besser als im ersten Design [Fel13], da Fehler zum frühst möglichen Zeitpunkt berichtet werden können.

### 5.2.1. Überblick

In diesem Abschnitt soll zunächst ein Überblick über das Gesamtformat vermittelt werden. Die Details der einzelnen Aspekte und Designentscheidungen werden anschließend in eigenen Abschnitten beleuchtet. Beispiele für das Format finden sich in §5.2.2.

Um das Design des Binärformats verstehen zu können, muss man die Anbindung kurz umreißen. Die Anbindung verwaltet Objekte, welche Instan-

[1] Wie etwa dem AST einer `.sf`-Datei, der in einem weiteren Pass attributiert wird. Das Format ist so entworfen, dass eine Attributierung in einem Pass erfolgen kann, ohne dabei einen AST aufzubauen.

[2] Ersteres ergab sich vor allem aus dem Plan, Bauhaus nach spätestens einem Jahr vor allem durch studentische Arbeiten auf SKilL umstellen zu können. Letzteres basierte auf der Beobachtung, dass man während der Entwicklung neuer Analysen sehr viel Zeit damit verbringt, gerade gewonnene Analyseergebnisse zu untersuchen. Dabei ist man primär damit beschäftigt, Daten zu lesen und in sehr kompakter Form darzustellen.

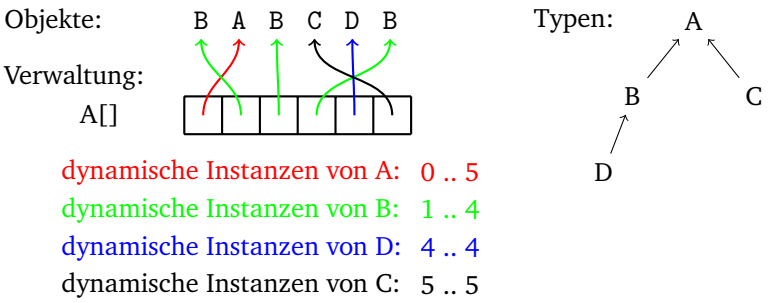



Abbildung 5.5.: Objektverwaltungsbeispiel.

zen der Typdefinitionen der IR-Spezifikation sind. Diese Verwaltung erfolgt konzeptionell[1] durch ein Array aus Zeigern auf alle verwalteten Objekte. Das Zustandekommen dieses Arrays wird im Kapitel Implementierung besprochen, da es für das Design von SKilL nicht erheblich ist. In Abbildung 5.5 findet sich ein Beispiel. In diesem Beispiel werden zu einer Typhierarchie mit vier Typen sechs Instanzen erzeugt und verwaltet. Die Verwaltung erfolgt durch ein Array aus Zeigern. Der Zeigertyp entspricht dem Basistyp der Typhierarchie. Das Array enthält für jedes verwaltete Objekt exakt einen Zeiger. Die Zeiger sind im Array topologisch sortierten Buckets angeordnet, d.h. falls zwei Zeiger desselben Typs existieren, dann sind auch alle Zeiger dazwischen von diesem Typ und falls ein Zeiger auf die Instanz eines Typs $T$ zeigt, dann sind Zeiger auf Instanzen eines Typs $U$ mit $U <: T$ rechts davon angeordnet. Eine Konsequenz dessen ist, dass alle Zeiger auf dynamische Instanzen eines beliebigen Typs in einem zusammenhängenden Indexbereich liegen, dieser also durch den Start- und Endindex repräsentiert werden kann. Der im Folgenden immer wieder auftauchende Begriff *Base Pool Offset (BPO)* bezeichnet eben diesen Startindex. Mit dieser Repräsentation im Hinterkopf kann nun das Austauschformat und dessen Eigenschaften grob umrissen werden.

[1] Die Wahrheit ist aufgrund der Objektorientierung und der Möglichkeit, Objekte hinzuzufügen und zu löschen, etwas komplizierter (siehe §6.3.3).

Auf oberster Ebene wird die `.sf`-Datei in sogenannte Blöcke unterteilt. Dies ist eine Konsequenz der Möglichkeit, durch reines Anhängen von Daten neue Felder, neue Instanzen oder neue Typen zu erzeugen.[1]

Da Typen in SKilL über ihre Namen identifiziert werden, werden Strings vor allen anderen Informationen gespeichert. Um das bedarfsorientierte Lesen der Daten aus einer `.sf`-Datei zu realisieren, sind Strings so strukturiert, dass sie einzeln bei Bedarf gelesen werden können.[2] Dieser Teil wird Stringblock genannt. Den Rest nennen wir Typblock. Eine `.sf`-Datei besteht aus einer alternierenden Sequenz von Stringblöcken und Typblöcken, die immer mit einem Typblock endet. Ein leerer Block belegt dabei exakt ein Byte, d.h. die Regularität hat einen geringen Preis. Eine `.sf`-Datei wurde erfolgreich gelesen, wenn am Ende eines Typblocks das Dateiende erreicht wurde.

Ein Typblock besteht zunächst aus Typdefinitionen.[3] Diesen folgen die Felddefinitionen. Am Ende finden sich schließlich die Felddaten. Um Typinformationen bei jedem Vorkommen auf Konsistenz mit der Werkzeugspezifikation prüfen zu können, sind Typen topologisch sortiert, d.h. Supertypen müssen vor ihren Subtypen gespeichert werden. Da Felddefinitionen hinter allen Typdefinitionen angeordnet sind, kann auch hier jeder vorkommende Feldtyp direkt auf Konsistenz mit der Werkzeugspezifikation geprüft werden. Diese Struktur ist in Abb. 5.6 skizziert.

Objekte existieren im Binärformat nur implizit als Instanzen von Typen.[4] Jeder Typ enthält in seiner Definition die Anzahl seiner dynamischen Instanzen.[5] Betrachtet man nur die dynamischen Instanzen von Basistypen, so ergibt sich ein einfaches und eindeutiges Benennungsschema über ihren Index.[6] Um die entsprechenden Indices auch für Subtypen effizient berechnen zu können ohne auf das Objekt zurückgreifen zu müssen,[7] wird jedem Subtyp eine Verschiebung des ersten Index gegenüber dem Basistyp zugeordnet.

[1] Forderung aus §1.1.11
[2] Forderung aus §1.1.10
[3] Forderung aus §1.1.7
[4] Forderung aus §1.1.8
[5] D.h. direkte Instanzen plus Summe dynamischer Instanzen aller direkter Subtypen.
[6] Das serialisierte Typsystem verwendet Einfachvererbung.
[7] Es gibt keinen Grund anzunehmen, dass ein Objekt immer oder überhaupt existiert.

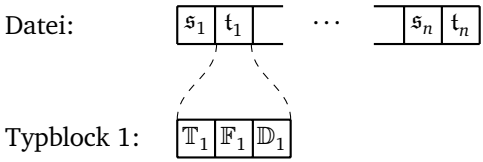



Abbildung 5.6.: Grobstruktur des Binärformats für eine Datei, die aus einer Schreib- und $n-1$ Anhängeoperationen hervorgegangen ist. Bezeichner orientieren sich an [Fel17] §B: $\mathfrak{s} \cong$ String-Block, $\mathfrak{t} \cong$ Typblock, $\mathbb{T} \cong$ Typdeskriptoren, $\mathbb{F} \cong$ Felddeskriptoren, $\mathbb{D} \cong$ Daten.

Diese Verschiebung wird BPO genannt und sorgt dafür, dass das Intervall zulässiger Indices der zu einem Typ gehörenden Objekte für jeden Typ in konstanter Zeit berechnet werden kann. Zur Laufzeit wird jedes gelesene Objekt mithilfe eines Arrays im Basistyp verwaltet. Die Offsets entsprechen dabei genau der Verschiebung gegenüber dem ersten zulässigen Index in diesem Array. Wären Offsets relativ zum Supertyp, so wäre der Zugriff auf dieses Array linear teuer in der Anzahl der Supertypen. Die Intervalle sind für alle Typen zusammenhängend, da Objekte in Typordnung sortiert gespeichert werden. Die direkten Instanzen eines Typs finden sich also zwischen dem BPO und dem niedrigsten BPO seiner Subtypen und die Instanzen, also inklusive Subtypen, befinden sich zwischen BPO und BPO + count, also der Zahl der Instanzen des Typs. Gibt es keine Subtyen, so sind alle Instanzen auch direkte Instanzen.

Die hierdurch entstehende Kodierung des Typsystems ist nicht nur effizient, sondern bietet auch eine leicht prüfbare Redundanz, welche Implementierungsfehler in der Anbindung entdecken kann. Auch bei Instanzen gilt eine Sortierung in topologischer Ordnung. Dadurch kann man Bedingungen prüfen, wie etwa die, dass ab dem ersten Subtyp alle Indices genau einem Subtyp zugeordnet sind und dass die Anzahl der Instanzen den Indexberei-

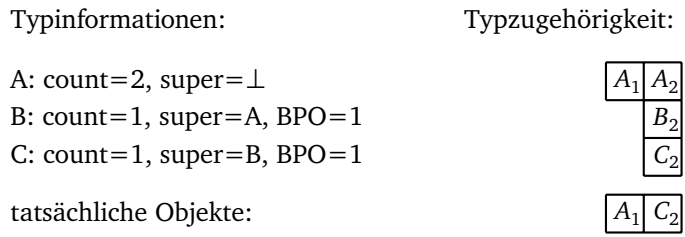



Abbildung 5.7.: Indexbereiche im Kontext von Subtypen. Indexierung der Objekte und BPOs sind immer relativ zum Basistyp einer Hierarchie. BPOs sind Verschiebungen des logischen Index relativ zum Basistyp.

chen der Subtypen entsprechen.[1] Diese Prüfung ist pro Typ konstant teuer, wenn man lediglich mit Indexbereichen rechnet, und damit günstig. Das Schema ist in Abb. 5.7 dargestellt.

Die Felder eines Objekts stehen erst fest, nachdem alle Typblöcke gelesen wurden. Jede Typdefinition jedes Blocks definiert, wie viele Felder sie enthält. Im Abschnitt der Felddefinitionen werden die einzelnen Definitionen in der Reihenfolge der Typdefinitionen abgearbeitet, sodass aus dem Kontext geschlossen werden kann, zu welchem Typ sie gehören. Mit dieser Kodierung erhält man das Wissen über alle existierende Typen zum Zeitpunkt der Interpretation eines Feldtyps, ohne in jedem Feld speichern zu müssen, zu welchem Typ es gehört.

Felddefinitionen enden mit einem Offset in den Felddatenbereich. Der Felddatenbereich ist so groß wie der größte Offset und beginnt bei 0. Da es sich bei Felddefinitionen um relativ kleine und variabel lange Daten handelt, sind die Mehrkosten für die Berechnung der tatsächlichen Wertebereiche eine Variable, die den letzten Offset enthält. Diese Zuordnung ist in Abb. 5.8 skizziert.

Da nun die Struktur des Binärstroms und das enthaltene Typsystem be-

[1] Diese Konsistenzeigenschaften werden vom Format vorgeschrieben. Sie ermöglichen es, für beliebige Typen mit $n$ dynamischen Instanzen, diese durch eine Arraytraversierung mit genau $n$ Inkrementierungen zu besuchen.

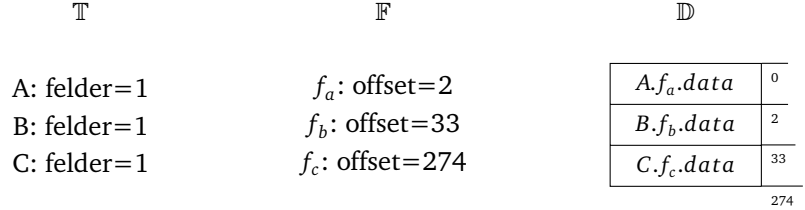



Abbildung 5.8.: Zuordnung von Feldern und Felddaten. Die Inhalte der einzelnen Abschnitte des Blocks werden der Übersichtlichkeit halber übereinander dargestellt. Offsets in diesem Zusammenhang sind relativ zum Datensegment $\mathbb{D}$ und haben die Einheit Byte. Ihr Wert entspricht der Position des letzten, zu den Daten des Feldes gehörenden Bytes.

kannt ist, bleibt lediglich die Frage, wie die eigentlichen Felddaten der impliziten Objekte so abgelegt werden können, dass eine Zuordnung zu den Objekten erfolgen kann. Auch hier wird wieder auf die Reihenfolge der Objekte zurückgegriffen. Jedes Felddatum wird in aufsteigender Indexreihenfolge in den Datenbereich des entsprechenden Feldes geschrieben. Hierbei sei bemerkt, dass das, die entsprechende Hardware und Bibliotheken vorausgesetzt, natürlich inhärent parallelisierbar ist, da Felder auf jeweils getrennten Speicherbereichen operieren (siehe Abb. 5.9).

Neben der offensichtlichen Parallelisierbarkeit der Deserialisierung von Felddaten fällt auch auf, dass einzelne Felder auch bei Bedarf deserialisiert werden können. In diesem Szenario werden zunächst die strukturellen Informationen verarbeitet. Dabei handelt es sich um wenige hundert bis ein paar tausend Bytes. Den Großteil der Daten kann man überspringen. Werden die Offsets zu nicht gelesenen Feldern gespeichert, so kann man diese jederzeit[1] nachladen. Dabei ist es natürlich ohne Weiteres möglich, diese Entscheidung pro Feld zu treffen, was bedeutet, dass einzelne interessante Felder direkt beim Öffnen der `.sf`-Datei geladen werden können und weniger interessante Felder erst dann, wenn sie wirklich genutzt werden. Felder, die nicht in der

[1] Unter der Annahme, dass es keine parallelen externen Modifikationen der `.sf`-Datei gibt.

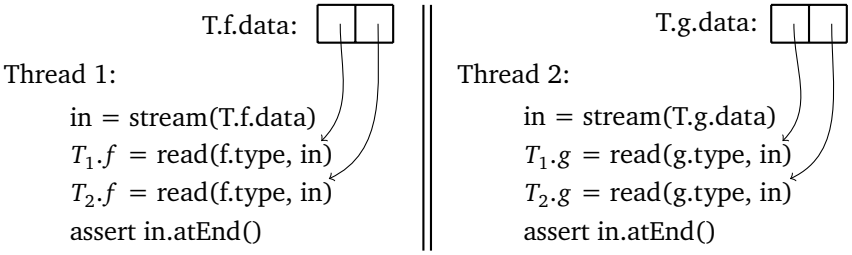



Abbildung 5.9.: Parallele Deserialisierung der Daten zweier Felder eines Typs $T$ mit zwei Instanzen. So können beispielsweise die in Abb. 5.10 bzw. 5.12 dargestellten Felder `fx` und `farbe` parallel bearbeitet werden.

```
1 Node {
2   i8 fx;
3 }
```

Listing 5.11: Spezifikation für Beispielbinärdatei

Werkzeugspezifikation enthalten sind, werden pauschal als uninteressant betrachtet und erst bei Bedarf gelesen. An dieser Stelle ist klarzustellen, dass ein Neuschreiben einer `.sf`-Datei einer Benutzung aller Felder entspricht, da sich hierdurch Indexbereiche verschieben können und diese die alte Serialisierung der dadurch repräsentierten Referenzen ungültig macht. Kann man das Ergebnis eines Werkzeugs hingegen an eine `.sf`-Datei anhängen, so ist es tatsächlich möglich, Daten nicht zu interpretieren. Letzteres gilt natürlich auch für rein lesende Werkzeuge oder Werkzeuge, die andere `.sf`-Dateien erzeugen, wie etwa ein Binder.

### 5.2.2. Beispiele für .sf-Dateien

Die in diesem Abschnitt zur Illustration des Formats verwendeten Beispiele sind an Beispiele aus der SKilL-Sprachspezifikation angelehnt [Fel17] (Fig.

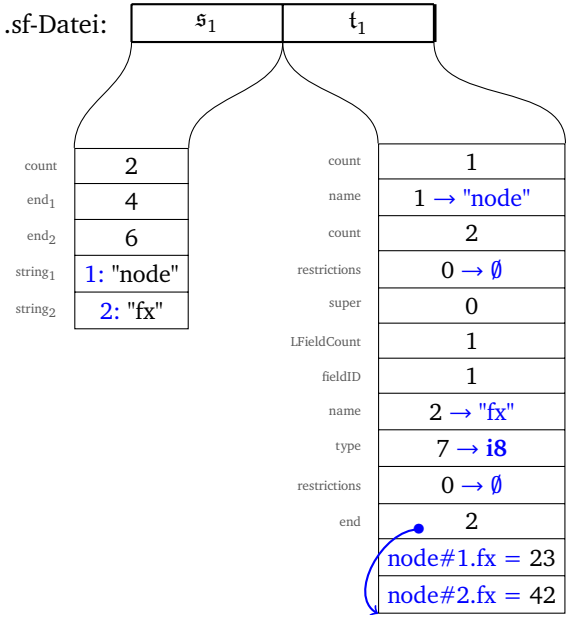



Abbildung 5.10.: Beispiel einer einfachen .sf-Datei, welche zwei Objekte enthält. Die Interpretation der Werte ist blau dargestellt. Die Bedeutung des Werts aus Sicht des Formats ist grau dargestellt.

3 & 4). Das erste Beispiel verwendet die Spezifikation in Listing 5.11. Es werden zwei Knoten erzeugt. Als Werte für `fx` wird 23 respektive 42 gewählt. Die beim Schreiben entstehende Datei ist in Abb. 5.10 dargestellt. Das verwendete Schema entspricht den Schemata der folgenden Abschnitte (siehe Abb. 5.13 und 5.14).

Als nächstes wird das Beispiel mit einer Anhängeoperation erweitert, damit ein zweites Paar aus Blöcken ensteht. Hierfür wird dieselbe Spezifikation verwendet. Es werden zwei Knoten angehängt, deren `fx`-Felder die Werte -1 und 2 erhalten. Die entstehende Datei ist in Abb. 5.11 dargestellt. Da sowohl `node` als auch `fx` bereits bekannt sind, sind die Typinformationen

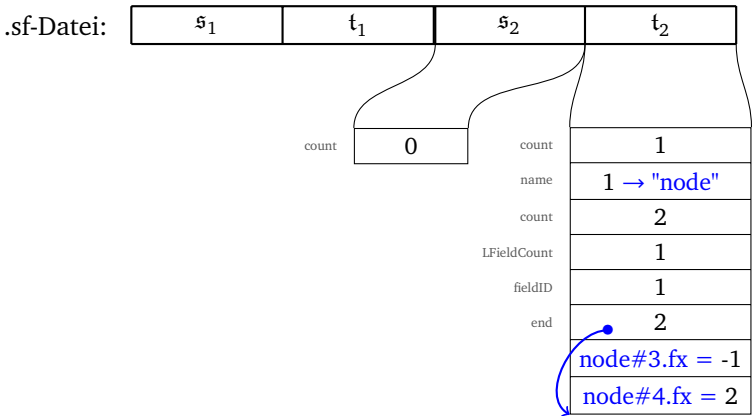


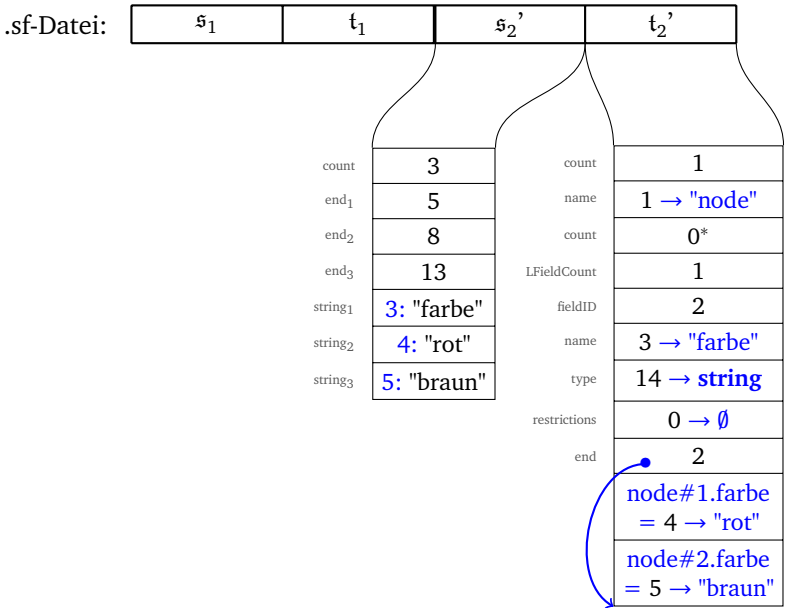

Abbildung 5.11.: Beispiel einer .sf-Datei, welche zwei weitere Objekte hinzufügt. Die Interpretation der Werte ist blau dargestellt. Die Bedeutung des Werts aus Sicht des Formats ist grau dargestellt.

```
1 Node {
2   string farbe;
3   i8 fx;
4 }
```

Listing 5.12: Erweiterte Spezifikation für Beispielbinärdatei

nicht vorhanden.

Um das Anhängen neuer Felder an bestehende Objekte zu illustrieren, wird ein weiteres Werkzeug mit einer erweiterten Spezifikation erzeugt. Es wird das Feld `farbe` vom Typ `string` eingeführt (siehe Listing 5.12). Das neue Werkzeug liest die erste Beispieldatei und ordnet dem ersten Knoten die Farbe `rot` und dem zweiten `braun` zu. Das Ergebnis ist in Abb. 5.12 dargestellt.




*Es entstehen keine neuen Instanzen

Abbildung 5.12.: Beispiel einer .sf-Datei, welche ein neues Feld anhängt. Die Interpretation der Werte ist blau dargestellt. Die Bedeutung des Werts aus Sicht des Formats ist grau dargestellt.

### 5.2.3. Typordnung

Um den Aufbau des Formats verstehen zu können, ist es zunächst sinnvoll, sich mit der in SKilL eingeführten Typordnung zu beschäftigen. Dabei werden Typen zunächst in topologischer Ordnung sortiert, sprich Typen stehen vor ihren Subtypen. Diese Eigenschaft ist für die Prüfbarkeit der Gültigkeit von Supertypdeklarationen in einer Typdefinition bereits ausreichend.

Tatsächlich besitzt eine topologische Ordnung Freiheitsgrade, welche zusätzlich für sekundäre Ziele benutzt werden können. Verwendet man als nachgelagerte Ordnung eine lexikalische Ordnung, so erhält man für jede IR-Spezifikation eine totale Ordnung. Dies zu spezifizieren hat den Vorteil,

dass es wahrscheinlicher wird, dass gleiche Graphen in gleichen `.sf`-Dateien resultieren. Diese Eigenschaft ist besonders beim späteren Testen ungemein wertvoll. Außerdem ändert eine Umsortierung der Typdefinition in der IR-Spezifikation nichts an den entstehenden Anbindungen[1] und damit auch nichts an den entstehenden `.sf`-Dateien. Verwendet man verschiedene partielle IR-Spezifikationen in einer Werkzeugkette und hängt neue Typdefinitionen an bestehende `.sf`-Dateien an, so geht zwar die lexikalische Ordnung verloren, nicht aber die topologische.[2] Ebenso bleiben uns die übrigen Vorteile erhalten. Hier ist jedoch wichtig anzumerken, dass man sich auf ein lexikalisch sortiertes Vorkommen spezifizierter Typen in einer gelesenen `.sf`-Datei nicht verlassen kann, weil Typen erst in späteren Blöcken eingeführt werden können.[3]

### 5.2.4. Identifikatoren

Alle in einer `.sf`-Datei gespeicherten Daten sind eindeutig über ihnen implizit zugeordnete Identifikatoren (IDs) benannt. Bei diesen IDs handelt es sich um natürliche Zahlen. Da der Bezug auf gespeicherte Informationen über diese IDs erfolgt und kleine Zahlen effizienter dargestellt werden können als große, werden möglichst viele separate Namensräume für IDs, welche diesen eine Bedeutung zuordnen, verwendet. Zunächst bilden Typen einen solchen Namensraum. Typen erhalten IDs in Reihenfolge ihres Auftretens in der `.sf`-Datei. Da es bereits vordefinierte Typen gibt, beginnt die erste neue Typ-ID bei 32. Typ-IDs zwischen 0 und 31 werden für vordefinierte Typen verwendet. Typ-IDs dienen primär der Typisierung von Feldern. Um den Wertebereich für vordefinierte Typen verschobene Typ-IDs werden zudem zur Identifikation des Supertyps verwendet. Typen werden, wie auch Objekte, in einem Array aus Zeigern auf Typen verwaltet. Zieht man 32 von der Typ-ID ab, so erhält man den Index des Typs in diesem Array. Die Supertyp-IDs sind nur um 31 verschoben, damit der Wert 0 für Typen ohne Supertyp

[1] Ebenso trägt eine alphabetische Sortierung von Feldern zur API-Stabilität bei.
[2] Es ist nicht möglich bestehende Supertyp-Beziehungen zu ändern.
[3] Beispiel: Block 1: A, C <: A. Block 2: B <: A. In Block 3 würde B hinter C auftreten, da es in der Datei später definiert wurde.

verwendet werden kann. Hierdurch wird die Spezialisierung vordefinierter Typen auf Formatebene verhindert, da nur Typ-IDs in Typdeskriptoren im Binärformat verwendet werden können, die sich auf andere Typdeskriptoren und damit benutzerdefinierte Typen beziehen.

Für jeden Typ erhalten dessen Felder einen Namensraum. Dieser beginnt bei 1. Feld-IDs werden im Binärformat derzeit nur zum Anhängen neuer Daten verwendet.[1]

Um Objekten eine eindeutige ID zuzuordnen, werden diese aus Sicht ihres Basistyps durchnummeriert (siehe Abb. 5.5). Um Nullzeiger effizient darzustellen, beginnen Objekt-IDs bei 1, d.h. sie sind gegenüber dem Index im Verwaltungsarray um 1 verschoben. Negative Objekt-IDs können in der Repräsentation im Hauptspeicher eine Rolle spielen, werden hier aber nicht weiter betrachtet.

Ein Objekt eines beliebigen Typs ist damit in einer `.sf`-Datei eindeutig durch Kombination von Typ-ID und Objekt-ID identifizierbar.

Die Vergabe neuer IDs erfolgt ab der ersten ID stets lückenlos. Dies ist nicht nur effizient, sondern macht zulässige IDs auch leicht prüfbar, vorausgesetzt man kennt die Größe des Namensraums. Wenn das Objektverwaltungsarray nicht über das API der Anbindung exportiert wird, ist die Größe des Namensraums bei jedem Zugriff bekannt.

IDs bestehender Entitäten werden von Anhängeoperationen nicht verändert. Dies ist zwingend erforderlich, da man die für die Vergabe verantwortlichen Daten nicht modifiziert. Dagegen erhält eine Schreiboperation bestehende IDs im Allgemeinen nicht. Daher ist es für Werkzeugbauer nicht ratsam, sich auf Objekt-ID zu beziehen, selbst wenn ein API diese IDs zur Verfügung stellen würde. Umgekehrt gilt für Schreiboperationen, dass sie Objekte in Typordnung sortieren. Da diese Sortierung jeweils für den geschriebenen Block erfolgt, kann man sich im Kontext von Anhängeoperationen im Allgemeinen nicht auf eine Sortierung verlassen.[2]

[1] D.h. wenn neue Instanzen eines bestehenden Typs mit mindestens einem Feld an eine Datei angehängt werden. Dies erspart eine mehrfache Beschreibung desselben Feldes, was Platz spart und eine Fehlerquelle eliminiert.

[2] Beispiel: Block 1: S, A <: S, C <: S. Block 2: B <: A. In Block 3 dürfte B vor C auftreten, da es in der Datei später definiert wurde, aber ein Subtyp des früher definierten A ist.

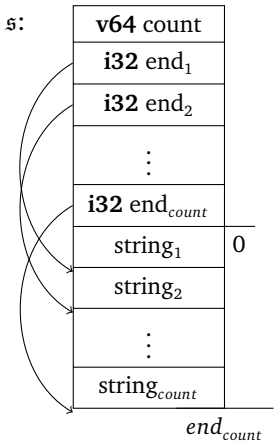



Abbildung 5.13.: Schematische Darstellung eines String-Blocks

Darüber hinaus können benutzerdefinierte Typen und Felder innerhalb einer Typdefinition eindeutig über ihre in Kleinbuchstaben konvertierten Namen identifiziert werden. Dies ist für den Abgleich von Werkzeugspezifikation und Dateispezifikation beim Lesen einer `.sf`-Datei erforderlich.

### 5.2.5. Der String-Block

Nachdem die grobe Struktur des Formats beschrieben wurde, werden nun die einzelnen Komponenten im Detail festgelegt. Für konkrete Datentypen werden hier die in SKilL gebräuchlichen Typnamen, also v64 für variabel lange 64bit Ganzzahlen usw., verwendet.

Der String-Block ist in Abb. 5.13 schematisch dargestellt. Er beginnt mit einem v64 *count*, welcher die Zahl der Strings in diesem Block speichert. Er wird gefolgt von *count* i32 Werten, welche die Endpositionen der entsprechenden Strings speichern. Diesen folgen die Zeichen der einzelnen Strings. Danach ist das Ende des String-Blocks erreicht. War *count* gleich 0, so ist das Ende des String-Blocks direkt danach erreicht.

Es werden *count* neue Strings eingeführt. Die Positionen der Zeichen dieser

Strings ergeben sich aus der Endposition und der Endposition des Vorgängers, falls es in diesem Block einen gibt. Sonst ist es der erste String des Blocks und die Anfangsposition ist 0. Da Strings wie Zeigertypen verwendet werden, beginnen ihre IDs bei 1. Über IDs können Strings beim Lesen mithilfe eines Arrays in Zeiger auf Strings konvertiert werden, indem man in diesem Array die String-Instanzen referenziert, die man aus der Datei gelesen hat. Die Position im Array entspricht dabei dem Vorkommen des Strings in der `.sf`-Datei. Die Verwendung der ID 0 entspricht dem Nullzeiger. Strings sind utf-8[Yer03] codiert.

Die Interpretation von Anzahl und Offsets erfolgt vorzeichenlos, um den zulässigen Wertebereich zu verdoppeln. Die Wahl von i32 als Datentyp ermöglicht es, sowohl die Offsets als auch die Daten zunächst zu überspringen, da man mit `count` zum letzten offset und von dort zum Ende des Stringblocks kommt. Die Beschränkung auf Zeichenketten mit höchstens grob 4 Milliarden Zeichen erscheint unerheblich angesichts des Vorteils, einen größeren Datenblock zunächst überspringen zu können. Ebenso ist es beim Schreiben in Verbindung mit *mmap* (siehe [POS16] S. 1338ff) möglich, den Endpositionsbereich an eine Speicheradresse zu binden und ihn mit Werten zu befüllen, während man die einzelnen Strings in die `.sf`-Datei schreibt. Hierdurch spart man eine Iteration über alle zu schreibenden Strings.

### 5.2.6. Typ- und Felddeskriptoren

Der Typblock ist in Abb. 5.14 schematisch dargestellt. Er beginnt mit einem v64 *count*, welcher die Zahl der Typdeskriptoren in diesem Block speichert. Es folgen *count* Typdeskriptoren. Ein solcher beginnt mit der Referenz auf den Namen des Typs, sowie der Zahl der Objekte, die in diesem Block zum Typ hinzugefügt werden.

Aus Name und Objektzahl wird beim Lesen kontextsensitiv entschieden, um welchen Typ es sich handelt und wie dieser verändert wird. Handelt es sich um einen Typ, der in der aktuellen `.sf`-Datei noch nicht vorgekommen ist, so folgt zunächst die Typbeschreibung. Ist der Typ bereits bekannt, so ist diese nicht mehr erforderlich und folglich auch nicht präsent (Felder mit

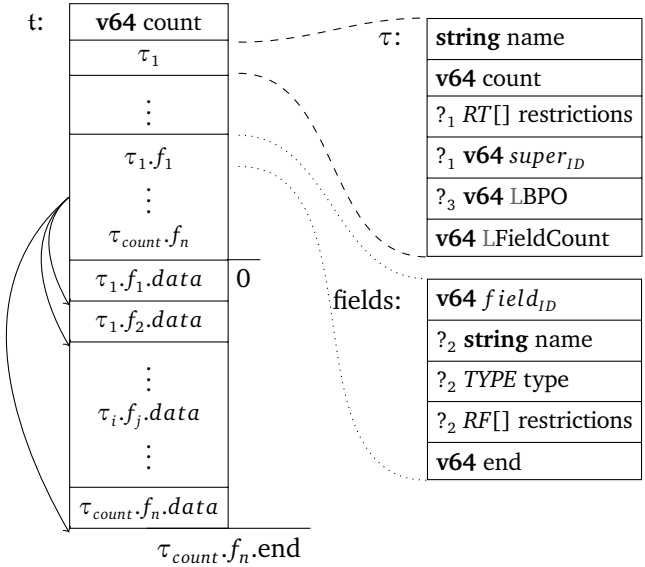



Abbildung 5.14.: Schematische Darstellung eines Typblocks. Daten mit $?_1$ und $?_2$ sind nur beim ersten Vorkommen präsent. Daten mit $?_3$ sind nur präsent, falls $count \neq 0 \wedge super_{ID} \neq 0$.

$?_1$). Dies betrifft Typrestriktionen, auf die später noch genauer eingegangen werden wird (siehe §5.2.7). Ferner ist der danach folgende logische Offset des Supertyps betroffen. Dieser ist 0, falls es keinen Supertyp gibt und sonst die Supertyp-ID - 31, sprich vom ersten deklarierten Typ erbt man mit der $super_{ID}$ 1.

Falls der gelesene Typ einen Supertyp hat und die lokale Objektzahl (`count`) ungleich 0 ist, folgt der *LBPO*.[1] Hierbei handelt es sich um die Verschiebung der lokal, d.h. in diesem Typblock, hinzugefügten Instanzen relativ zum Basistyp. Aus diesem Wert kann man unter Hinzunahme der

[1] Local Base Pool Offset. Falls eine Datei aus nur einem Block besteht, gilt BPO = LBPO. Falls es mehrere Blöcke gibt, so erhält man aus dem LBPO den BPO, indem man die Zahl der in vorherigen Blöcken bestehenden Instanzen des Basistyps aufaddiert. Ein Typ hält für jeden Block, in dem er Instanzen hat, einen BPO.

$$[\![t]\!]_{type} = \begin{cases} [\![\texttt{Typ-ID}(t)]\!]_{v64} \circ [\![val(t)]\!]_t & \texttt{Typ-ID}(t) \in [0,4] \\ [\![\texttt{Typ-ID}(t)]\!]_{v64} & \texttt{Typ-ID}(t) \in [5,14] \\ \texttt{0x0F} \circ [\![i]\!]_{v64} \circ [\![T]\!]_{type} & t = T[i] \\ \texttt{0x11} \circ [\![T]\!]_{type} & t = T[] \\ \texttt{0x12} \circ [\![T]\!]_{type} & t = list\langle T \rangle \\ \texttt{0x13} \circ [\![T]\!]_{type} & t = set\langle T \rangle \\ \texttt{0x14} \circ [\![K]\!]_{type} \circ [\![V]\!]_{type} & t = map\langle K, V \rangle \\ [\![\texttt{Typ-ID}(t)]\!]_{v64} & t \in \mathcal{U} \end{cases}$$

Abbildung 5.15.: Definition von $[\![\_]\!]_{type}$ gemäß [Fel17] §7.4. Vordefinierte Typ-IDs sind [Fel17] §G zu entnehmen, sofern nicht ersichtlich. Die IDs 0 bis 4 gehören zu Integer-Konstanten.

Objektzahl den Indexbereich im aktuellen Block berechnen. Verschiebt man diesen um den Startindex des Basistyps im aktuellen Block, so erhält man den absoluten Indexbereich der hierdurch implizit angelegten Objekte. Die tatsächlichen Typen dieser Objekte lassen sich erst bestimmen, wenn man den Block zu Ende gelesen hat, da Subtypen folgen könnten. Der Typdeskriptor endet immer mit der Zahl der lokal, d.h. in diesem Typblock, definierten Felder. Diese sind immer vorhanden, da man einen bestehenden Typ, ohne diesem neue Instanzen hinzuzufügen, um Felder erweitern kann. Diese Zahl wird zunächst mit einem Verweis auf den Typ in einer Liste abgelegt, um nach den Typdeskriptoren die entsprechenden Felddeskriptoren lesen und zuordnen zu können.

Ein Felddeskriptor wird entsprechend dieser Liste erwartet und interpretiert. Er beginnt mit der Feld-ID. Gehört die ID zu einem noch unbekannten Feld,[1] so folgen Name und Typ des Feldes sowie Feldrestriktionen. Der Name ist eine String-Referenz. Der Typ wird mit der in Abb. 5.15 beschriebenen Funktion $[\![\_]\!]_{type}$ dargestellt. Diese verwendet im Wesentlichen die Typ-ID und ist für Containertypen rekursiv auf Typparametern. Feldrestriktionen werden später zusammen mit den Typrestriktionen genauer betrachtet. Es folgt immer die Endposition der Felddaten.

Hat man alle Felddeskriptoren gelesen, so ergeben sich die Bereiche der

[1] Es ist dann zwingend erforderlich, dass die ID exakt $1 + \#bekannteFelder$ ist.

Felddaten aus den Endpositionen der Felder. Diese sind relativ zur dann aktuellen Position, sprich logische Indices innerhalb des Datenblocks. Der Typblock endet an der Endposition des letzten Feldes.

Von Typ-IDs abgesehen sind alle auftretenden IDs und Offsets im jeweiligen Kontext monoton wachsend.[1] Aus dieser Eigenschaft kann man eine günstige Konsistenzprüfung ableiten. Diese Eigenschaft verursacht beim Erzeugen einer `.sf`-Datei keine Mehrkosten. Typ-IDs sind im Allgemeinen nur innerhalb des ersten Blocks monoton. Eine Verletzung der Monotonie ist möglich, indem man in weiteren Blöcken Subtypen einfügt und danach in einem Block nur einige davon erweitert. Da Typdeskriptoren in Typordnung auftreten und diese leicht durch rekursiven Abstieg zu etablieren ist, kann man einen später angehängten Subtyp vor einem im ersten Block definierten Typ auf gleicher Vererbungsebene beobachten. Dieser Umstand scheint im ersten Moment der topologischen Ordnung durch Typ-IDs zu widersprechen. Er ist aber darin begründet, dass topologische Ordnungen einen Freiheitsgrad in der Besuchsreihenfolge von Knoten auf gleicher Ebene haben.[2]

#### 5.2.6.1. Repräsentation von Referenzen

Referenzen im Binärformat werden über die Objekt-ID, also eine Zahl, dargestellt. Da in den meisten Fällen der Typ des referenzierten Objekts statisch bekannt ist, wurde entschieden, IDs pro Typ zu vergeben, sodass die zur Serialisierung verwendete Variable length 64-bit signed integer (v64)-Kodierung in der Praxis möglichst kurz ausfällt. Ist der Typ statisch nicht bekannt, sprich es handelt sich um den SKilL-Typ `annotation`, so wird die Typ-ID der Objekt-ID vorangestellt.[3]

Um effizient ein Objekt in eine ID zu konvertieren, wird diese zur Laufzeit

[1] Tatsächlich nicht streng monoton, da keine neuen Daten, Instanzen etc. entstehen müssen.

[2] Dieser wird für `.sf`-Dateien, welche nur aus einem Block bestehen durch die Hinzunahme der lexikalischen Ordnung beseitigt. Diese Beseitigung hat allerdings den Effekt, dass man bei der Betrachtung von Testdaten zunächst den Eindruck gewinnen kann, dass Typ-IDs monoton wären, was bei einer Prüfung zu falschen Fehlermeldungen führt.

[3] Die Typisierung ist dem Felddeskriptor zu entnehmen. Sie gilt für alle Felddaten.

des Werkzeugs im Objekt gespeichert. Da die Spezifikationssprache Vererbung enthält und man nicht pro geerbtem Typ eine ID speichern will, werden IDs nur für Basistypen, d.h. Typen ohne eigenen Supertyp, vergeben. Um effizient eine ID in ein Objekt zu konvertieren, wäre es hilfreich, wenn es sich dabei im Wesentlichen um einen Arrayindex handelt. Tatsächlich muss man einen `null`-Zeiger effizient darstellen können, sodass 0 für `null` verwendet wird und alle IDs von 1 ab für echte Objekte.[1] Dadurch können beide Konvertierungen mit konstantem Zeit und Speicherbedarf durchgeführt werden. Die erste Funktion in Abb. 5.16 beschreibt die Serialisierung. Die Auflösung ID nach Objekt bei der Deserialisierung erfolgt mittels eines Arrays auf die Instanzen des entsprechenden Objekt-Typs. Eine Typprüfung nach der Auflösung der ID in eine Referenz kann in konstanter Zeit etwa mithilfe von Typbereichsprüfung [SPT83] erfolgen. In der Praxis wird, soweit möglich, auf eine Typprüfung durch das Zielsprachentypsystem zurückgegriffen.

#### 5.2.6.2. Repräsentation anderer Felddaten

Eine vollständige Beschreibung der Übersetzungsfunktion für Felddaten findet sich in Abb. 5.16.[2] Im Wesentlichen werden Werte der verbleibenden Datentypen einfach in die `.sf`-Datei kopiert. Da Plattformunabhängigkeit gefordert wird (siehe §1.1.3), muss die Endianness festgelegt werden. Die Festlegung selbst ist im Wesentlichen willkürlich. Die Wahl hierfür fiel auf *network byteorder* [RP94]. Dies hat die Konsequenz, dass man bei Entwicklung auf Intel Prozessoren gezwungen wird, sich mit dem Thema zu befassen. Da es sich beim Gros der Daten erwartungsgemäß um Referenzen handelt, welche über v64 abgebildet werden, ist die Entscheidung für die Gesamtperformanz nicht ausschlaggebend. Container werden als Sequenzen ihrer Elemente gespeichert, denen die Zahl der Elemente vorangestellt ist.

[1] Enthält ein Feld einen Zeiger auf einen Typ $T$, so kann eine Referenz, die durch die ID $i$ serialisiert wurde, mit folgendem Ausdruck `i==0?null:T.data[i-1]` aufgelöst werden. Werden Referenzen im Block deserialisiert, so kann die Dereferenzierung `T.data` vor die Schleife geschoben werden.

[2] Die Serialisierung von Maps ist bewusst für alle Typen definiert, obwohl das nicht erforderlich wäre, da entsprechende Typen in `.skill`-Spezifikationen nicht zulässig sind.

$$\forall t \in \mathcal{U} \cup \{\textbf{string}\}.[\![f]\!]_t = \begin{cases} \texttt{0x00}, & f = \texttt{NULL} \\ [\![\texttt{Objekt-ID}(f)]\!]_{v64} & else \end{cases}$$

$$[\![f]\!]_{\textbf{annotation}} = \begin{cases} \texttt{0x00 0x00}, & f = \texttt{NULL} \\ [\![\texttt{Typ-ID}(t_f) - 31]\!]_{v64} \circ [\![\texttt{Objekt-ID}(f)]\!]_{v64} & else \end{cases}$$

$[\![\top]\!]_{\textbf{bool}} = [\![\varepsilon x.x \neq 0]\!]_{i8}$

$[\![\bot]\!]_{\textbf{bool}} = \texttt{0x00}$

$\forall t \in \mathcal{I} \setminus \{\textbf{v64}\}.[\![f]\!]_t = f$

$[\![f]\!]_{\textbf{v64}} =$ siehe [Fel17] §C.

$[\![f]\!]_{\textbf{f32}} = [\![f]\!]_{\textbf{i32}}$, $[\![f]\!]_{\textbf{f64}} = [\![f]\!]_{\textbf{i64}}$

$$\forall g \in \mathcal{T}, n \in \mathbb{N}^+.t = g\,[n] \implies [\![f]\!]_t = \bigcirc_{i=0}^{n-1} [\![f_i]\!]_g$$

$$\forall g \in \mathcal{B}, n = size(f), t \in \{g\,[\,], \texttt{set<}g\texttt{>}, \texttt{list<}g\texttt{>}\}.[\![f]\!]_t = [\![n]\!]_{v64} \bigcirc_{i=0}^{n-1} [\![f_i]\!]_g$$

$$\forall r,s,t \in \mathcal{T}.[\![\_]\!]_{map<r,s,t>} = [\![\_]\!]_{map<r,map<s,t>>}$$

$$\forall k,v \in \mathcal{T}, n = size(f), t = \texttt{map<}k,v\texttt{>}.[\![f]\!]_t = [\![n]\!]_{v64} \bigcirc_{i=0}^{n-1} [\![f.k_i]\!]_k [\![f\,[k_i]]\!]_v$$

Abbildung 5.16.: Übersetzungsfunktionen für Feldwerte nach [Fel17] §7.4. $\mathcal{U}$ ist die Menge nutzerdefinierter Typen; $\mathcal{I}$ ist die Menge der Integer-Typen; $\mathcal{B}$ ist die Menge der Nicht-Container-Typen; $\mathcal{T}$ ist die Menge aller Typen.

Wahrheitswerte werden über ein Byte dargestellt, welches für falsch 0 ist und sonst als wahr interpretiert wird. Hierdurch kann in gängigen Darstellungen der Werte beim Schreiben einfach kopiert werden. Auf eine Bit-Vektor-Darstellung wurde verzichtet, da die kleinste Einheit ein Byte[1] ist.

#### 5.2.6.3. Konsequenzen des Feldlayouts

Da das serialisierte Typsystem den eigentlichen Daten vorangestellt ist, wird die Forderung nach einem spezifikationsunabhängigen Objektlayout[2] schon dadurch erfüllt, dass das Objektlayout ausschließlich durch das serialisierte

[1] Historisch motiviert ist dies neben der etwas einfacheren Implementierung auch durch den Umstand, dass zum Veröffentlichungszeitpunkt der ersten SKilL-Sprachspezifikation Lua [IdFC15a] keine Bit-Operationen unterstützte, eine Anbindung an Lua davon abgesehen aber denkbar schien. Diese wurden in Lua 5.3 in die Programmiersprache integriert [IdFC15b].

[2] Siehe Forderung §1.1.12

Typsystem bestimmt ist. Verwendet man variabel lange Kodierungen für Daten, um der Forderung nach einer kompakten Darstellung[1] gerecht zu werden, so wird die Erfüllung der Forderungen nach bedarfsorientiertem Lesen[2] zunächst kritisch. Weil Felddaten nach Feld gruppiert in einem Block gespeichert werden und deren Position günstig berechnet werden kann, können diese günstig übersprungen werden. Würde man hingegen die Felddaten nach ihren besitzenden Objekten gruppieren, so müsste man in jedem Objekt über einzelne Felddaten springen, deren Länge eventuell nicht bekannt wäre. Man hat also die Wahl zwischen einem Sprung insgesamt und jeweils einem Sprung pro Instanz.

Die Aufteilung der Daten nach Feld hat zudem den Vorteil, dass es offensichtlich ist, wie man neue Felder an bestehende Objekte anhängt,[3] ohne bestehende Objekte zu manipulieren. Hängt man in einem weiteren Block am Ende der `.sf`-Datei diese Daten an, so bekommt ein Objekt bei der Deserialisierung automatisch die richtigen Werte.

Ferner haben Objekte durch diese Darstellung keine expliziten Eigenschaften mehr. So konnte erreicht werden, dass Objekte im Binärformat nur noch implizit existieren. Zudem hat dies auf realen Daten zur Folge, dass `.sf`-Dateien mit allgemeinen Kompressionsverfahren effizient komprimiert werden können, sodass SKilL auch hier im Vergleich gut abschneidet (siehe §7.8.2).

#### 5.2.6.4. Abgleich von Datei- und Werkzeugspezifikation

In dieser Arbeit verstehen wir unter Änderungstoleranz die Fähigkeit, automatisch Änderungen zwischen tatsächlicher und erwarteter Typ- bzw. Feldstruktur zu erkennen und auf diese gemäß SKilL-Sprachspezifikation zu reagieren. Insbesondere ist es möglich, qualifiziert auf die Unterschiede hinzuweisen.

Die in SKilL zentrale Änderungstoleranz wird im Wesentlichen durch den

[1] Siehe Forderung §1.1.8
[2] Siehe Forderung §1.1.10
[3] Siehe Forderung §1.1.11

Abgleich von Datei- und Werkzeugspezifikation beim Lesen einer `.sf`-Datei realisiert. Die dabei angewendeten Regeln sind denkbar einfach. Existiert eine Typdefinition nur in einer Spezifikation, so wird sie beim Lesen ins Laufzeittypsystem übernommen. Existiert eine Felddefinition nur in einer Spezifikation, so wird sie ebenso beim Lesen ins Laufzeittypsystem übernommen. Bedeutet dies, dass aus einer Datei gelesene Instanzen um ein Feld erweitert werden, wird dieses mit Standardwerten gefüllt. Diese Werte entsprechen genulltem Speicher, was den Konventionen von Java und C++ entspricht.

Widersprechen sich zwei gleich benannte Typdefinitionen in der Wahl des Supertypen, so ist das ein Fehler, der nicht automatisch bearbeitet wird. In dieser Situation wird das Einlesen einer `.sf`-Datei also verhindert. Eine automatische Adaptierung einer veränderten Typhierarchie[1] würde eine Typprüfung betroffener Zeigerwerte erfordern. Zudem ist eine solche Veränderung ein deutliches Anzeichen einer groben Restrukturierung der IR.

Widersprechen sich zwei Felddefinitionen eines Typs, so ist ein Fehler zu erzeugen und die `.sf`-Datei ist nicht zu lesen. Eine automatische Behandlung ist in manchen Fällen denkbar und in [Fel17] teilweise spezifiziert, wurde aber nie implementiert. Im Fall einer automatischen Behandlung wäre hier ein Anhängen an eine `.sf`-Datei nicht mehr zulässig, weswegen dieser Weg nicht ernsthaft weiter verfolgt wurde.

#### 5.2.6.5. Erweiterbarkeit des Typsystems

Das Typsystem ist so gestaltet, dass Typen über IDs serialisiert werden. Dabei enthält der Wertebereich Löcher (siehe [Fel17] §G),[2] die in Zukunft für Erweiterungen des Typsystems verwendet werden können, welche alte Daten intakt lassen. Die hierfür verwendbaren IDs sind historisch bedingt nicht zusammenhängend.

Da neue IDs alte Implementierungen unbrauchbar machen würden sobald diese tatsächlich verwendet werden, kann man hierüber auch neue

[1] Im Gegensatz zu einer unvollständigen, wie es bei fehlenden Typen der Fall ist.
[2] Die Instruktionscodes in Java-Bytecode [LY99] sind ähnlich gestaltet.

Strukturen in das Datenformat einführen. Man könnte z.B. einen Container-by-Reference Typ einführen. Sollte dieser tatsächlich in einer `.sf`-Datei Verwendung finden, könnte man einen weiteren Abschnitt zwischen Felddaten und Felddeklarationen einführen, der alle notwendigen strukturellen Informationen zu diesen Referenzen enthält. Dies würde das alte Format nicht zerstören, da eine zulässige Implementierung bereits bei der neuen ID einen Fehler meldet. Eine Implementierung, die diese ID kennt, verarbeitet auch die entsprechende Anpassung am Binärformat.

Die Zahl der auf diesem Wege einführbaren Erweiterungen ist unbeschränkt, da bei Vergabe einer ID über Typargumente, ähnlich wie bei Containern, neue freie IDs geschaffen werden können.

### 5.2.7. Erweiterungen durch Restrictions

Wenn man sich mit einer in Ada implementierten IR befasst, ist es naheliegend, nicht-null Referenzen und Bereichstypen in SKilL zu integrieren. Um diese Eigenschaft in einer `.sf`-Datei ablegen zu können, ohne sie direkt zu einem Teil des SKilL-Typsystems zu machen, wurden sogenannte Restrictions eingeführt (siehe §5.1.6). Dabei handelte es sich konzeptionell um Einschränkungen bestehender Typen, wie beispielsweise das Entfernen des Nullzeigers aus dem Wertebereich eines Feldes. Da diese Einschränkungen sowohl in der Deklaration als auch der Verwendung eines Typs zulässig sein sollen, gibt es sie in Form von Typ- und Feldrestriktionen. Das Konzept wurde verallgemeinert und ist mit Java Annotations (siehe [GJS+14] §9.7) vergleichbar.

Die SKilL-Sprachspezifikation [Fel17] ist dabei vor allem eine Wunschliste. Bei der Implementierung zeigte sich relativ schnell, dass das Ignorieren von Restrictions sehr unterschiedlichen Konsequenzen für den tatsächlichen Einsatz hat. Daher wurde beschlossen, Restrictions in zwei Kategorien zu unterteilen und die Behandlung mancher Restrictions optional zu handhaben, solange man darüber klar informiert.

Eine optionale Restriktion ist beispielsweise `@nonnull`. Diese in der Implementierung zu ignorieren hat keine Konsequenzen, sofern sich der Werk-

zeugbauer an die Semantik hält. Nicht optional dagegen ist beispielsweise eine Veränderung der Feldkodierung durch `@coding`. Hierbei handelt es sich um eine sehr spezielle Form von Restrictions, da sich hierdurch der Wertebereich von Daten nicht ändert. Das Binärformat ist so konstruiert, dass sich die Konsequenzen von `@coding` nur auf die Datensegmente des Feldes erstrecken. Das bedeutet zwar, dass eine Implementierung `@coding` erkennen muss, um beim Lesen der Felddaten nicht fälschlicherweise einen Fehler zu produzieren. Es ist aber dennoch möglich die Felddaten zu überspringen, wenn man die spezielle Kodierung nicht kennt. Hierdurch ergibt sich prinzipiell der Vorteil der Kombinierbarkeit von bedarfsorientiertem Lesen und Dateikompression. Eine Evaluation dieser Fähigkeiten hat sich mangels Zeit und Dringlichkeit nicht ergeben.[1]

Restriktionen werden von der SKilL-Spezifikation eine ID zugeordnet, über die sie in der `.sf`-Datei identifizierbar sind. Jede einzelne Restriktion definiert eine Serialisierungsfunktion. Diese ist in vielen Fällen leer. Ein Beispiel für eine nichtleere Serialisierungsfunktion ist `@range`. Hier werden zwei Werte gespeichert, die den zulässigen Wertebereich einschränken.

Eine effiziente Umsetzung von Restrictions in einer Anbindung ist relativ aufwändig. Daher können alle hier evaluierten Implementierungen `.sf`-Dateien, die diese enthalten, zwar lesen, die Semantik der Restrictions wird allerdings ignoriert. Um die in §7 durchgeführten Vergleiche fair zu gestalten, wurde, insbesondere weil die verglichenen Formate keine vergleichbare Funktionalität aufweisen, der Code zum Serialisieren von Restrictions entfernt. D.h. die in Abb. 5.14 dargestellten *restrictions*-Einträge werden durch ein 0-Byte, und damit durch ein leeres Array, geschrieben. Die Serialisierungsfunktionen für Restrictions sind in [Fel17] §G übersichtlich dargestellt.

### 5.2.8. Skalierbarkeit – Numerische Grenzen

Hätte man sich auf konstant breite Indices versteift, so müsste man sich entscheiden, ob man nur $2^{32} - 1$ Instanzen erlaubt oder ob man nahezu

[1] Eine Architektur zu entwickeln, die die bedarfsorientierte Allokation von Objekten ermöglicht, sollte zunächst den größeren Mehrwert darstellen.

doppelt so große Austauschdateien verwendet. Da die v64-Kompression effizient ist, kann auf echte numerische Grenzen weitgehend verzichtet werden. Eine Austauschdatei mit über $2^{32}$ Typen erscheint wenig praxisnah. Zu betonen ist aber, dass es hier aber nicht an SKilL scheitern würde. Würde man eine `.sf`-Datei mit $2^{64}-32$ Typen und je $2^{64}-1$ Instanzen erstellen, hätte man mit allen gängigen Dateisystemen Probleme. Die Kosten für diese Grenzen sind dabei marginal.

Tatsächlich stellt man bei der Implementierung fest, dass die Präsenz von signed 32bit Integern als Indextypen[1] zu wesentlich kleineren praktikablen Limits führt. Daher wird in [Fel17] Anhang F die Garantie auf $2^{30}$ Instanzen pro Basistyp beschränkt. Die größten in dieser Arbeit verwendeten Testdaten reizen das zu 2% aus. Würde man dieses Limit überschreiten, wäre voraussichtlich ein Eingriff in den generierten Code erforderlich, der sich nicht im API äußert.[2] Messungen zeigen, dass auf heute gängigen Systemen der Hauptspeicher vor Erreichen dieses Limits ausgeht. Der Punkt an dem man Graphen bearbeiten möchte, die die SKilL inhärenten Limits von $2^{64}-1$ Instanzen pro Basistyp überschreiten, liegt weit hinter einem Punkt an dem man von 64bit auf 128bit breite Speicherbusse umgestiegen ist.

### 5.2.9. Formalisierung

Eine möglichst einfache formale Spezifikation des Binärformats ist erforderlich, um sicherzustellen, dass von Studenten angefertigte Anbindungen tatsächlich mit anderen Anbindungen kombinierbar sind. Die Spezifikation der Repräsentation von Felddaten in Form von Übersetzungsfunktionen ist zudem kompakt und bildet alle Fälle auf einer DIN-A4-Seite ab. Für die Spezifikation des Gesamtformats wurde zunächst auf Text und veranschaulichende Bilder zurückgegriffen. Auch hier lässt sich eine formale Spezifikation leicht formulieren (siehe [Fel17] Anhang B).

[1] Dieses Problem wird von Arrayzugriffen auf der JVM verursacht, siehe [LY99].

[2] Dieser Punkt wird tatsächlich von Ada- und C++-Implementierungen genutzt um ObjektIDs in 32bit Integern zu speichern.

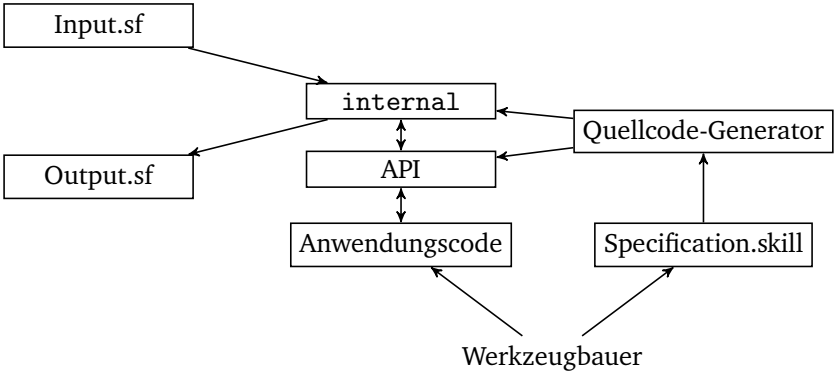



Abbildung 5.17.: Informationsfluss aus Sicht des Werkzeugbauers.

## 5.3. Werkzeugintegration

Die Integration von SKilL in ein Werkzeug ist nur insofern spezifiziert, als `.sf`-Dateien über ein generiertes API manipuliert werden. Aus Sicht des Werkzeugbauers gibt es neben dem Anwendungscode nur das API, dessen Implementierung, die Werkzeugspezifikation, sowie den Quellcode-Generator (siehe Abb. 5.17). Die Absicht ist, dass sich der Werkzeugbauer nur mit der IR-Spezifikation und dem Anwendungscode befassen muss. Die Integration des generierten Codes in den Erstellungsprozess (englisch build process) ist üblicherweise nahezu aufwandsfrei. Die eigentliche Quellcode-Generierung lässt sich zwar in den Erstellungsprozess integrieren. Dies wird hier aber offen gelassen, da es erfahrungsgemäß weder erforderlich noch hilfreich ist, weil die Änderungstoleranz Anpassungen automatisch abfängt.

Der Quellcode-Generator liest eine IR-Spezifikation und generiert daraus mithilfe einiger weiterer Parameter eine Anbindung, d.h. ein API und dessen Implementierung. Weitere Parameter sind beispielsweise Paket- oder Modulnamen, die für erzeugte Datentypen verwendet werden, sowie der Inhalt von Kommentaren im Kopf der generierten Quellen. Einige Quellcode-Generator-Back-Ends erlauben zum Generierungszeitpunkt zu entscheiden,

ob man Objekt-IDs über das generierte API sehen kann. Ferner können einige Quellcode-Generator-Back-Ends einen Besucher (Visitor) [GHJV95] über die spezifizierten Typen generieren. Hierbei handelt es sich lediglich um Beispiele, eine Anbindung auszugestalten. Definiert ist ein Besucher aus Sicht von SKilL nicht.

Die Ausgestaltung des APIs ist wie auch der Quellcode-Generator nicht direkt spezifiziert. Eine strikte Festlegung durch die SKilL-Sprachspezifikation verbietet sich schon aufgrund der Sprachunabhängigkeit. Würde man ein konzeptionell objektorientiertes API spezifizieren, wäre dies in rein prozeduralen oder funktionalen Programmiersprachen kaum intuitiv integrierbar. Ebenso zeigt sich, dass sich die APIs in Programmiersprachen mit automatischer Speicherverwaltung von denen in Programmiersprachen mit manueller Speicherverwaltung teilweise unterscheiden. Daher wird in diesem Abschnitt die Intention des APIs anhand der grundlegenden Struktur moderner SKilL-APIs erläutert. Die hier verwendete objektorientierte Formulierung lässt sich leicht auf nicht objektorientierte Programmiersprachen übertragen. Die in diesem Abschnitt erläuterte API-Architektur ist das Ergebnis eines langen Evolutionsprozesses, in dem Benutzbarkeit und geringe Ausführungszeit optimiert wurden.

Das API besteht aus zwei Einstiegspunkten. Der erste ist das Dateiverwaltungsobjekt, welches den Zustand einer `.sf`-Datei und somit auch deren Inhalt verwaltet. Der zweite Einstiegspunkt sind die Objekte selbst.

Befassen wir uns zunächst mit dem Dateiobjekt. Dieses kann erzeugt werden, indem man eine neue `.sf`-Datei öffnet. Dabei werden Bearbeitungsmodi und ein Pfad gesetzt. Diese Modi sind für die Instantiierung *erzeugen* oder *lesen*. Für schreibende Aktionen gibt es die Modi *schreiben*, *anhängen* und nicht schreibbar, also *nur lesen*. Pfade können im Nachhinein beliebig verändert werden. Der Instantiierungsmodus ist nur während der Instantiierung von Bedeutung. Der Schreibmodus kann verändert werden, falls dies von der Implementierung realisiert werden kann. So kann z.B. ein *nur lesen* nicht verändert werden, es kann aber jeder Modus dahin überführt werden. Es ist immer möglich, von *anhängen* nach *schreiben* zu wechseln. Der umgekehrte Weg ist unter Umständen nicht möglich, da es hier, insbesondere

in Kombination mit dem Wechseln des Zielpfades Implementierungsaufwand gibt, dessen Umfang sehr von den Mitteln der Standardbibiliothek der Zielsprache abhängt.

Das Dateiobjekt bietet als Hauptzugriffspunkt für jeden spezifizierten Typ einen Zugriff auf dessen Instanzen. Diese sind benannt und vom Dateiobjekt nur eine[1] Zeigerdereferenzierung entfernt. Um Reflection zu realisieren, bietet das Dateiobjekt zudem Iteratoren über alle verwalteten Typen. Da Strings in SKilL per Referenz verwaltet werden, existiert auch ein Zugriffsobjekt auf Strings. Dieses bearbeitet für den Werkzeugbauer hauptsächlich die Aufgabe, der `.sf`-Datei neue Strings hinzuzufügen, und unterscheidet sich in der Implementierung deutlich und in der API leicht von regulären Zugriffsobjekten.

Vom Dateiobjekt zur Verfügung gestellte Zugriffsobjekte für Strings und benutzerdefinierte Typen bieten Iteratoren über alle, dem Dateiobjekt zugehörigen, Instanzen des jeweiligen Typs. Unter den Iteratoren gibt es immer mindestens einen, der in $O(t)$ konstruiert werden kann und in $O(1)$ das nächste Element erreichen kann, bzw. prüfen kann, ob ein solches existiert. Dabei ist $t$ die Zahl der Subtypen des gewählten Typs.[2] Zudem gibt es eine Funktion, die einen Zugriff in O(1) über die ID eines Objekts gestattet, falls die IDs im API exportiert werden.

Daneben existieren Methoden zur Abfrage der Anzahl der Instanzen eines Typs, des Namen eines Typs und der Vererbungshierarchie. Ebenso existiert ein Iterator über die Felder eines Typs – unspezifizierte Felder mit eingeschlossen. Die dadurch zugreifbaren Feldrepräsentanten ermöglichen dem Werkzeugbauer die Manipulation von Feldwerten eines Objekts auch für

[1] Die Zugriffsobjekte können als private Felder realisiert werden. Im Fall von nicht in der Werkzeugspezifikation enthaltenen Typen muss über Reflection zugegriffen werden, was teurer ist.

[2] An dieser Stelle sei bemerkt, dass die Standardimplementierung die Laufzeit für die Konstruktion erst beim Iterieren verwendet. Die Darstellung der entsprechenden Komplexität wäre aber bestenfalls verwirrend. Hat jeder Typ mindestens eine Instanz, so sinkt die Komplexität der Operationen des Iterators auf $O(1)$, weil die Kosten für das Traversieren der Typstruktur in die Traversierung der Objekte mit eingerechnet werden kann. Wie man in §6.3.3 sehen wird, kann die Traversierung zum nächsten Typ tatsächlich zu konstanten Kosten erfolgen.

unbekannte Felder und unbekannte Typen. Objekte bieten neben dem reflektiven Feldzugriff auch benannte get- und set-Funktionen für alle statisch bekannten Felder. Zugriffsmethoden von Objekten verbergen vor dem Werkzeugbauer, ob der Zugriff an ein Feld des Objekts oder an eine verteilte Datenstruktur delegiert wird. Hierdurch lässt sich die tatsächliche Darstellung von Objekten verändern, ohne das API zu beeinflussen. Handelt es sich um einen echten Feldzugriff, so wird jeder vernünftige Compiler den Methodenaufruf durch Inlining beseitigen.

Die Speicherverwaltung aller mit einer `.sf`-Datei assoziierter Objekte verantwortet das Dateiobjekt. Daher werden neue Instanzen eines Typs üblicherweise[1] über eine Fabrikmethode erzeugt. Ebenso können Objekte nicht direkt gelöscht werden. Tatsächlich erfordert das unmittelbare, konsistente Löschen eines Objekts die Traversierung des kompletten Graphen. Deswegen bietet das API nur die Möglichkeit, ein Objekt bei der nächsten Schreiboperation zu löschen. Hier kann die Konsistenz mit geringem Mehraufwand während der ohnehin notwendigen Traversierung des Graphen gewährleistet werden. Die Löschfunktion ist im Dateiobjekt zu finden. In Datenobjekten ist sie deplatziert, da diese üblicherweise weder ihr Zugriffs- noch ihr Dateiobjekt kennen. Zugriffsobjekte würden das Löschen von Objekten mit passendem statischem Typ gestatten. Da die Speicherverwaltung jedoch den passenden dynamischen Typ ermitteln müsste, ist die beste Position im Zugriffsobjekt, welches die notwendigen Informationen selbst verwaltet.

Da dem Werkzeugbauer Zeiger auf Objekte gegeben werden und in Objekten die Manipulation von Felddaten nicht wesentlich eingeschränkt werden soll, um den Zugriff nicht zu verlangsamen, kann die Manipulation von Objekten nicht eingeschränkt werden. Daher ist der Werkzeugbauer für die Konsistenz des Zustands verantwortlich. Damit ist die Abgeschlossenheit des Zustands über erreichbare Objekte gemeint. Eine Prüfung der Abgeschlossenheit ist vergleichsweise teuer und für die meisten Anwendungen nicht

[1] In Programmiersprachen mit automatischer Speicherverwaltung kann man Objekte auch extern allokieren und später dem Dateiobjekt zur Verfügung stellen. Die Realisierung mit manueller Speicherverwaltung organisiert auch die Freigabe der Objekte, was üblicherweise zu unflexibel implementiert ist, um extern erzeugte Objekte zu behandeln.

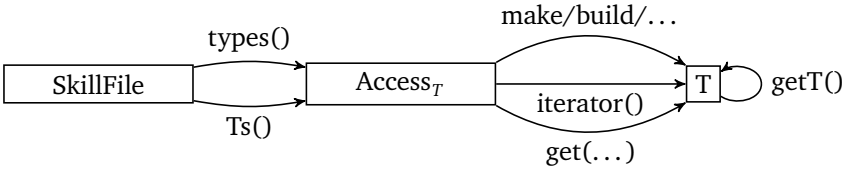



Abbildung 5.18.: Zugriffsmethoden in einem schematischen API für einen einfachen Typ T.

```
1 sf = new SkillFile("out.sf", Create, Write)
2 t1 = sf.Ts().make()
3 t2 = sf.Ts().make()
4 t1.setRef(t2)
5 t2.setRef(t1)
6 sf.close() // schreibt alle verwalteten Objekte
```

Listing 5.13: Schematische Verwendung des APIs

erforderlich. Sie ungefragt durchzuführen erscheint daher nicht gerechtfertigt. Ebenso wird eine `.sf`-Datei nicht vor externen Manipulation während des Werkzeuglaufs geschützt. Eine manuelle Prüfung der Abgeschlossenheit kann durch den Werkzeugbauer in der SKilL/Scala-Implementierung durchgeführt werden.

Ein Dateiobjekt bietet eine Methode, die die Konsistenz des Zustands prüft. Dabei wird im Allgemeinen jedoch nicht geprüft, ob ein *Anhängen*[1] legal wäre. Daher obliegt es der Verantwortung des Werkzeugbauers, den richtigen Schreibmodus zu wählen.

Zur Illustration sind die Zugriffsmethoden zu einer einfachen IR-Spezifikation `T { T ref; }` in Abb. 5.18 dargestellt. Der Zugriff auf das spezifizierte Feld `ref` durch get- und set-Funktionen hat den Vorteil, dass die Realisierung des Feldes im Nachhinein geändert werden kann, ohne die API zu verändern.

[1] Um dies zu gewährleisten müsste man die Ursprungsdatei erneut einlesen und mit dem aktuellen Zustand vergleichen, was einen nicht zu rechtfertigenden Aufwand darstellt.

```
SkillFile.open("in.sf", Mode.ReadOnly).Ts().stream().
    forEach(System.out::println);
```

Listing 5.14: Schreibt alle Ts der Datei auf die Strandardausgabe

Die konzeptionelle Erzeugung einer `.sf`-Datei mit zwei `T`-Instanzen, die über `ref` einen Zyklus bilden ist in Listing 5.13 dargestellt. Es ist jedoch zu beachten, dass die Typisierung oder Benennung der Zugriffsfunktionen nicht eingeschränkt werden sollte, um eine möglichst natürliche Integration in eine Zielsprache zu ermöglichen. So kann man in Java beispielsweise alle `T`-Instanzen einer `.sf`-Datei mit dem Statement in Listing 5.14 ausgeben. Mit dem Iterator über alle Typen (`types`) kann analog dazu die Ausgabe aller Instanzen aller Typen implementiert werden.

Die generierte Anbindung kann zusätzlich mithilfe sogenannter Hints (siehe §5.1.6) durch die Werkzeugspezifikation verändert werden. Hints führen konzeptionell zu keinen sichtbaren Veränderungen des APIs. Die wichtigsten Beispiele sind `!distributed` und `!ondemand`. `!distributed` fordert, dass die Daten des annotierten Felds nicht im Objekt selbst gespeichert werden, sondern in einer verteilten Datenstruktur. Hierdurch kann das Laufzeitverhalten optimiert werden. `!ondemand` impliziert `!distributed` und fordert zusätzlich, dass die Felddaten erst beim ersten Zugriff aus der `.sf`-Datei gelesen werden.

## 5.4. Erweiterungen der Spezifikationssprache

In diesem Abschnitt wird der bereits kennengelernte SKilL-Kern erweitert. Die SKilL-Sprachspezifikationen teilt Funktionalität in Kernbestandteile und Erweiterungen ein. Ziel dieser Einteilung ist primär einen benutzbaren Kern zu definieren, der erwartungsgemäß innerhalb von etwa 3 bis 6 Mannmonaten für eine neue Programmiersprache implementiert werden kann. Um mittelfristig eine Verfügbarkeit in verbreiteten Programmiersprachen zu gewährleisten, ist es wichtig die Einstiegshürden bei der Implementierung

einer Anbindung für eine neue Programmiersprache möglichst niedrig zu halten. Die Erweiterungen sind so entworfen, dass ihr Fehlen in einer Implementierung eines Werkzeugs die Interaktion mit anderen Werkzeugen, welche diese implementieren, nicht beeinträchtigt.

Diese Erweiterungen sind alle so konstruiert, dass sie sich in der Zwischendarstellung des Quellcode-Generators so bearbeiten lassen, dass sie bei Nichtbeachtung durch das Quellcode-Generator-Back-End keine Inkompatibilität verursachen. Ihre Implementierung in einer SKilL-Anbindung ist also optional. Alle hier eingeführten Konstrukte werden im Namensraum der Typen geführt und in diesem Zusammenhang wie reguläre Typdefinitionen behandelt.

### 5.4.1. Interfaces

Interfaces existieren in SKilL um Eigenschaften von Objekten weiter zu strukturieren, als das in einer Vererbungshierarchie mit Einfachvererbung möglich ist. So könnte man beispielsweise in der strukturellen Beschreibung eines analysierten Programms Knoten, deren Wert aus einem Token hervorgeht, durch Einführung eines Supertyps *Literal* zusammenfassen wollen. Von diesen Knoten wird beispielsweise eine Zeigeranalyse unterschiedlich beeinflusst. Eine Zwischendarstellung für eine Analysekette direkt so zu gestalten, dass diese Knotentypen einen gemeinsamen Supertyp haben, führt zu einer wenig hilfreichen Typisierung, insbesondere, weil sich die Menge der Knoten-Typen, die diese Eigenschaft haben, je nach Art und Präzision einer Zeigeranalyse unterscheiden kann. Um dem zu begegnen, wurden Interfaces eingeführt, die aus Spezifikationssicht wie Versprechen von Typen behandelt werden, die von der tatsächlichen Implementierung dann automatisch einzulösen sind. Die Einfachvererbung auf regulären benutzerdefinierten Typen wird also durch eine Mehrfachvererbung über Interfaces ergänzt.

Um Interfaces in SKilL einzuführen, muss man zunächst die Spezifikationssprache erweitern. Da wir in Deklarationen bereits mehrere Supertypen zugelassen haben, müssen wir nur noch die Deklarationsregel um eine Ableitungsregel für Interfaces erweitern (siehe Abb. 5.19). Dabei verhalten

⟨*declaration*⟩ ::= ⋯
| ⟨*interface-type*⟩

⟨*interface-type*⟩ ::= ⟨*comment*⟩? `interface` ⟨*ID*⟩
(( ':' | `with` | `extends` ) ⟨*ID*⟩)* '{' ⟨*field*⟩* '}'

Abbildung 5.19.: Ableitungsregel für Interfaces

sich Interfaces weitgehend wie reguläre Typen. Es wird hier lediglich unterbunden, Hints und Restrictions auf das Interface selbst anzuwenden. Ursächlich hierfür ist, dass das Interface schon in der Zwischendarstellung des Quellcode-Generators entfernbar sein muss, da Interfaces sonst zum SKilL-Kern gehören müssten. Eine solche Verarbeitung führt dazu, dass entsprechende Informationen schon den Back-Ends des Quellcode-Generators nicht mehr vorliegen.

Auf Seiten der semantischen Analyse des Quellcode-Generator muss die Typprüfung angepasst werden. Zunächst erhalten wir die Regel zur Eindeutigkeit von Feldern, d.h. alle von einem Typ aus über die Vererbungshierarchie sichtbaren Felder müssen einen eindeutigen Namen haben. Ist dasselbe Feld über zwei unterschiedliche Vererbungspfade sichtbar, so gilt es als eindeutig.[1] Weil es sich bei Interfaces konzeptionell um strukturelle Versprechen handelt, ist unerheblich, wie oft diese gegeben werden. Ein Interface kann also beliebig oft auf beliebigen Pfaden geerbt werden. Die Semantik bleibt, dass die dadurch geerbten Supertypen und eingeführten Felder nur einmal eingeführt werden. Erbt ein Typ und dessen Supertyp dasselbe Interface, so werden die eingeführten Felder nur im Supertyp eingeführt. Dies ist jedoch für die semantische Analyse transparent.

Interfaces dürfen das Versprechen, Subtyp eines anderen regulären Typs zu sein, realisieren, indem sie einen regulären Typ als Supertyp deklarieren. Hierdurch verändert sich die Typprüfung allerdings ein wenig. Zunächst wird der gesamte Typgraph aufgebaut und topologisch sortiert, um

[1] Im Sinne des Quellcode-Generators haben alle Felddeklarationsknoten eindeutige Namen.

dabei Zyklen zu erkennen und als Fehler zu bearbeiten. Der zyklenfreie Vererbungsgraph wird nun vereinfacht, um zu einer leichter bearbeitbaren Zwischendarstellung zu gelangen. Hierfür werden reguläre Typen (im Folgenden Klassen) und Interfaces getrennt. Für alle Typen ist deren Supertyp und vor allem dessen Eindeutigkeit zu bestimmen. Erbt eine Klasse $C$ eine Klasse und ein Interface $I$, so ist, falls $I$ selbst von einer Klasse $T$ erbt, zu prüfen, dass $T$ ein transitiver Supertyp von $C$ ist. Vereinfacht gilt: $C <: I \land C <: S \land I <: T \Rightarrow C <: T \land (T <: S \lor S <: T)$.

Diese Einschränkung sorgt für die Wohlgeformtheit der auf Klassen projizierten Typhierarchie, welche für die Abbildung auf eine Interface-freie Zwischendarstellung erforderlich ist. In der bereits topologisch sortierten Form kann man dies einfach dadurch erreichen, dass man von oben nach unten die Superklassen aller Supertypen bestimmt und schneidet. Ist der Schnitt trotz vorhandener Superklassen leer, so ist ein entsprechender Fehler zu melden, da die Vererbungshierarchie nicht konsistent ist. In einem letzten Schritt sind alle Interface-Vererbungen zu entfernen, die bereits durch einen Supertyp bestehen. Dieser Schritt vereinfacht die Arbeit auf der Zwischendarstellung erheblich.

Eine Konsequenz dieses Vorgehens ist, dass eine Vererbungsbeziehung zwischen zwei Klassen bestehen kann, obwohl diese nicht explizit in der IR-Spezifikation vorhanden ist, wenn diese über ein Interface eingeführt wird, beispielsweise $T <: I <: S$. Das führt dazu, dass das generierte API auf den ersten Blick einer anderen Vererbungshierarchie folgt, welche $I$ nicht mehr erhält, also $T <: S$. Tatsächlich ist dies aber nicht der Fall. Eine derartige Verwendung von Interfaces zu verbieten, bietet keine erkennbaren Vorteile.

Auf Ebene der Zwischendarstellung lassen sich Interfaces leicht entfernen. Die von den Interfaces eingeführten Felder werden in topologischer Reihenfolge in alle direkten Subtypen kopiert. Alle Verwendungen von Interfaces werden durch deren Superklasse ersetzt. Existiert keine Superklasse, so wird stattdessen `annotation` verwendet. Danach werden alle Interface-Deklarationen und damit verbundene Vererbungsbeziehungen entfernt.

Diese Art der Darstellung von Interfaces hat nicht nur den Vorteil, dass

```
typedef natural
  @min(0)
  v64;

Person {
  natural age;
}
```

Listing 5.15: Definition und Verwendung von Typedefs

sie keinen Zusatzaufwand für die Implementierer der Quellcode-Generator-Back-Ends mit sich bringt, sie erlaubt auch ein nachträgliches Gruppieren von Eigenschaften in Interfaces, ohne die Zwischendarstellung zu ändern.

Implementiert eine Anbindung Interfaces tatsächlich, so ist im Allgemeinen in den generierten Get-Funktionen eine Laufzeittypprüfung erforderlich. Diese verlagert sich aber lediglich über die Fassade der Implementierung vom Anwendungscode in den generierten Code, da die vom Interface zur Verfügung gestellten Felder ohne eine Typkonvertierung ohnehin nicht zugänglich wären.

### 5.4.2. Typedefs

In IR-Spezifikationen dienen Typedefs der Gruppierung ähnlicher Verwendungen von Feldtypen. In Listing 5.15 ist ein Beispiel[1] für die Deklaration eines Typs `natural`, welcher nur natürliche Zahlen speichert, dargestellt. Ein solcher Typ ist im SKilL-Kern nicht vorhanden. Die entsprechende Restriktion immer wieder schreiben zu müssen, ist allerdings benutzerunfreundlich und bietet Raum für Tippfehler.

Die Syntax der Spezifikationssprache ist durch eine entsprechende Deklarationsregel zu erweitern (siehe Abb. 5.20). Die semantische Prüfung erscheint zunächst trivial, da für Typedefs keine Regeln gelten müssen.

[1] Die verwendete Restriktion `min` ist syntaktischer Zucker für `range` von 0 bis zum maximalen Wert des Datentyps; beide Grenzen inklusive.

⟨*declaration*⟩ ::= ⋯
| ⟨*typedef*⟩

⟨*typedef*⟩ ::= ⟨*comment*⟩? `typedef` ⟨*ID*⟩ (⟨*restriction*⟩ | ⟨*hint*⟩)* ⟨*type*⟩ ';'

Abbildung 5.20.: Ableitungsregel für Typedefs

Tatsächlich müssen aber alle übrigen Prüfungen so erweitert werden, dass Typedefs für diese transparent werden. Da Typedefs Typen modifizieren können, ist zudem darauf zu achten, dass diese Modifikationen zulässig sind. Die einfachste Methode dies zu erreichen ist, Typedefs aus der Zwischendarstellung des Quellcode-Generator zu entfernen und das Resultat zu prüfen und dann auf die Originaldarstellung zu schließen. Die dadurch entstehenden Fehlermeldungen sind zwar von suboptimaler, aber immer noch ausreichender, Qualität. Die dabei zu verwendende Transformation ist eine einfache Projektion. Die Typhierarchie ist topologisch zu sortieren. Danach können Verwendungen von Typedefs durch deren Definition substituiert werden. Transitive Typedefs werden hierdurch automatisch bearbeitet. Würde bei einer Substitution eine Restriction oder ein Hint in eine Vererbungsbeziehung eingeführt, so ist eine entsprechende Fehlermeldung zu erzeugen, weil Typen nicht im Rahmen einer Verwendung als Supertyp modifiziert werden können.

### 5.4.3. Enums

In SKilL existiert ein Enumerationskonzept, welches von enums in Java (siehe [GJS+14] §8.9) inspiriert ist. So können enums neben benannten Werten auch Felder besitzen. Die erforderliche Erweiterung der Spezifikationssprache ist in Abb. 5.21 dargestellt. Da die Verwendung von Restriktionen und Hints nicht möglich ist, erfordert die semantische Prüfung keine erwähnenswerten Schritte.

Die Projektion in der Zwischendarstellung wird anhand eines kurzen

⟨*declaration*⟩ ::= ⋯
| ⟨*enum-type*⟩

⟨*enum-type*⟩ ::= ⟨*comment*⟩? `enum` ⟨*ID*⟩ '{' ⟨*ID*⟩ (',' ⟨*ID*⟩)* ';' ⟨*field*⟩* '}'

Abbildung 5.21.: Ableitungsregel für Enums

```
1 enum E {
2   A, B;
3   bool f;
4 }
```

Listing 5.16: Enum vor Projektion

Beispiels erläutert[1] (Lst. 5.16 und 5.17). Für jedes Enum wird zunächst unter dessen Namen ein abstrakter Basistyp eingeführt, der alle Felder des Enums erhält. Für jede Instanz wird ein neuer Typ eingeführt, dessen Name aus dem des Enums und dem der Instanz getrennt durch einen Doppelpunkt besteht. Hierdurch ist Kollisionsfreiheit sichergestellt. Doppelpunkte werden bei der Bereinigung der Bezeichner automatisch verarbeitet. Jeder Instanztyp erhält ein `@singleton`, damit es genau eine entsprechende Instanz gibt.

Diese Art der Darstellung von Enumerationstypen hat den Vorteil, dass sie sich gut in die bestehende Infrastruktur integriert und man ohne weiteres Zutun die Änderungstoleranz der übrigen Typen in SKilL erbt. In dieser Form haben `enums` in SKilL für einige Zeit existiert. Während der Verwendung haben sich jedoch einige erwähnenswerte Kritikpunkte dieser Darstellung herauskristallisiert. Der wohl wichtigste Punkt ist, dass für keine Anbindung eine unprojizierte Form der enums entwickelt wurde, da sich eine natürliche Integration in keiner Programmiersprache anbietet. Das primäre Problem dabei ist die Änderungstoleranz, welche eine Abbildung auf enums schwierig

[1] Es handelt sich nach der Projektion in der hier dargestellten Form nicht mehr um eine valide IR-Spezifikation, da es keine Möglichkeit gibt, einen Doppelpunkt in einem Bezeichner zu platzieren.

```
@abstract
E {
  bool f;
}

@singleton
'E:A' extends E {}

@singleton
'E:B' extends E {}
```

Listing 5.17: Enum nach Projektion (Illustration)

macht, da es Werte geben kann, die statisch nicht bekannt sind. In Java ergibt sich zudem das Problem, dass enums keinen Supertyp haben können, was in Kombination mit dem SKilL-Typ `annotation` zu einem Problem führt.[1]

Die ursprüngliche Abbildung der zur Evaluation verwendeten Zwischendarstellung verwendete Enums, wie sie hier beschrieben sind. Die nun tatsächlich verwendete Abbildung verwendet für Enums speziell markierte Integer (siehe [Prz16]). In der Masterarbeit von Constantin Weisser [Wei16] wird die bestehende Abbildung zu Recht kritisiert und weiterhin vorgeschlagen, diese durch Strings zu ersetzen. Diese Punkte sollten Anlass sein, sich in zukünftigen Revisionen des SKilL-Standards dem Thema `enums` erneut zu widmen.

### 5.4.4. `custom`- und `auto`-Felder

Es ist vorgesehen, dass Werkzeuge direkt auf den generierten Datenobjekten arbeiten. Manche Algorithmen profitieren dabei sehr von der Möglichkeit, diese temporär um weitere Informationen zu erweitern, die nur während einer Berechnung von Relevanz sind. Ein Beispiel wäre etwa ein Flag, das zur

[1] Ohne Enums kann man einen Typ SkillObject einführen, der seinen SkillTyp kennt und eine Objekt-ID in einem Feld hält. Würde man auf diesen Typ verzichten, so würden Serialisierungsfunktionen für `annotation` erheblich teurer werden.

⟨*field*⟩ ::= ⟨*description*⟩ (⟨*constant*⟩ | ⟨*data*⟩ | ⟨*custom*⟩) ';'

⟨*data*⟩ ::= (`auto`)? ⟨*type*⟩ ⟨*ID*⟩

⟨*custom*⟩ ::= `custom` ⟨*ID*⟩ ('!' ⟨*ID*⟩ (⟨*STRING*⟩ ('(' ⟨*STRING*⟩* ')')?)?)* ⟨*STRING*⟩ ⟨*ID*⟩

Abbildung 5.22.: Ableitungsregel für `auto`- und `custom`-Felder

Zyklenerkennung verwendet wird. Hierfür existieren in SKilL zwei Mechanismen. Zum einen können mit dem Schlüsselwort `auto`[1] reguläre Felddefinitionen verwendet werden, um über das SKilL-Typsystem Felder zu deklarieren, die von der Serialisierung ausgenommen sind. Dieser Weg hat den Vorteil einer sehr einfachen Integration in die bestehende Infrastruktur. Es ist aber offensichtlich nicht möglich, eine Referenz auf eine Systemressource, wie etwa einen Mutex, direkt in einem Datenobjekt abzulegen.[2]

Um dies ebenso zu ermöglichen, wurde die Spezifikationssprache um sogenannte `custom`-Felder erweitert. Diese Felddefinitionen beginnen mit dem Schlüsselwort `custom` und dem Namen eines Quellcode-Generator-Back-Ends. Es folgt eine Liste von Optionen in an Hints angelehnter Syntax (siehe §5.1.6). Die Definition endet mit einem String und einem Bezeichner. Der String stellt den Typ in der Zielsprache dar, der Bezeichner dient als Feldname. Im Gegensatz zu anderen Felddefinitionen wird bei der Beurteilung der Eindeutigkeit nicht nur der Bezeichner, sondern auch der Name des Quellcode-Generator-Back-Ends miteinbezogen. Hierdurch kann sichergestellt werden, dass vergleichbare Anbindungen für mehrere Programmiersprachen geschaffen werden können und die generierten Feldnamen dennoch eindeutig bleiben. Die Darstellung des Feldtyps über einen String ermöglicht die Verwendung von Leerzeichen, Punkten, Sternen, usw. Die

[1] `auto` ist in C und C++ ein Schlüsselwort, welches auf Serialisierung übertragen keine Semantik hat. Es wurde daher für *auto*matisch generierte nicht-serialisierte Felder in der generierten IR verwendet.

[2] Ebenso können diese nicht in einem `auto`-Feld abgelegt werden, da diese über das SKilL-Typsystem typisiert werden, welches nur serialisierbare Typen enthält.

```
Custom {
    custom ada
    !with "System.Address"
    "System.Address" any;

    custom java
    !import "java.lang.Object"
    !modifier "public transient"
    "Object" any;

    auto annotation anySkillObject;
}
```

Listing 5.18: Custom- und Auto-Felder

Abbildung und jede weitere semantische Prüfung hängen inhärent von der Zielsprache ab.

Eine Projektion nicht serialisierter Felder kann durch simples Entfernen realisiert werden. Die Anpassung der Grammatik ist in Abb. 5.22 dargestellt. Ein kurzes Beispiel findet sich in Listing 5.18.

Kapitel 6

# Implementierung

In diesem Kapitel werden Implementierungsstrategien der später vermessenen SKilL-Implementierungen im Detail erläutert. Die Implementierung beschreibt, wie man die unterspezifizierten Teile des Designs ausfüllen kann, um zu einer leicht implementierbaren Architektur zu gelangen. Die Darstellung erfolgt dabei vom API in Richtung Austauschformat, da diese Richtung verständlicher ist als der umgekehrte Weg. Eine Erklärung auf Basis des zur Konstruktion der Implementierung genutzten Refinement-Ansatzes würde der Verständlichkeit schaden, da vom Design zur Implementierung zu viele Zwischenschritte notwendig sind. Die Beschreibung bezieht sich dabei primär auf die Werkzeugsicht und damit auf die Repräsentation der IR im Hauptspeicher.[1] Obwohl die Code-Generierung weitgehend auf klassischen Compilerbaumethoden basiert, wird diese hier ebenfalls kurz erläutert. Danach wird die Architektur der Werkzeugschnittstelle grob umrissen und nicht-triviale Aspekte der Implementierung werden im Detail betrachtet.

Zur hier vorgestellten Architektur sind zunächst einige grundlegende Entscheidungen zu erläutern. Um möglichst effiziente Analysen implementieren

[1] Diese ist sowohl für die Effizienz des darauf aufbauenden Anwendungscodes als auch für die der Anbindung entscheidend.

zu können, sind Instanzen durch Objekte und Referenzen durch Zeiger repräsentiert. Alle Daten, die bekannt und nicht bedarfsorientiert sind, werden direkt während der Leseoperation deserialisiert. Die Architektur wurde entwickelt, um die Laufzeit auf relevanten[1] Datensätzen zu minimieren und bestmöglich auf große Datensätze zu skalieren. D.h. sowohl die tatsächliche worst-case-Komplexität, als auch der Speicherverbrauch spielen eine untergeordnete Rolle. Die Minimierung von Code-Menge, Compile-Zeit und Größe des ausführbaren Programms wurden als sekundäres Ziel aufgefasst.

An dieser Stelle seien noch die zahlreichen studentischen Arbeiten erwähnt, die insbesondere im Bereich Usability einen äußerst positiven Einfluss auf den Status Quo dieses Projektes hatten (im einzelnen [Prz14; Rot15; Ung14; Har14; Völ15]).

## 6.1. Code-Generierung

Der Quellcode-Generator besteht zunächst aus einem Front-End für `.skill`-Spezifikationen, einer Zwischendarstellung, diversen Back-Ends und einem Kommandozeilen-Interface. Der Quellcode-Generator ist hauptsächlich[2] in Scala implementiert und folgt in weiten Teilen einem funktionalen Programmierstil.

Das Quellcode-Generator-Front-End hat keine besonderen Eigenschaften und ist selbst wenig optimiert, da es auf den Gesamtressourcenverbrauch des Quellcode-Generators kaum Einfluss hat. Einzig die Behandlung von Bezeichnern ist ungewöhnlich. Hier ist es möglich, durch Parameter eine automatische Erkennung von Wortbestandteilen zu steuern. Diese erlaubt es etwa, CamelCaseWorte in die Bestandteile *Camel*, *Case* und *Worte* zu trennen, sodass im Quellcode-Generator-Back-End einfach Bezeichner generiert werden können, die besser zur Konvention der Zielsprache passen. Die hierfür verwendete Strategie trennt bei einer Folge von Unterstrichen

[1] Neben ein paar artifiziellen Tests sind das insbesondere die im Kapitel Evaluation verwendeten Daten.

[2] Die Zwischendarstellung ist in Java implementiert, um Studenten die Arbeit auf dieser zu erleichtern.

am ersten Unterstrich und wirft diesen weg, d.h. `a_b` wird {`a`, `b`} und `a__b` wird {`a`, `_b`}. Außerdem wird vor einem Großbuchstaben, der von einem Kleinbuchstaben gefolgt ist, getrennt, d.h. `IRSpec` wird {`IR`, `Spec`}, `HUGE` wird {`HUGE`} und `myName` wird {`my`, `Name`}. Großbuchstaben bleiben in Wortbestandteilen bestehen, um Akronyme korrekt zu behandeln. Die SKilL-Namen werden erzeugt, indem alle Bestandteile konkateniert und in Kleinbuchstaben umgewandelt werden.

Die Zwischendarstellung des Quellcode-Generators ist in Java implementiert, um ihre Verwendung nach Außen, d.h. für den Implementierer eines Back-Ends, einfacher zu gestalten. Die Datenstrukturen sind allesamt nach Außen unveränderlich. Nach Innen werden einige Attribute, wie etwa verschiedene Repräsentationen von Bezeichnern, zwischengespeichert, um mehrfache Berechnungen dieser zu vermeiden. Auf Zwischendarstellungsebene lassen sich die in §5.4 beschriebenen Erweiterungen der `.skill`-Spezifikationssprache durch vergleichsweise einfache Ersetzungen auf den SKilL-Kern abbilden. Die Ersetzungen folgen dem bereits beschriebenen Vorgehen.

Die Quellcode-Generator-Back-Ends lassen sich in zwei Kategorien einteilen. Die erste Kategorie bilden vergleichsweise einfache Back-Ends, die etwa unkomplizierte statistische Erhebungen der IR-Spezifikation durchführen oder diese gleichmäßig formatiert in einer Datei ausgeben (vergleiche z.B. Tab. 7.5). Die zweite Kategorie besteht aus Back-Ends, die zu einer Programmiersprache eine Anbindung generieren. Die detaillierte Beschreibung der Back-Ends befasst sich nur mit letzterer Variante, weil nur diese Back-Ends wirklich erwähnenswerte Eigenschaften haben.

Gemeinsame Funktionen der meisten Quellcode-Generator-Back-Ends sind in einer Klasse zusammengefasst. Diese stellt sicher, dass einige notwendige Funktionen, wie etwa die Bereinigung von Bezeichnern, implementiert werden, und bietet Hilfsfunktionen etwa zum Erzeugen einer neuen Quelldatei. Diese Klasse dient vor allem als Schnittstelle zum Kommandozeilen-Interface. Hierdurch können Optionen wie Ausgabepfade, der Inhalt des generierten Kopfkommentars oder der für das Generat zu nutzende Paketname leicht gesetzt werden.

Die Quellcode-Generator-Back-Ends sind durch Komposition von rein funktionalen Übersetzungsfunktionen realisiert. Es existiert üblicherweise auf oberster Ebene eine Funktion für jedes erzeugte Dateischema. Ein Dateischema ist eine einheitliche Gruppe von Quelltextdateien, also beispielsweise Typdefinitionen für das API für alle in der IR spezifizierten Typen oder die Implementierung des Dateiverwaltungsobjekts. Die gemeinsam genutzten Übersetzungsfunktionen wie etwa die Abbildung von Zwischendarstellungstypen auf die Zielsprache, die Abbildung von Kommentaren, Parameterlisten oder Standardwerten erfolgt üblicherweise an zentraler Stelle. Lokal werden Übersetzungsfunktionen bei Bedarf weiter zerlegt, etwa um die Implementierung der Deserialisierung eines Feldes zu realisieren.

Da das Front-End die Zwischendarstellung im Quellcode-Generator in Typordnung sortiert erzeugt, kann ein Quellcode-Generator-Back-End Code ohne Berücksichtigung oder Prüfung einer Reihenfolge generieren. So wird beispielsweise, um sicherzustellen, dass alle spezifizierten Typen auch vorhanden sind, zu jedem Typ eine Prüfung, ob der Typ vorhanden ist, generiert. Falls der Typ nicht vorhanden ist, kann dieser direkt erzeugt werden. Der dabei notwendige Zugriff auf dessen Supertyp, falls er einen hat, ist ungeprüft zulässig.

## 6.2. Architektur der Werkzeuganbindung

Die Fragen, wie eine Anbindung intern aufgebaut ist oder wie eine Anbindung in die Zwischendarstellung eines Werkzeugs integriert wird, werden vom Design des Serialisierungssystems bewusst nicht beantwortet. Hier bieten sich grundsätzlich einige Alternativen. Die involvierten Komponenten in den zu betrachtenden Architekturen sind der Anwendungscode, die Werkzeug-IR,[1] die SKilL-IR,[2] die generierte SKilL-Implementierung, SKilL-`common`, sowie der Datenstrom.

[1] In dieser Betrachtung ist unerheblich, ob sich mehrere Werkzeuge einer Werkzeugkette dieselbe IR-Implementierung teilen. Ferner kann die Werkzeug-IR Teil das Anwendungscodes sein.

[2] Also die von der Anbindung verwendeten Datenobjekte.

Die erste Fragestellung ist, ob Werkzeug-IR und SKilL-IR identisch sind. Eine schematische Darstellung der wesentlichen Varianten bietet Abb. 6.1. In dieser Arbeit wird vorgeschlagen, als Werkzeug-IR das vom Quellcode-Generator erzeugte API zu verwenden, welches von der SKilL-IR realisiert wird. Hierdurch muss keine eigene Werkzeug-IR entwickelt werden, es muss nicht zwischen zwei Darstellungen konvertiert werden und der Quellcode-Generator kann sich blind auf Invarianten der eigenen IR-Implementierung verlassen. Um dieses Vorgehen auch bei komplexeren Werkzeugen noch produktiv durchführen zu können, wurden die in §5.4.4 beschriebenen nicht serialisierten Felder eingeführt. Besteht bereits eine Werkzeug-IR, so ist dieses Vorgehen jedoch nicht gangbar. Hier bieten sich im Wesentlichen drei Alternativen. Neben der Konvertierung zwischen den IR-Darstellungen kann versucht werden, eine der beiden Darstellungen durch die jeweils andere zu ersetzen.

Der Weg der Konvertierung bietet sich prinzipiell an, wenn SKilL lediglich als Alternative angebunden wird. Dieser Weg wurde beispielsweise bei der Konvertierung von Bauhaus-IML in eine äquivalente SKilL-Darstellung verwendet. Dieser Ansatz bringt neben den Konvertierungskosten das Problem mit sich, dass man die Strukturen der beiden Darstellungen aufeinander abbilden muss. Im Fall des erwähnten `iml2sf`-Konverters verfügen beide IRs über ein Laufzeittypsystem, welches für eine weitestgehend automatische Abbildung genutzt werden kann.

Constantin Weißer untersucht in seiner Masterarbeit [Wei16], inwieweit man eine bestehende Werkzeug-IR nutzen kann, um eine SKilL-Anbindung darauf aufzubauen. Hier wird der SKilL-Quellcode-Generator über sogenannte Mappingfiles parametrisiert, um den übrigen generierten Code so zu produzieren, dass er die bestehende IR verwendet. Das erfordert natürlich, dass Objekte von außen sichtbar und veränderbar sind.

Dennis Przytarski untersucht in seiner Masterarbeit [Prz16] den umgekehrten Weg. Hier wird die überwiegend generierte Bauhaus-IR im Wesentlichen durch eine SKilL-IR ersetzt, indem das bestehende Bauhaus-IR-API durch Verweise auf das SKilL-API ersetzt wird. Bei diesem Ansatz ist der Anpassungsaufwand für bestehende Werkzeuge zu berücksichtigen. Ist es nicht

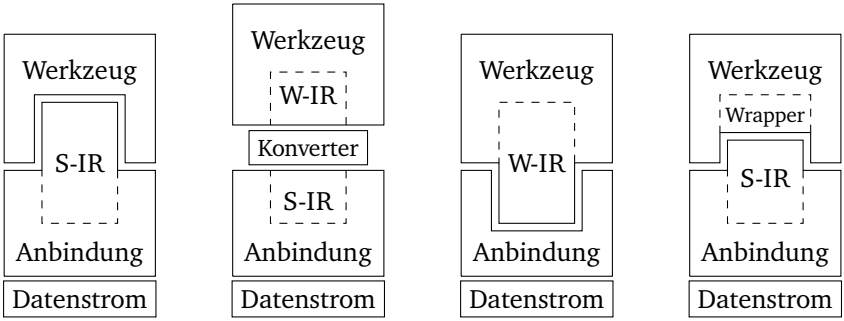



Abbildung 6.1.: Integrationsmöglichkeiten von **W**erkzeug- und **S**KilL-**IR**. Von links nach rechts: Keine eigene Werkzeug-IR; Konvertierung von/nach SKilL beim Lesen/Schreiben; Anbindung gegen spezifisches Werkzeug-IR implementiert (siehe [Wei16]); Werkzeug-IR durch Wrapper auf SKilL-IR migriert (siehe [Prz16]). SKilL-IR, -Implementierung und `-common` sind hier als Binding zusammengefasst. In allen Fällen schirmt das Binding den Datenstrom vom Anwendungscode ab.

möglich, die Werkzeug-IR durch eine API-kompatible Anpassung zu ersetzen, so wird der Anpassungsaufwand in den meisten Fällen zu hoch sein. In diesem Fall kann man auf eine Konvertierung für Bestandswerkzeuge ausweichen und bei Neuentwicklungen entscheiden, ob auf die generierte SKilL-IR zurückgegriffen wird.

Die zweite Architekturfrage betrifft die Rolle von SKilL-`common`. Diese Komponente fasst spezifikationsunabhängige Teile einer Anbindung in einer Bibliothek zusammen (siehe Abb. 6.2). Diese Komponente muss nicht prinzipiell existieren und war in den frühen SKilL-Implementierungen nicht vorhanden. Es zeigte sich jedoch, dass Quellcode-Generator-Back-Ends übersichtlicher werden, wenn man gemeinsam genutzte Teile generierter Anbindungen in eine Bibliothek auslagert. Diese Komponente ist sprachspezifisch und bietet beispielsweise eine Kapselung des Binärstroms, oder eine Implementierung des unparametrisierten Laufzeittypsystems. In der hier vorgeschlagenen Architektur erfolgt die Parametrisierung teilweise durch

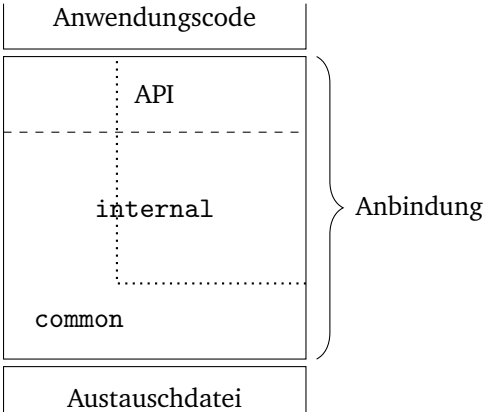



Abbildung 6.2.: Schaubild zu Komponenten innerhalb einer Anbindung.

Vererbung und Instantiierung von Methoden mit generiertem Code, was dazu führt, dass Anpassungen an Generator oder `common` oft auch eine Anpassung der jeweils anderen Komponente nach sich ziehen. Denkbare Änderungen an `common` ohne Anpassung des Quellcode-Generators, und damit auch des generierten Codes, wären in gewissen Grenzen etwa die Implementierung weiterer Restrictions oder eine Implementierung von Manipulationen am Laufzeittypsystem.[1] Auch neue Funktionen, wie etwa das Lesen einer `.sf`-Datei aus einem gegebenen Stück Hauptspeicher anstelle einer physischen Datei sollte ohne Anpassung eines Generats möglich sein.

## 6.3. Die Werkzeugschnittstelle

Um die Darstellung der Implementierung verständlich zu gestalten, wird hier ein fiktives einfaches API definiert. Außerdem wird in diesem Abschnitt eine IR-Spezifikation mit zwei Typen $S$ und $T$ verwendet. $S$ habe ein Feld $f$ und $T$ erbe von $S$ und habe ein Feld $g$. Weitere Eigenschaften sind hier

[1] Dazu zählt auch das Nachrüsten eines Prädikats für jeden Laufzeitrepräsentanten eines Feldes, welches beschreibt, ob das Feld bereits in der eingelesenen `.sf`-Datei enthalten war.

zunächst nicht relevant. In der Beschreibung wird auf eine von UML und Java inspirierte Notationen zurückgegriffen. Es wird keine existierende Notation verwendet, da sich teilweise Freiheitsgrade ergeben, die man sprachabhängig auszufüllen hat. In den einzelnen Schaubildern werden immer nur die lokal relevanten Felder und Methoden dargestellt, um diese übersichtlich zu halten. Außerdem sind insbesondere die generischen Typparameter in einigen Programmiersprachen in der Implementierung nicht sichtbar, da sich diese auf das Typsystem der Programmiersprache nicht geeignet abbilden lassen. Die Realisierung der hier unterspezifizierten Aspekte ist im konkreten Kontext leicht ersichtlich.

Die Beschreibung beginnt zunächst mit dem üblicherweise verwendeten API. Dieses ist schematisch in Abb. 6.3 dargestellt. Reflection wird einfachheitshalber zunächst ausgeblendet. Alle Teile des Pakets `common` werden durch eine gemeinsam genutzte Bibliothek zur Verfügung gestellt. Die Klassen `S` und `T` zu generieren, ist weitgehend geradlinig. Sind die Felder im Objekt gespeichert, so sind noch entsprechend private Felder zu generieren; die Implementierung der Zugriffsmethoden ist dann offensichtlich. Der Fall verteilter Felder wird im Rahmen der Reflection bearbeitet.

Der Typ `skillID` wird als Platzhalter für die Repräsentation der SkillID verwendet. Dabei handelt es sich um die Objekt-ID.[1] In einer Zielsprache, die ohnehin keine $2^{64}$ Objekte anlegen könnte, ist es unsinnig, diese durch long zu repräsentieren, da sie in jedem Objekt gespeichert werden muss. Hier einen `int` zu verwenden, liefert in der Praxis gute Ergebnisse, ist im Nachhinein leicht austauschbar und wird von den in §5.2.8 getroffenen Einschränkungen gedeckt. Die ID selbst wird als Feld in `SkillObject` gespeichert, idealerweise so, dass sie nur für die interne Implementierung sichtbar ist. Ob die Funktion `getSkillID` sichtbar ist, muss sprachabhängig entschieden werden. Soll diese Frage zum Generierungszeitpunkt entscheidbar sein, so kann die Funktion als `protected` markiert werden und in jedem Basistyp eine Funktion generiert werden, die den entsprechenden Aufruf

[1] Der Name ist historisch bedingt und kommt aus einer Architektur in der nur Objekte IDs hatten. Der Name wird in Implementierungen konsequent verwendet und wurde daher hier nicht angepasst.

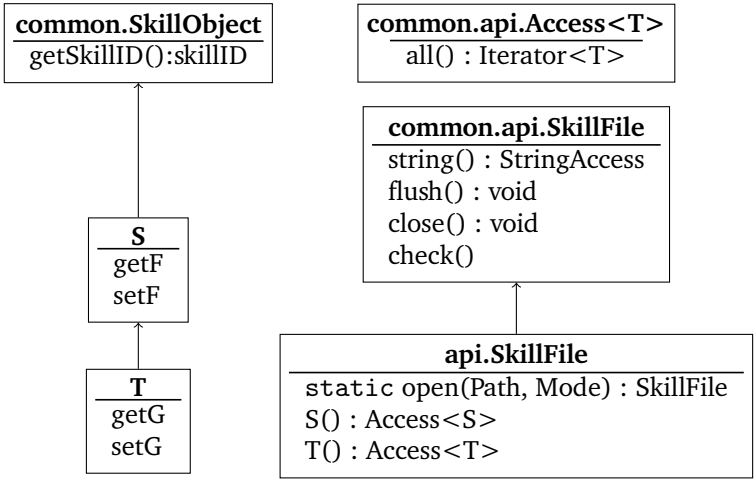



Abbildung 6.3.: Werkzeugschnittstelle aus Sicht des Werkzeugbauers.

kapselt. Die entsprechenden Funktionen werden im Rahmen des Funktionsinlinings entfernt und stellen somit keinen Performanznachteil dar.

Der Typ `api.SkillFile` ist eine abstrakte Klasse, welche lediglich die Zugriffsmethoden deklariert. Die Funktion `open` delegiert je nach Modus entweder an den `.sf`-Datei-Parser oder an eine entsprechende Funktion der internen Implementierung des `SkillFiles`. Es hat sich gezeigt, dass SKilL aus Sicht des Werkzeugbauers leichter zu verstehen ist, wenn man ein C-inspiriertes[1] API zum Lesen und Schreiben von `.sf`-Datei verwendet.

Die Operationen, die das API zur Verfügung stellen soll sind die Folgenden:

- **Open**: Ein *File* wird angelegt, es wird eine Zieldatei, sowie ein Erzeugungs- und ein Flushmodus festgelegt. Erzeugungsmodi sind *Lesen* oder *Erzeugen*. Flushmodi sind *Schreiben*, *Anhängen* und *NurLesen*. Falls der Flushmodus *NurLesen* ist, so wird ein Schreiben der Datei verhindert.
- **Flush**: Änderungen am Graph werden entsprechend des Flushmodus

[1] siehe [ISO11] §7.21.5

geschrieben.

- **Close**: Es wird *flush* ausgeführt und danach aller Speicher freigegeben bzw. ein weiteres *flush* verhindert, indem der Flushmodus auf *NurLesen* gesetzt wird.
- **Check**: Eine Prüfung aller Restrictions wird durchgeführt.

### 6.3.1. Zustandsverwaltung

Bei einer Werkzeug-IR handelt es sich aus Sicht von SKilL um potentiell unzusammenhängende Graphen, deren Struktur sich während der Benutzung beliebig ändern kann. Natürlich könnte man dem Werkzeugbauer die Verwaltung aufbürden, um sich nicht selbst mit diesem Thema beschäftigen zu müssen. Ultimativ würde dies aber dazu führen, dass, insbesondere in Programmiersprachen mit manueller Speicherverwaltung, die meisten Werkzeugbauer Speicherlecks verursachen würden, was aufgrund der potentiell mehrere GB großen Graphen im Allgemeinen keine Option ist. Wenn man die Komponenten eines sich verändernden, unzusammenhängenden Graphen verwaltet, benötigt man ein Objekt, das jeden verwalteten Knoten erreichen kann. Wenn also ein Objekt vorliegt, das ohnehin jedes verwaltete Objekt kennt, dann kann hierauf auch das API aufbauen und dem Werkzeugbauer erlauben, über alle verwalteten Objekten nach diversen Kriterien zu iterieren. Dieses Objekt wird `SkillState` genannt und ist die interne Realisierung des `SkillFile`.

Wie man in Abb. 6.4 sehen kann, beschränkt sich der spezifikationsabhängige Teil der Zustandsverwaltung auf die passende Implementierung der Typverwaltung. Die Verwaltung von Strings ist in allen Implementierungen gleich. Die konkrete Implementierung vor dem Werkzeugbauer zu verbergen, hat den Vorteil, dass man Zugriffsmethoden zur Zustandsverwaltung vor dem Werkzeugbauer verstecken kann, um sicherzustellen, dass der Werkzeugbauer keine Invarianten der Zustandsverwaltung verletzt. Hier ist, wie auch bei der Typverwaltung, ein wesentlicher Teil der Verwaltungsobjekte vor dem Werkzeugbauer verborgen. Die Implementierungen können aber in

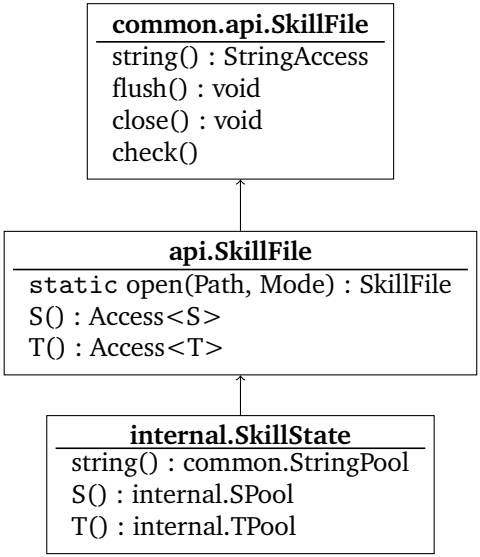



Abbildung 6.4.: Realisierung der Zustandsverwaltung.

den meisten Programmiersprachen so gestaltet werden, dass der Compiler der Programmiersprache dennoch in der Lage ist, die konkret aufgerufenen Funktionen zu identifizieren, da es zumeist global nur eine Implementierung zu einer abstrakten Methode gibt. Durch ein derartiges Design können nicht nur dispatchende Aufrufe vermieden werden, sondern es kann auch Funktions-Inlining durch den Compiler betrieben werden, was erlaubt, defakto Feldzugriffe als scheinbare Funktionsaufrufe zu repräsentieren. So ist beispielsweise anzunehmen, dass ein Aufruf von `api.SkillFile.S` direkt durch einen entsprechenden Feldzugriff im `SkillState` abgebildet wird, da dem Compiler bzw. der VM das nötige Wissen zur Verfügung steht.

Die Funktion `close` lässt sich spezifikationsunabhängig implementieren, indem `flush` aufgerufen wird. In Programmiersprachen mit manueller Speicherverwaltung ist zu entscheiden, ob diese Funktion zudem das Zustands-

objekt zerstört. Wird der Zustand nicht zerstört,[1] so wird der Flushmodus auf `NurLesen` gesetzt.

Die Funktion `flush` prüft den Flushmodus und ruft die passende Schreiboperation auf. Deren Implementierung erfolgt durch `common.FileWriter` und ist spezifikationsunabhängig. Der Grund für die Spezifikationsunabhängigkeit ist der Umstand, dass das Laufzeittypsystem der verarbeiteten `.sf`-Datei zwar zur Werkzeugspezifikation passt, dieses aber im Allgemeinen erweitert. Hierauf wird bei der Betrachtung der Reflection näher eingegangen. Die Funktion `check` wird beim Schreiben oder Anhängen vom `FileWriter` aufgerufen. Die Funktion selbst ist über Reflection realisiert.

### 6.3.2. Zugriff auf Daten

Die wohl erstaunlichste Erkenntnis dieser Arbeit ist, dass der überwiegende Teil des Forschungsaufwands verwendet wurde, um herauszufinden wie man Objekte im Hauptspeicher am besten repräsentiert. Zu diesem Thema findet man nur Literatur, die sich mit grundlegenden Problemen befasst. Schon die Frage, wie man über Instanzen von Typen mit oder ohne Untertypen effizient iteriert, ohne unverhältnismäßig große Indexstrukturen anzulegen, ließ sich nicht mehr mit bereits existierende Arbeiten beantworten. Da sich die hier entwickelten Techniken anderweitig wohl kaum veröffentlichen lassen, sie aber für die gemessene Performance essentiell sind, werden sie hier im einzelnen erläutert.

#### 6.3.2.1. Zugriff auf Elemente

Es gibt prinzipiell drei Arten, auf verwaltete Objekte zuzugreifen: über einen Iterator, über einen Index und über einen Zeiger.

Der Zugriff über einen Zeiger erfolgt nur im Rahmen eines Feldes eines anderen Objekts. Hier bieten sich keine wesentlichen Freiheitsgrade. Es ist zudem nicht möglich, den Zugriff zu beobachten, ohne ihn wesentlich zu verlangsamen. Daher muss man davon ausgehen, dass es durch diese

[1] Bei automatischer Speicherverwaltung ist das immer der Fall.

Zugriffsform zu beliebigen Veränderungen an der Graphstruktur kommen kann. Es ist also nicht möglich, sich Wurzelknoten zu merken, da sich die Eigenschaft, eine Wurzel zu sein, durch nicht beobachtete Operationen verändern kann.

Der Zugriff über einen Index scheint zunächst leicht realisierbar zu sein, ist tatsächlich aber kompliziert, da man als Werkzeugbauer einen logischen Index für jeden Typ erwartet, der zusammenhängend ist, möglichst bei 0 beginnt und bei der Anzahl der Instanzen endet. Die Komplexität entsteht hier durch die Vererbung, die dazu führt, dass man einem Objekt, je nachdem über welchen Typ man darauf zugreift, unterschiedliche Indices geben muss. Dies sollte natürlich möglichst in konstanter Zeit erfolgen. Tatsächlich ist die effizienteste der evaluierten Darstellungen für bereits in der `.sf`-Datei vorhandene Objekte linear in der Anzahl der Blöcke und für neue Objekte linear in der Anzahl der Subtypen. Erlaubt man dem Werkzeugbauer die Speicherstruktur, wie vor einem *Schreiben*, neu zu organisieren, so ergibt sich eine konstante Zugriffszeit. Die Neustrukturierung selbst ist natürlich linear in der Zahl der Objekte.

Der Zugriff über einen Iterator dient zum einen dem Werkzeugbauer und der Implementierung einiger API-Funktionen in einigen Programmiersprachen wie *map* oder *foreach*. Zum anderen werden einige Iteratoren von der Serialisierung selbst benötigt. Das Lesen von Felddaten erfordert einen Iterator, der alle Instanzen eines Typs in aufsteigender ID-Ordnung zurückgibt. Das Umsortieren eines Pools zur Vergabe neuer IDs in Typordnung erfordert einen Iterator, der alle statischen Instanzen eines Pools in beliebiger Reihenfolge zurückgibt. Es ist dabei zwingend erforderlich, dass das Iterieren zum nächsten Element nicht nur konstant teuer, sondern auch möglichst günstig ist. Das API bietet üblicherweise die Möglichkeit, alle statischen Instanzen in ID-aufsteigender sowie alle dynamischen Instanzen in ID-aufsteigender bzw. in Typ-aufsteigender Ordnung zu traversieren. Um Objekte in konstanter Zeit anlegen zu können, können sie nicht direkt allen Unter- oder Obertypen bekannt gemacht werden, sondern werden nur im anlegenden Pool verwaltet. Würde man Objekte jedem Pool der transitiven Supertypen bekannt machen, wären die Kosten hierfür linear in der Zahl der transitiven Supertypen.

#### 6.3.2.2. Graphstruktur

SKilL schränkt die Struktur serialisierter Graphen in keinster Weise ein. Es gibt insbesondere keine ausgewiesenen Wurzelknoten und Graphen sind im Allgemeinen nicht zusammenhängend. Außerdem ist es zulässig, dass ein Werkzeugbauer einen Graphen lädt, alle Referenzen, sprich Kanten, irgendwo zwischenspeichert, auf *null* setzt und danach beliebig neu vergibt. Davon darf die Zustandsverwaltung nicht beeinträchtigt werden. Wie bereits erwähnt wurde, existieren Referenzen flach in Objekten, sodass deren Verwendung zwar effizient ist, von der Zustandsverwaltung aber nicht beobachtet wird. Hieraus ergibt sich die Notwendigkeit einer Referenz auf jedes verwaltete Objekt in einem Zustandsobjekt, was zu einem leicht erhöhten Verwaltungsaufwand und einer weniger kompakten Darstellung gegenüber der serialisierten Form und anderen Serialisierungslösungen führt (siehe Abschnitt 7.8.4).

Andererseits kann dies durch ein Array aus Zeigern realisiert werden, was sich direkt für eine effiziente Traversierung, zumindest von Basistypen, eignet. Da dieses Array ein wesentlicher Teil des internen Zustands der Typverwaltung ist, sollte dieses nicht über das API exportiert werden. Die bereits erwähnte Möglichkeit, auf Objekte via `skillID` zuzugreifen, ist trivial realisierbar. Dieser Zugriffspfad hat allerdings den Nachteil, dass die erste ID eines Typs keinen vorhersehbaren Wert hat, da beliebig viele IDs davor an Instanzen von Supertypen vergeben werden können. Daher wird in der Werkzeug-API üblicherweise auf Iteratoren zurückgegriffen, deren Implementierung im Folgenden erläutert wird.

### 6.3.3. Iteratoren

In diesem Abschnitt wird die Implementierung effizienter Iteratoren über Instanzen eines Typs sowie die dafür erforderlichen Zwischenschritte hergeleitet. Gäbe es keine Vererbung, so wäre die Implementierung der Iteratoren nicht erwähnenswert. Selbst wenn es nur Iteratoren für Basistypen gäbe, könnte man einfach das Array der bekannten Instanzen ablaufen und da-

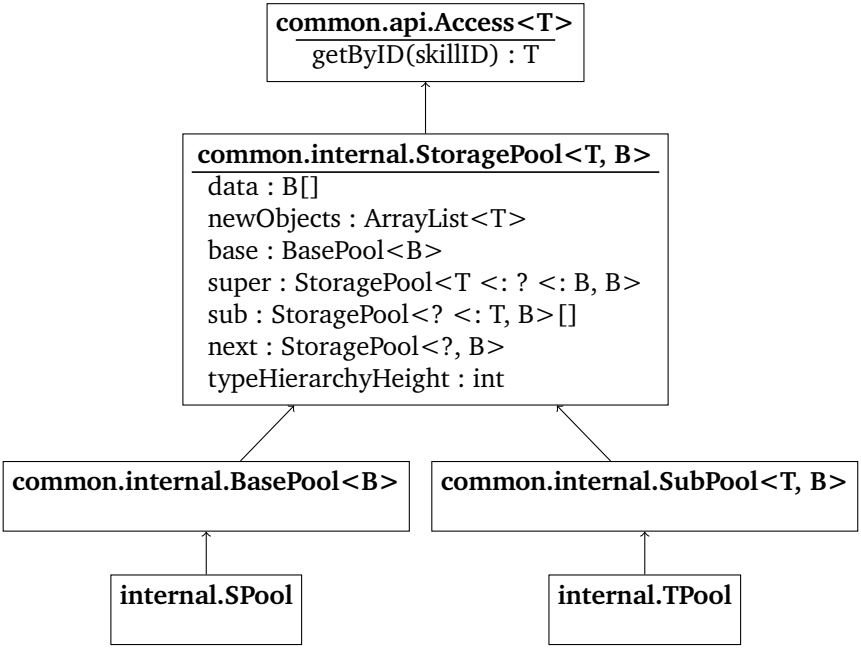



Abbildung 6.5.: Implementierung der Zugriffsobjekte/Speicherpools.

nach alle neuen Instanzen durch Traversierung der Subtypen erreichen. Will man hingegen die Instanzen eines beliebigen Typs traversieren ohne jede Referenz erst einer Typprüfung zu unterziehen und gegebenenfalls zu überspringen, so erfordert die Lösung weitere Informationen. Zunächst wird also die bisher nicht näher betrachtete `Access`-Struktur um intern genutzte strukturelle Informationen erweitert (siehe Abb. 6.5). Die Beschreibungen der Datenstrukturen verwenden dabei die generischen Typparameter `T` und `B`. Ein `T` wird dabei mit dem repräsentierten Typ instantiiert und ein `B` wird mit dem Basistyp des repräsentierten Typs instantiiert. Im Fall des Basispools sind die beiden Typen identisch.

Die in diesem Kapitel verwendeten Typoperatoren `<:`, `>:` und `?` bezeichnen eine transitiv symmetrische Subtypbeziehung, eine transitiv symme-

trische Supertypbeziehung und einen Typen, welcher die entsprechende Gleichung erfüllt. Der Operator `<:` entspricht `extends` in Java bis auf Symmetrie. Das bedeutet, dass beispielsweise die Gleichung $? <: T$ irgendeinen Typ bezeichnet, welcher selbst $T$ oder ein transitiver Subtyp von $T$ ist. Des Weiteren kann die Gleichung $T <: ? <: B$ genau mit jedem Element des Intervalls $[T;B]$ erfüllt werden.
Für Typparameter hier eingeführter Typen gelten folgende Eigenschaften:

- $\forall T,B.B <: \texttt{SkillObject} \wedge T <: B \Rightarrow$
  $\texttt{StoragePool} < T,B > <: \texttt{Access} < T >$
- $\forall B.B <: \texttt{SkillObject} \Rightarrow$
  $\texttt{BasePool} < B > <: \texttt{StoragePool} < B,B >$
- $\forall T,B.B <: \texttt{SkillObject} \wedge T <: B \Rightarrow$
  $\texttt{SubPool} < T,B > <: \texttt{StoragePool} < T,B >$

In Worten bedeuten diese Gleichungen, dass ein *T*-*B*-`StoragePool` von einem *T*-`Access` erbt, wobei in diesem Zusammenhang alle Instantiierungen von *B* transitiv symmetrische Subtypen von `SkillObject` und alle Instantiierungen von *T* transitiv symmetrische Subtypen von *B* sind; ein *B*-`BasePool` erbt von einem *B*-*B*-`StoragePool`, wobei in diesem Zusammenhang alle Instantiierungen von *B* transitiv symmetrische Subtypen von `SkillObject` sind; ein *T*-*B*-`SubPool` erbt von einem *T*-*B*-`StoragePool`, wobei in diesem Zusammenhang alle Instantiierungen von *B* transitiv symmetrische Subtypen von `SkillObject` und alle Instantiierungen von *T* transitiv symmetrische Subtypen von *B* sind.

Entsprechend gilt für die Pools der Beispieltypen, dass `internal.SPool` von `common.BasePool<S>` erbt. Analog erbt der Pool `internal.TPool` von `common.SubPool<T, S>`. Der Typparameter B wird hauptsächlich für die Typisierung das `data` Arrays benötigt.

In die Poolstruktur wird nun die Laufzeitvererbungshierarchie eingebaut. Die in diesem Kapitel entwickelten Iteratoren werden in Form des von Java bekannten APIs beschrieben, welches ein Prädikat `hasNext` definiert, welches genau dann gilt, wenn es zulässig ist, ein nächstes Element zu

```
val endHeight : Int
var current : StoragePool<?, B>

def hasNext() : Boolean = {
  return null != current
}

def next() : StoragePool<T, B> = {
  val r = current
  val n = current.nextPool
  if (null != n && endHeight < n.typeHierarchyHeight)
    current = n
  else
    current = null
  return (StoragePool<T, B>) r
}
```

Listing 6.1: Next/HasNext für TypeHierarchyIterator

besuchen, und eine Funktion `next`, welche das nächste Element zurückgibt und den Iterator ein Element weiter bewegt. Eine effiziente Implementierung dieses APIs erfordert, dass das Prädikat `hasNext` ausgesprochen effizient berechnet werden kann.

#### 6.3.3.1. `TypeHierarchyIterator` – alle Typen

Der erste Schritt in Richtung Implementierung der gewünschten Iteratoren besteht darin, einen effizienten Iterator über die Typhierarchie zu erzeugen. Dieser Iterator verwendet für die Traversierung von StoragePools deren Felder `next` und `typeHierarchyHeight`. Das Feld `next` zeigt auf den nächsten Pool in der Typordnung.[1] Das Feld `typeHierarchyHeight` enthält die Anzahl der Supertypen.[2] Der Zustand des Iterators besteht aus einer Endhöhe, welche der Typhöhe des initialen Pools entspricht, und einem Zei-

[1] Das Feld kann beim Allokieren der Pools effizient gesetzt werden.

[2] Effizient zu berechnen durch Erhöhung des Wertes des Supertypes um 1 zur Konstruktionszeit der Poolstruktur.

ger auf den aktuellen Pool, welcher mit dem initialen Pool initialisiert wird. Es gibt ein nächstes Element, falls der Zeiger auf den aktuellen Pool nicht `null` ist. Das nächste Element wird erreicht, indem man dem `next`-Zeiger des aktuellen Pools folgt. Führt dies zu einem aktuellen Pool, dessen Typhöhe kleiner oder gleich der initialen Höhe ist, so ist der aktuelle Pool auf `null` zu setzen. Die Typordnung der `next`-Zeiger garantiert, dass dieses Kriterium ausreichend ist. Dieser Iterator erlaubt eine rekursionsfreie Traversierung der Typhierarchie mit konstantem Speicheraufwand. Die Implementierung ist in Listing 6.1 veranschaulicht.

Alternativ dazu würde eine vergleichbare Traversierung über die `super`- und `sub`-Zeiger anstelle der Endhöhe den eigenen Index im `sub`-Array des Supertyps in jedem Pool erfordern, um mit konstantem Speicher- und Zeitaufwand traversieren zu können. Es müsste in diesem Fall aber im Schnitt zwei Zeigern pro Traversierungsschritt gefolgt werden, was weniger effizient ist. Der Zustand des Iterators würde dabei vergleichbar ausfallen, da man sich ebenso neben dem aktuellen Pool einen Abbruchwert merken müsste.

#### 6.3.3.2. `DynamicNewObjects` – dynamische neue Instanzen

Mithilfe des Typhierarchieiterators lässt sich direkt ein Iterator über alle neuen dynamischen Instanzen definieren. Das Feld `newObjects` enthält alle neuen statischen Instanzen eines Typs. Kombiniert man also den Typhierarchieiterator mit dem Iterator über `newObjects` des jeweils aktuellen Typs, so ergibt sich direkt und effizient das gewünschte Verhalten. Dieser Iterator liefert alle anzuhängenden Instanzen eines Typs in der passenden Reihenfolge. Die Implementierung ist in Listing 6.2 veranschaulicht.

#### 6.3.3.3. `DynamicObjects` – alle dynamischen Instanzen

Um Iteratoren über alle Instanzen definieren zu können, wird mehr Wissen über die Zusammensetzung von `data` benötigt. Da die Indizierung innerhalb von `data` der logischen Indizierung der gelesenen `.sf`-Datei entspricht, wird die Information der darin enthaltenen Blöcke zur Dekodierung benötigt. Ein

```
var ts : TypeHierarchyIterator
var is : Iterator<T>

def hasNext() : Boolean = {
  return null != is
}

def next() : T = {
  val result = is.next
  if (!is.hasNext) {
    var r : Iterator<T> = null
    while (null == r && ts.hasNext) {
      val t = ts.next
      if (t.newObjects.size != 0)
        r = t.newObjects.iterator
    }
    is = r
  }

  return result
}
```

Listing 6.2: Next/HasNext für DynamicNewObjects

Blockobjekt ist eine flache Struktur, welche den `BlockIndex`, den absoluten BPO[1] sowie die statische und dynamische Instanzzahl enthält. Blöcke werden im Pool in aufsteigender Reihenfolge gemäß `BlockIndex` gespeichert. Die statischen Instanzen des verwalteten Typs finden sich dann im Intervall [BPO, BPO + statische Anzahl]. Die dynamischen Instanzen finden sich im Intervall [BPO, BPO + dynamische Anzahl]. Mit ersterem Intervall kann man beispielsweise beim Lesen eines Zustands erforderliche Objektallokationen trivial parallelisieren. Mit letzterem Intervall kann der gerade gesuchte Iterator über alle Instanzen effizient implementiert werden.

[1] Der Base Pool Offset entspricht der Verschiebung der Indices relativ zum Basispool. Diese Verschiebung erfolgt nicht relativ zum Superpool, da dies entweder eine Bereinigung erfordern würde oder zu erhöhten Laufzeitkosten einiger Operationen, wie etwa der parallelen Allokation aller Instanzen, führen würde.

```
var ts : TypeHierarchyIterator
var secondIndex : Int
val lastBlock : Int
var index : Int
var end : Int
var newObjects : Iterator<T>

def hasNext : Boolean = { return null != ts }

def next() : T = {
  if (index < end) {
    val r = (T) p.data[index++]
    while (index == end && secondIndex < lastBlock) {
      val b = p.blocks[secondIndex]
      index = b.bpo
      end = index + b.dynamicCount
      secondIndex++
    }
    /* mode switch, if there is no other block */
    if (index == end) {
      secondIndex = 0
      while (ts.hasNext && null == newObjects) {
        val t = ts.next
        if (t.newObjects.size != 0)
          newObjects = t.newObjects.iterator
      }
      if (null == newObjects) ts = null
    }
    return r
  } else {
    val r = newObjects.next
    if (!newObjects.hasNext) {
      newObjects = null
      while (ts.hasNext && null == newObjects) {
        val t = ts.next
        if (t.newObjects.size != 0)
          newObjects = t.newObjects.iterator
      }
      if (null == newObjects) ts = null
    }
    return r
}}
```

Listing 6.3: Next/HasNext für DynamicObjects

Der Iterator über alle Instanzen eines Pools hat als Zustand den Pool, einen Typehierarchieiterator,[1] den aktuellen und letzten Blockindex, sowie den aktuellen und letzten Index. Die Traversierungsstrategie besteht darin, mittels der Blockindices die Blöcke und mittels der Indices innerhalb der Blöcke zu traversieren. Befindet man sich in einem Block, so ist das aktuelle Element $data[aktuellerIndex]$. Hat man den letzten Block erreicht, so werden die neuen Instanzen wie oben traversiert. Recycelt man für die Traversierung den aktuellen/letzten Index, so kann `hasNext` implementiert werden, indem man prüft ob die beiden Werte gleich sind. Der soeben beschriebene Iterator ist die Implementierung der Funktion `common.Access.all()`. Die Implementierung ist in Listing 6.3 veranschaulicht.

#### 6.3.3.4. `TypeOrderIterator` – dynamische Instanzen in Typordnung

Daneben existiert noch ein zweiter Iterator über alle Instanzen, welcher diese in Typordnung besucht. Dieser Iterator ist erforderlich, um beim Schreiben einer Datei effizient Objekt-IDs neu zu vergeben. Dieser entsteht durch Komposition des Typhierarchieiterators mit einem Iterator über die statischen Instanzen eines Typs. Letzterer lässt sich analog zum Iterator über dynamische Instanzen implementieren, indem man auf den Typhierarchieiterator verzichtet und anstelle der dynamischen die statische Anzahl in den Blöcken verwendet.

#### 6.3.3.5. Zugriff über typspezifischen logischen Index

Es ist denkbar, die Blockstruktur der `StoragePools` zu nutzen, um einen logischen Index zu implementieren, indem man solange über Blöcke iteriert und die dynamische Objektzahl des Blocks abzieht, bis sich daraus ein zulässiger Index ergibt und mithilfe des BPOs des Blocks dann auf das Objekt zugreift. Findet sich kein Index in einem Block, so ist unter Verwendung eines Typhierarchieiterators analog mit den `newObjects` Feldern zu verfahren.

[1] Tatsächlich könnte man anstelle des Typhierarchieiterators die Endhöhe speichern und die Logik von dort kopieren. Dies ist aber in den vermessenen Anbindungen nicht gemacht worden.

```
val ts : TypeHierarchyIterator
var is : StaticInstancesIterator<T>

def hasNext() : Boolean = {
  return null != is
}

def next() : T = {
  val result = is.next
  if (!is.hasNext) {
    var r : StaticDataIterator<T> = null
    while (null == r && ts.hasNext) {
      val t = ts.next
      if (t.staticSize != 0)
        r = new StaticDataIterator(t)
    }
    is = r
  }

  return result
}
```

Listing 6.4: Next/HasNext für TypeOrderIterator

Eine derartige Zugriffsmethode findet sich in keiner Anbindung mehr, da man hiermit leicht unnötig inperformanten Code schreiben kann.

### 6.3.4. Implementierung des Laufzeittypsystems

SKilL-Anbindungen implementieren ihren eigenen Reflection-Mechanismus, da die Zielsprache eventuell keinen eigenen zur Verfügung stellt oder dieser ungeeignet ist. Um Reflection seitens der Pools zu realisieren, fehlt lediglich eine Funktion, die den SKilL-Namen des repräsentierten Typs zurückgibt, sowie ein Iterator über alle Laufzeitrepräsentanten der Felder des Pools.[1] Für die praktische Arbeit ist es hilfreich, eine Hilfsfunktion zur Verfügung zu

[1] D.h. `Iterator<FieldDeclaration<?, T>> StoragePool<T, B>.fields()`

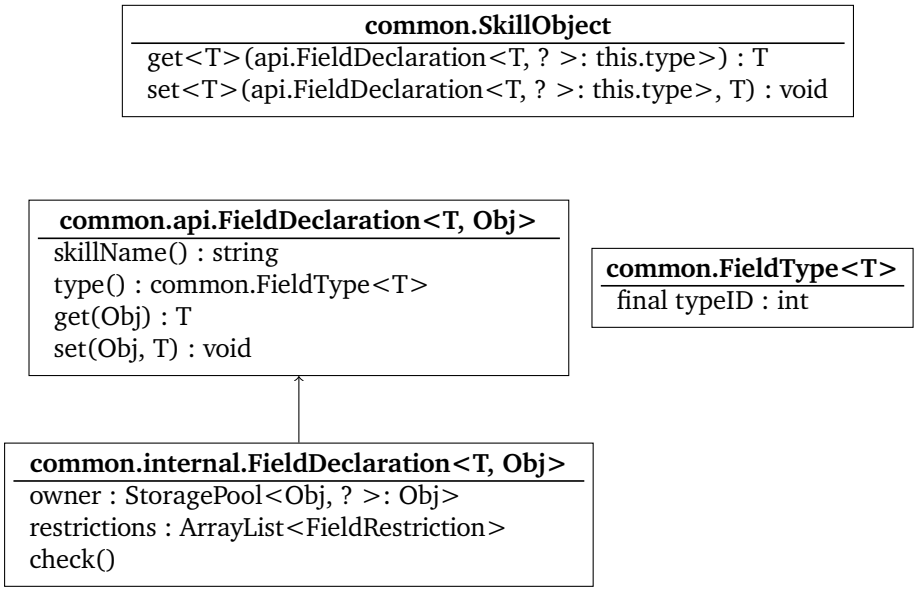



Abbildung 6.6.: Laufzeit-Reflection für Felder.

stellen, welche neben den eigenen Feldern auch die Felder der Superpools zurückgibt.[1] Der Pool fungiert selbst als Laufzeitrepräsentant seines Typs. Dies hat zur Folge, dass der Typ `common.Access` tatsächlich ein Subtyp von `common.FieldType` ist. Werden die Verweise auf Super- und Subtypen in `common.Access` durch Zugriffsfunktionen verfügbar gemacht, so sind aus Sicht des Werkzeugbauers alle relevanten Informationen vorhanden.

In Abb. 6.6 sind die für den Feldzugriff notwendigen Bestandteile der Architektur skizziert. An den etwas komplexer ausfallenden Signaturen der generischen Zugriffsmethoden in `SkillObject` ist bereits zu erkennen, dass sich das intendierte Verhalten in verbreiteten Typsystemen nicht beweisen und damit nicht direkt abbilden lässt. Die Idee der Signatur ist es, für ein Feldobjekt mit statisch beweisbar passendem Typ den entsprechen-

[1] D.h. `Iterator<FieldDeclaration<?, ? >: T>> StoragePool<T, B>.allFields()`

den Wert zu holen oder zu setzen. Da das Feld auch einem Supertyp des statischen Typs angehören kann, stößt man hier an die Grenzen der Möglichkeiten verbreiteter Typsysteme. Daher wird in allen Anbindungen der zweite Typparameter von `api.FieldDeclaration` durch `SkillObject` ersetzt. Das ist insofern unproblematisch, als die Parameter in den meisten Anwendungsszenarien ohnehin nicht statisch bekannt sind.

Für jedes statisch bekannte Feld wird eine Subklasse `common.internal.FieldDeclaration` generiert, die die Funktionen `get` und `set` implementiert. Für statisch unbekannte Felder, also solche, die in der Dateispezifikation aber nicht in der Werkzeugspezifikation enthalten sind, werden verteilte Felder verwendet, welche die Daten selbst verwalten.

Die Funktion `check` übernimmt die Prüfung der Restrictions des Feldes. Damit das auch für Felder möglich ist, die als Felder eines Objekts ausgeprägt sind, wird ein Zeiger auf den besitzenden Pool gespeichert. Hierüber kann ein Iterator über alle zu prüfenden Objekte erzeugt werden.

Feldtypen haben aus Sicht des Werkzeugbauers zunächst keine interessanten Eigenschaften. Für eingebaute Feldtypen existieren zur Laufzeit Repräsentanten. Ist der Feldtyp zustandsfrei, wie etwa der Repräsentant von `bool`, so wird dieser durch eine Konstante bzw. ein Singleton ausgeprägt. Für benutzerdefinierte Typen ist der Pool der Repräsentant. Dieser ist in der Vererbungshierarchie über `Access` eingeordnet, um dem Werkzeugbauer die Arbeit mit dem Laufzeittypsystem zu erleichtern. Analog dazu ist der `StringPool` der Repräsentant von `string`. Der Repräsentant von `annotation` ist zustandsbehaftet und wird daher einmal pro `SkillState` instantiiert. Ursächlich hierfür ist die Interaktion zwischen Laufzeittypsystem und Serialisierungscode auf die in §6.4 noch eingegangen wird.

### 6.3.5. Verteilte Felder

Der Fall nicht spezifizierter verteilter Felder wird von der Architektur bereits abgedeckt. Im Fall spezifizierter Felder ist vorgesehen, dass das generierte API unverändert bleibt, was bedeutet, dass ein Objekt auf den Laufzeitrepräsentanten des Feldes zugreifen können muss. Um dies zu gewährleisten,

wird in Objekten mit verteilten Feldern ein Verweis auf den besitzenden `SkillState` gespeichert. Von diesem aus kann der Pool mit einem Feldzugriff und das Feld mit einem weiteren Feldzugriff erreicht werden. Dies erfordert allerdings, dass für bekannte verteilte Felder ein entsprechendes Feld im Pool des Typs generiert wird. Der weitere Zugriff erfolgt dann wie im Fall unbekannter Felder. Es ist durchaus denkbar, anstelle des Verweises auf den `SkillState` einen Verweis auf den Pool zu verwenden, falls in einer Typhierarchie[1] alle verteilten Felder zum selben Pool gehören oder direkt auf das Feld zu verweisen, falls es nur ein verteiltes Feld in einer Typhierarchie gibt. Eine derartige Optimierung wurde jedoch nie implementiert, da sie die Code-Generierung in unangemessener Weise verkomplizieren würde.

Die Haltung der Felddaten im Feldrepräsentanten erfolgt in aller Regel in Form einer HashMap. Für aus der `.sf`-Datei gelesene Werte lässt sich hier über Optimierungen nachdenken. Die C++-Anbindung verwendet für kleine oder zusammenhängende Indexbereiche ein Array anstelle einer HashMap. Wurden die Objekte, welche zu den Felddaten gehören, alle in derselben Serialisierungsoperation der `.sf`-Datei hinzugefügt, so ist letztere Bedingung erfüllt. Dies gilt insbesondere immer dann, wenn die gelesene `.sf`-Datei aus einer Schreiboperation resultiert. Diese Optimierung ist insofern erwähnenswert, als es sich hierbei um den Großteil der unbekannten Felddaten handelt.

## 6.4. (De-)Serialisierung

Die Implementierung von Serialisierung und Deserialisierung kann im Wesentlichen in das Laufzeittypsystem ausgelagert werden. Für das Lesen wird eine Funktion `read` eingeführt, welche das Auslesen eines Datenblocks übernimmt. Für das Schreiben werden die Funktionen `offset` und `write` verwendet, welche im ersten Durchgang alle Offsets berechnen und in einem zweiten Durchgang einen Datenblock schreiben. In der Klasse `FieldDeclaration` bearbeiten diese drei Funktionen die gesamten Feld-

[1] Es wäre kontraproduktiv, in einem Objekt pro transitiven Supertyp, welcher ein verteiltes Feld enthält, einen Verweis zu speichern.

daten des repräsentierten Feldes. Außerdem gibt es in `FieldType` entsprechende Funktionen, die nur ein einziges Element innerhalb eines Datenblocks bearbeitet. Aus letzteren können zur Laufzeit die Funktionen für `FieldDeclarations` unbekannter Felder zusammengesetzt werden. Im Fall bekannter Felder wird von den Quellcode-Generatoren für diese drei Funktionen üblicherweise spezialisierter Code erzeugt, der es dem Compiler erlaubt, Code zu produzieren, der effizienter ist als die Standardimplementierung. Hier kann es schon ausreichend sein, den Typ des Feldes auf den korrekten Typ des Laufzeitrepräsentanten zu casten, wodurch es für den Compiler möglich ist zu erkennen, welche Funktion tatsächlich aufgerufen wird. Insbesondere sei hier darauf hingewiesen, dass die Implementierungen für Felder mit Referenztyp ohne weitere Aufrufe erfolgen kann. In Sprachen wie Java kann die Implementierung so erfolgen, dass der JIT-Compiler alle Aufrufe durch Inlining entfernen kann. Dies trägt wesentlich zu dem in §7.8 untersuchten guten Laufzeitverhalten bei.

### 6.4.1. Der Datei-Parser

Beim SKilL-Binärformat handelt es sich um ein kontextsensitives Format, welches ohne Lookahead[1] von links nach rechts in einem Durchgang gelesen werden kann.[2] Der Parse-Zustand besteht dabei aus dem `StringPool`, einer Liste von StoragePools, einer Map, welche Typnamen auf StoragePools abbildet, und dem Laufzeitrepräsentanten von `annotation` (`AnnotationType`).

Der `StringPool` ist erforderlich, um Strings, und damit auch Typnamen, aus der `.sf`-Datei zu lesen. Er hat außerdem die Aufgabe, Strings zu unifizieren, um die Speicheradresse des Strings für `hash`- und `equals`-Funktionen von HashMaps verwenden zu können. Falls der `StringPool` so implementiert ist, dass er Strings bedarfsorientiert auslesen kann und Felder

[1] Wenn man das Dateiende als Zustand des Eingabestroms und nicht als Sonderzeichen auffasst.

[2] Aus Effizienzgründen werden üblicherweise Teile übersprungen und erst nachdem alle Typinformationen bekannt sind die Daten verarbeitet, welche sich dann in der Mitte der `.sf`-Datei befinden können.

parallel deserialisiert[1] werden können, so ist das Lesen von Strings durch den `StringPool` vor parallelem Zugriff zu schützen, um die Integrität der Verwaltungsstrukturen im `StringPool` zu gewährleisten.

Die Liste von StoragePools wird für die Auflösung von *poolOffsets* und *typeIDs* benötigt. Die Map wird benötigt, um den eigentlichen Zustand zu konstruieren. Der `AnnotationType` wird benötigt, um Felder vom Typ `annotation` lesen zu können. Dieser muss Zeiger auf die StoragePoolListe und -Map mitführen, da die `typeID` eines serialisierten `annotation` Werts relativ zum Inhalt der `.sf`-Datei ist.

Zur Fehlererkennung und -meldung wird zudem die Menge der bereits gesehenen Typen innerhalb eines Blocks sowie der aktuelle Blockindex im Parse-Zustand verwaltet. Der eigentliche Lesevorgang ist eine weitgehend geradlinige Abbildung der in §5.2 dargestellten Struktur. Um `.sf`-Dateien effizient und parallel Lesen zu können, empfiehlt es sich, basierend auf `mmap` oder vergleichbarer Technologie, eine Art Token-Strom zu implementieren. Dieser kann für flache Typen, wie etwa v64, den Wert direkt aus der `.sf`-Datei lesen. Außerdem sollte er sich in mehrere parallel benutzbare Token-Ströme aufteilen lassen und Daten überspringen können. Hierdurch können für alle String- und Felddaten die entsprechenden Datensegmente an die für das Lesen verantwortlichen Laufzeitrepräsentanten weitergegeben werden.

Es verbleibt also noch die Interpretation der Typinformationen und der Abgleich dieser mit der Werkzeugspezifikation. Für jeden neuen Typ, den man in der `.sf`-Datei findet, wird ein StoragePool angelegt. Die hierfür verantwortliche Funktion muss generiert sein, da sie die der Werkzeugspezifikation entsprechenden generierten Pools instantiieren muss. Der Abgleich der Vererbungshierarchie lässt sich an dieser Stelle leicht integrieren. Ein vergleichbares Vorgehen wird für die Erzeugung von Feldrepräsentanten gewählt. Hier bietet der StoragePool eine Methode, um ein neues Feld anzulegen. Diese sorgt auch für eine Typprüfung.

Die Felddaten eines Blocks können wahlweise direkt oder nach dem Lesen aller Blöcke interpretiert werden. Es hat sich gezeigt, dass es bei einer

[1] Im Allgemeinen ist das der Fall, da der Werkzeugbauer parallel das Auslesen von bedarfsorientiert gelesenen Feldern erzwingen kann, ohne dass es für ihn direkt ersichtlich ist.

parallelen Deserialisierung im Allgemeinen effizienter ist, zuerst alle Typinformationen zu lesen. Hierfür müssen zunächst alle referenzierbaren Objekte allokiert werden. Um dies zu erreichen, werden im ersten Schritt alle `data` Arrays der StoragePools allokiert bzw. der Zeiger vom BasePool kopiert. Nun können für alle Pools und alle Blöcke parallel Instanzen erzeugt werden. Die einzig erforderliche Synchronisation ist eine Barriere am Ende der Allokation. Danach können für alle Felder und Blöcke die Felddaten parallel interpretiert werden. Dies erfordert wiederum nur eine Barriere am Ende der Operation. Im Fall von bei Bedarf gelesenen Feldern wird hier lediglich der Tokenstrom für das spätere Lesen zwischengespeichert.

Wurden alle Felddaten erfolgreich deserialisiert, kann der `SkillState` erzeugt und dem Werkzeugbauer übergeben werden. Dieser enthält nun alle aus der `.sf`-Datei gelesenen Objekte mit korrekt gesetzten Feldwerten. Da der `SkillState` zuletzt erzeugt wird, kann dieser einerseits noch alle fehlenden Laufzeittypinformationen erzeugen und andererseits die Referenzen auf Pools bekannter Typen durch konstante Zeiger realisieren.

### 6.4.2. Der Dateischreiber

Das Schreiben erfolgt im Wesentlichen analog zum Lesen. Eine erwähnenswerte Feinheit besteht in der Bestimmung der Anzahl dynamischer Instanzen eines Pools in Linearzeit relativ zur Zahl seiner transitiven Subtypen, da jeder Typ seine neuen Instanzen nur selbst kennt. Um in Linearzeit[1] schreiben zu können, wird im ersten Schritt die Größe aller Pools von unten nach oben berechnet. Um ein Mindestmaß an Benutzbarkeit zu gewährleisten, werden vor dem Schreiben alle über Reflection erreichbaren Strings dem StringPool hinzugefügt. Für Objekte ist das nicht erforderlich, da diese nicht durch Literale entstehen können und es daher eine vertretbare Forderung an den Werkzeugbauer ist, Objekte nicht selbst sondern über den `Access` zu allokieren.

Als nächstes muss der Zustand in die erforderliche Blockstruktur gebracht

[1] Tatsächlich können die Kosten hierfür in die zwingend erforderliche Serialisierung der Größe des jeweiligen Pools amortisiert werden.

```
val d = B[size()]
var pos = 0
for x in TypeOrderIterator(this) {
  if (0 != x.skillID) {
    d[pos] = x
    pos += 1
    x.skillID = pos
  }
  /* else: x wurde gelöscht */
}
data = d

for p in TypeHierarchyIterator(this) {
  [...]/* update block-Struktur für p */

  p.data = this.data
  p.newObjects.clear()
}
```

Listing 6.5: Etablierung der Block-Struktur für Pools

werden. Hierfür sind für alle Pools neue Blockinformationen zu berechnen und das `data` Array muss angepasst werden. Hierbei werden auch alle `skillIDs` korrigiert. Mithilfe der bereits eingeführten Iteratoren ist dies problemlos realisierbar. Am Ende der Operation müssen alle `newObjects` aller Pools geleert werden, da die Objekte nun nicht mehr neu sind. Die Implementierung ist in Listing 6.5 dargestellt. Der Code wird für jeden `BasePool` ausgeführt, um jedes Objekt nur einmal anzufassen. Handelt es sich um ein Anhängen anstelle eines Schreibens, so ist statt des `TypeOrderIterator` der `NewObjects`-Iterator zu verwenden.

Da Felder nur ihren eigenen `offset` berechnen, können diese Berechnungen parallel zueinander und zum Schreiben des Blocks erfolgen. Lediglich beim Schreiben des jeweiligen `offset` Werts muss auf die Berechnung gewartet werden. Sind die Typinformationen geschrieben, so kann der schreibende Tokenstrom aufgeteilt werden, um alle Felder parallel zu schreiben.

Die einzige erforderliche Synchronisation ist eine Barriere, die das Ende aller parallelen Schreibjobs erkennt, damit der Aufruf der Schreibfunktion nicht zurückkehrt, bevor die Datei vollständig geschrieben ist. Wenn sich die `offset` und `write` Funktionen bei der Wahl der Daten auf den letzten Block beziehen, so ist der Code für das Schreiben und der Code für das Anhängen identisch. Beim Anhängen ist lediglich die Auswahl der Typen und Felder sowie die Erzeugung der Blöcke etwas unterschiedlich.

Sind alle Felddaten geschrieben, so endet die Operation erfolgreich. Falls der Zustand danach nicht zerstört wird, müssen temporäre Zustände wie fixierte Pool-Größen bereinigt werden.

## 6.5. Löschen von Objekten

Das Löschen von Objekten ist im Kontext von Speicherverwaltung, bedarfsorientiert ausgewerteten Feldern und Vererbung etwas trickreich. Im Allgemeinen erfordert das Löschen eines Objekts, welches aus einer `.sf`-Datei gelesen wurde, die Anpassung des gesamten Zustands, da Objekt-IDs neu vergeben werden müssen. Da dies zu ineffizient ist, wird dem Werkzeugbauer nur ermöglicht, Objekte als zu löschen zu markieren. Diese werden dann bei der nächsten Schreiboperation gelöscht. Hierfür wird die `skillID` auf 0 gesetzt. Dies hat neben Idempotenz[1] den Vorteil, dass bei der Serialisierung ein Verweis auf das Objekt automatisch zu `null` wird. Dadurch ist die Konsistenz der geschriebenen `.sf`-Datei gewährleistet.

Es bleibt noch die korrekte Berechnung der Anzahl serialisierter Objekte, sowie die korrekte Neuberechnung des `data`-Arrays beim Schreiben. Hierfür muss jeder Pool die Zahl der gelöschten statischen Instanzen kennen. Würde man dem Werkzeugbauer erlauben, die Objekte über den `Access` zu löschen, wäre er mit dem Problem konfrontiert, dass statischer und dynamischer Typ übereinstimmen müssen, da sonst dem falschen Pool mitgeteilt werden würde, dass dessen Instanzen gelöscht werden. Eine geeignetere Implementierung ist daher, eine `delete`-Funktion in `api.SkillFile` zur

[1] D.h. es gibt keine Probleme durch mehrfache Freigabe.

Verfügung zu stellen, die anhand des Typs des zu löschenden Objekts automatisch den korrekten Pool informiert. Auf doppeltes Löschen wird adäquat reagiert, indem die `skillID` vorab gegen 0 geprüft wird. Dem Werkzeugbauer sollte dennoch in `SkillObject` eine Funktion zur Verfügung gestellt werden, mithilfe derer abgefragt werden kann, ob ein Objekt als zu löschen markiert wurde oder nicht.

## 6.6. Unbekannte Typen

Die Erweiterung von Typen um unbekannte Felder erfolgt durch Instantiierung bedarfsorientiert lesender Feldrepräsentanten. Die dabei erforderlichen Feldnamen und Typen werden der Dateispezifikation entnommen. Bei der Erweiterung der bekannten Typhierarchie um unbekannte Typen ist zu beachten, dass dies unter Erhalt der statischen Garantien des generierten APIs geschieht. Das bedeutet: Wird ein Typ `T` um einen unbekannten Subtyp `U` erweitert, so müssen alle `U`s einerseits vom Typ `T` sein, sich aber andererseits als `U`s identifizieren lassen und alle Eigenschaften einer `U` Instanz aufweisen.

Da es im Allgemeinen nicht möglich ist, zur Laufzeit Erweiterungen des Typsystems eines Programms vorzunehmen, müssen auch die Objekte Teil des SKilL-Laufzeittypsystems werden. Hierfür wird in `SkillObject` eine Methode `getTypeName` eingeführt, welche den Typnamen des zugehörigen `SkillTypes` zurück gibt. Für statisch bekannte Typen ist die Implementierung dieser Methode durch Rückgabe eines Literalwerts zu realisieren. Für jeden generierten Typ aus der Werkzeugspezifikation muss durch den Quellcode-Generator neben dem Typ selbst auch noch jeweils ein Typ für unbekannte Subtypen generiert werden. Dieser unbekannte-Subtyp-Klassen werden verwendet, um für jeden unbekannten Subtyp zur Laufzeit Instanzen zu erzeugen, welche der Vererbungshierarchie der Werkzeugspezifikation entsprechen. Instanzen unbekannter Subtypen halten einen Verweis auf ihren Pool. Mithilfe dieses Verweises lässt sich zum einen leicht über Reflection auf dessen Felder zugreifen, zum anderen kann der Name über diesen Verweis effizient berechnet werden.

Da diese unbekannten Subtypen beim Lesen einer `.sf`-Datei korrekt allokiert werden müssen, muss jeder `StoragePool` eine Funktion bereitstellen, die einen passenden unbekannten `SubPool` erzeugt. Weil ein unbekannter Typ keinen bekannten Supertyp haben muss, muss es zudem einen entsprechenden `BasePool` geben, welcher unbekannte Subtypen von `SkillObject` erzeugt.

Kapitel 7

# Evaluation

Dieses Kapitel befasst sich mit den Eigenschaften der in dieser Arbeit vorgeschlagenen Lösung. Dabei wird nicht nur der Status quo begründet und die generelle Anwendbarkeit demonstriert, sondern es werden auch Vergleiche mit verwandten Arbeiten durchgeführt und potentielle zukünftige Forschungsziele aufgezeigt. Dass die Anforderungen aus Kapitel 1.1 erfüllt wurden, ergibt sich bis auf drei Punkte direkt aus der Konstruktion von SKilL. Diese Punkte sind Dateigröße, Leserate und Benutzbarkeit. Der Punkt Benutzbarkeit ist in seiner Intention durch kurze Experimente mit Studenten nicht gut zu erfassen. Um diesen Punkt zu untersuchen, wurden studentische Arbeiten vergeben, die auf SKilL aufbauen. Hieraus lässt sich zumindest ableiten, dass es prinzipiell möglich ist, mit dem System in seiner vorliegenden Form zu arbeiten. Im Folgenden werden also die Dateigröße und Leserate, sowie andere damit verwandte Größen untersucht und mit existierenden Verfahren verglichen.

Dieses Kapitel ist nach einem einleitenden Teil in thematische Abschnitte unterteilt, die jeweils mit einer Zusammenfassung ihrer Experimente enden.

## 7.1. Allgemeiner Testaufbau

In diesem Abschnitt werden Eigenschaften, die allen oder vielen Messungen zugrunde liegen, zentral dargestellt. Die verwendete Bauhaus-Revision ist 33227. Die verwendete Revision des SKilL-Quellcode-Generators lautet `003e331e45251028b0b3db102403e09285a89552`. Die gemeinsam genutzten Bibliotheken von SKilL/C++ und SKilL/Ada werden nicht vom Quellcode-Generator ausgeliefert. Die für C++ verwendete Revision ist `6973ae1015540ab47cb88a771de7b580b2a6afe5`; für Ada wurde die Revision `68ff83db024d0508a2084ee626e6d9b47c11f4e5` verwendet.

Die Implementierung des SKilL-Quellcode-Generators hat in der verwendeten Revision 3761 Testfälle, welche überwiegend die Aufgabe haben, die Spezifikationskonformität der Implementierungen im Punkt Änderungstoleranz sicherzustellen. Hiervon entfallen 435 auf hier nicht genutzte Back-Ends. Weitere 6 sind von der Ausführung ausgeschlossen, da es sich hierbei um Tests für nicht implementierte Funktionalität handelt. Die verbleibenden 3320 Tests lassen sich alle wiederholbar erfolgreich durchführen.

### 7.1.1. Verwendete IR-Spezifikation

Bei der verwendeten IR-Spezifikation gibt es Freiheitsgrade, die einerseits durch ungenutzte Typdefinitionen entstehen, welche in einer `.skill`-Spezifikation nicht spezifiziert werden müssten und andererseits durch die Abbildung nicht in IML spezifizierter Typen.[1] Die verwendete IR-Spezifikation ist die in der entsprechenden Bauhaus-Revision im Werkzeug `iml2sf` hinterlegte `.skill`-Spezifikation.[2]

Die IR-Spezifikation ist im Wesentlichen das Ergebnis der von Dennis Przytarski [Prz16] entwickelten automatischen Abbildung von IML auf SKilL. Lediglich nicht spezifizierte Typen wurden teilweise anders abgebildet als es in der aktuellen Implementierung des Übersetzungswerkzeugs der Fall ist.

[1] Die IML-Spezifikationssprache erlaubt es, externe Typen zu definieren, deren Implementierung nicht vom Codegenerator sondern manuell erzeugt wird.

[2] Die Spezifikation unterliegt einer Geheimhaltungserklärung und kann daher nicht angehängt werden.

Das Übersetzungswerkzeug verwendet für diese handgeschriebene Definitionen. Da in den Bereichen SSA, Patterns und Threads noch Abbildungen existieren, deren Korrektheit nicht gezeigt werden konnte, und diese für unsere Messungen keine Relevanz haben, wurden diese entfernt bzw. durch eine Abbildung auf `annotation` ersetzt.

Ferner wurde eine Typdefinition entfernt, die Dennis Przytarski eingeführt hat, um sicherzustellen, dass im generierten API nicht verwendete Container-Typen dennoch existieren. Dieses Vorgehen führt an anderer Stelle zu einer vereinfachten Implementierung der Anbindung. Eine derartige Typdefinition bietet im Kontext dieser Evaluation keinen Mehrwert, da die entsprechende Typdefinition ohnehin nie instantiiert wird.

Um sicherzustellen, dass die Abbildung passend ist, wurde das Konvertierungswerkzeug `iml2sf` so implementiert, dass es eine Warnung erzeugt, falls Daten vorgefunden werden, die nicht abgebildet werden können. Dies ist über den Abgleich von IML- und SKilL-Reflection zur Laufzeit ohne Weiteres möglich. Bei der Konvertierung ist eine solche Warnung nicht aufgetreten. Daher ist davon auszugehen, dass es sich um eine vollständige und realistische[1] Konvertierung handelt.

### 7.1.2. Testumgebungen

Die Tests wurden auf drei Testsystemen durchgeführt. Deren Charakteristika werden in den Tabellen 7.1 bis 7.4 zusammengefasst. Bei der dargestellten Plattengeschwindigkeiten handelt es sich um Spitzenwerte, welche schon beim linearen Schreiben respektive Lesen großer Blöcke aus Nullen nicht zuverlässig erreicht werden. Die entsprechenden Systeme werden fortan *Laptop*, *Server*, *Server2* und *Workstation* genannt und durch kursive Schreibweise als Bezüge kenntlich gemacht.

Die Beschränkung auf Linux-Systeme mit relativ ähnlicher Software erlaubt den Austausch von Binaries unter den Systemen und damit einen

[1] Es wären auch Abbildungen denkbar, die die Messungen zugunsten von SKilL beeinflussen würden. Ein Beispiel hierfür wäre die direkte Abbildung von `Identifier` auf `string`, da sich in SKilL hierdurch direkt das gewünschte Verhalten zeigt und weniger Knoten zu verwalten wären.

| Name | Laptop |
|---|---|
| CPU | Intel Core i5-2540M (4x 2.6GHz) |
| RAM | 8 GB DDR3 @ 1333 MHz |
| Disk | 100 MB/s read, 80 MB/s write |
| | HITACHI HTS725032A9A364 |
| OS | Ubuntu 16.10 |
| Compiler | gcc 5.3.1, gnat pro 7.2.2, OpenJDK 1.8.121 |

Tabelle 7.1.: Das initiale Entwicklungssystem. Disk wurde mangels Datenblatt selbst vermessen.

| Name | Server |
|---|---|
| CPU | 4x AMD Opteron 6174 ($\Sigma$ 48x 2.2GHz) |
| RAM | 256 GB DDR3 @ 1333 MHz |
| Disk | Eurostor ES-6600D (Raid 5) |
| | 7x Toshiba MG03SCA200 |
| OS | Debian 8 |
| Compiler | gcc 4.9.2, gnat pro 7.2.2, OpenJDK 1.8.121 |

Tabelle 7.2.: Ein Server, der von mehreren Nutzern benutzt wird und dessen Disk ein RAID ist, welches von mehreren Servern geteilt wird und über 4GB Cache verfügt, was die Datenrate für unsere Messungen weitgehend unvorhersehbar macht.

| Name | Server2 |
|---|---|
| CPU | 4x Intel Xeon E5-4640 v4 ($\Sigma$ 96x 2.1GHz) |
| RAM | 128 GB DDR4 @ 2400 MHz |
| Disk | PERC H730 (Raid 1) |
| | 2x 400GB Intel S3610 SSD |
| OS | Debian 8 |
| Compiler | gcc 4.9.2, gnat pro 7.2.2, OpenJDK 1.8.121 |

Tabelle 7.3.: Ein Server, der als Messsystem konzipiert ist. Die Geschwindigkeit des RAIDs ist wegen des großen Caches kaum abschätzbar.

| Name | Workstation |
|---|---|
| CPU | Intel Core i7-5930K (12x 3.5GHz) |
| RAM | 32 GB DDR4 @ 2400 MHz |
| Disk | 540 MB/s read, 520 MB/s write |
| | Samsung SSD 840 EVO 500GB |
| OS | Ubuntu 16.10 |
| Compiler | gcc 5.3.1, gnat pro 7.2.2, OpenJDK 1.8.121 |

Tabelle 7.4.: Das aktuelle Entwicklungssystem.

besseren Vergleich der Hardwareeigenschaften. Die Frage der Plattformunabhängigkeit wurde bereits durch andere Projekte hinreichend bearbeitet. So gibt es beispielsweise auch Messungen unter Mac OS [Prz14]. Unter Windows wurde SKilL unter anderem während der Entwicklung einer IDE [BDF+16] von Studenten eingesetzt. Ebenso wurde die Speicherverbrauchsevaluation von Jonathan Roth [Rot15] unter Windows durchgeführt. Bei den eingesetzten Compilern handelt es sich um gnatpro 7.2.2 (Ada) bzw. OpenJDK 1.8.121 (Java/JVM). Davon abgesehen wurden die distributionsspezifischen (gcc) bzw. die vom Erstellungssystem (englisch build system) spezifizierten (Scala) Compiler verwendet. Der JVM-Heap ist bei allen Messungen auf allen Maschinen via `-Xmx8G` beschränkt. Die vermessenen Binaries sind bis auf C++-basierte zwischen allen Systemen identisch. Für C++ existiert jeweils eines für die Server und eines für den Rest, da diese nicht kompatibel sind.

Die meisten Messungen wurden auf *Workstation* durchgeführt, da dieser Rechner leicht von Zugriffen anderer Nutzer abgeschirmt werden kann und die längste Zeit am Stück für Messungen zur Verfügung steht.

### 7.1.3. Status verglichener SKilL-Anbindungen

In diesem Abschnitt wird kurz auf die Status der verglichenen Anbindungen eingegangen. Die Anbindungen an C und Haskell genügen den Anforderungen an eine Evaluation nicht und werden daher auch nicht weiter berücksichtigt.

Die Java-Anbindung verwendet eine etwas ältere Architektur, die sich vor allem dadurch unterscheidet, dass ein Hilfsarray für die Verwaltung der statischen Instanzen jedes Pools verwendet wird. Diese Architektur wurde im Rahmen der Evaluation nicht angepasst, da die Ergebnisse insgesamt noch ausreichend sind und die Implementierung in der Form in einigen Werkzeugen zum Einsatz kam, was das Vertrauen in ihre Funktionsfähigkeit enorm steigert. Die Ada-Implementierung geht davon aus, dass es auf einem Zustand nur eine Schreiboperation gibt. Ferner können im Fehlerfall Ressourcen verloren gehen, da diese nicht korrekt freigegeben werden. Die Scala-Implementierung kann als einzige beim Schreiben irrelevante Typen

erkennen und fügt diese nicht dem Dateitypsystem hinzu. Dadurch sind entstehende `.sf`-Dateien teilweise deutlich[1] kleiner. Diese Fähigkeit wurde nicht genutzt, um die vermessenen `.sf`-Dateien kleiner zu machen.[2]

Die C++-Implementierung wurde, von den hier durchgeführten Messungen abgesehen, kaum verwendet. Es ist daher davon auszugehen, dass man hier bei intensiver Nutzung auf Fehler oder eine teilweise unvollständige Implementierung stoßen würde. Dies ist bei den drei vorherigen Architekturen jeweils der Fall gewesen. Es ist jedoch nicht davon auszugehen, dass hierdurch eine Messung wesentlich beeinflusst werden würde, da einerseits alle Tests der Testsuite und andererseits alle Validierungen der Messungen erfolgreich ausgeführt wurden.

Tatsächlich wurde bei der Sichtprüfung der C++-Implementierung eine unvollständige bzw. fehlerhafte Behandlung von verteilten Feldern in speziellen Fällen entdeckt. Diese liegt auf einem, für die hier durchgeführten Messungen, nicht erreichbaren Pfad, sodass nicht zu erwarten ist, dass eine Behebung einen erkennbaren Einfluss auf die gemessenen Größen hat. Da dieser Umstand für eine korrekte Implementierung der Kernfunktionalität zu beseitigen ist, wird dies in Anhang C nachgereicht und der Einfluss auf die Messgrößen wird dort überprüft.

Die verwendeten Datenstrukturen für Containertypen und die Implementierung des Serialisierungscodes bekannter Felder unterscheiden sich je nach Anbindung im Detail.

### 7.1.4. Codemenge generierter SKilL-Anbindungen

Den Untersuchungen liegen Analysegraphen aus Bauhaus zugrunde, um auf möglichst realitätsnahen Daten aufzubauen. Um dem Leser ein Gefühl für die Struktur und Größe der daraus resultierenden Zwischendarstellung zu geben, sind diese Eigenschaft in diesem Abschnitt kurz zusammengefasst. Eine vorläufige Diskussion von Migrationsansätzen nach SKilL für Bauhaus findet

[1] Für `empty` ohne Analysen liegt der Faktor zwischen alter und neuer Dateigröße bei 0,73.

[2] Die entsprechenden `.sf`-Dateien entstehen ohnehin im Rahmen der Messung in §7.3.2. Da die Erkennung nur auf dem Vorhandensein von Instanzen beruht, hat dies auch keinen spürbaren negativen Einfluss auf die Gesamtlaufzeit.

| Art der Definition | Anzahl |
|---|---|
| interface | 17 |
| enum | 0 |
| regulär | 313 |
| ... davon abstrakt | 67 |
| typedefs | 27 |

| Tiefe | Anzahl | Tiefe | Anzahl |
|---|---|---|---|
| 0 | 5 | 5 | 97 |
| 1 | 2 | 6 | 77 |
| 2 | 1 | 7 | 33 |
| 3 | 12 | 8 | 22 |
| 4 | 54 | 9 | 10 |

Tabelle 7.5.: Links: Zahl der IML darstellenden Typdefinitionen nach Art der Typedefinition der verwendeten `.skill`-Spezifikation. Rechts: Verteilung der Vererbungstiefe spezifizierter Typen.

sich in [FW16], wo auch auf die zur Serialisierung benötigten Codemengen eingegangen wird. Da sich seitdem Architektur und Implementierung für einzelne Anbindungen noch verändert haben, kommt es zu Abweichungen der dargestellten Größen.

Die generierte Implementierung muss mindestens die verwendeten Typen kennen und diese wie spezifiziert repräsentieren können. Ferner bietet der generierte Code üblicherweise Methoden zum Traversieren und zur Manipulation von Typen, Speicherverwaltung, sowie Reflection.

Die Notwendigkeit unbekannte Typen zu bearbeiten, welche unter bekannten Typen durch Vererbung in der Typhierarchie eingebettet sind, erfordert im Wesentlichen eine Duplizierung der Typdarstellung, was bei der gegebenen Struktur der IR-Spezifikation (siehe Tab. 7.5) zu den in Tabelle 7.6 dargestellten Code-Mengen führt. Eigenheiten von Zielsprache und Architektur der Anbindungen sorgen für die Unterschiede zwischen den Anbindungen. Dabei ist zudem zu beachten, dass sich aufgrund unterschiedlicher Importmechanismen und Übersetzungsansätze die Aufteilung der Implementierung auf Dateien, im Hinblick auf eine Minimierung der Compilezeit, stark unterscheidet. Vergleicht man die aktuellen Daten mit den in [FW16] publizierten Daten, so ist vor allem die Anzahl der Dateien (früher 16, heute 783) auffällig. Hier wurden einzelne Typdefinitionen aus einer gemeinsamen Datei ausgelagert, um eine bessere Parallelisierung des Compilerlaufs zu erreichen.

| Anbindung | files | comment | code |
|---|---|---|---|
| SKilL/Ada | 1104 | 15963 | 269163 |
| SKilL/C++ | 33 | 7866 | 102253 |
| SKilL/Java | 1109 | 26619 | 119995 |
| SKilL/Scala | 783 | 9785 | 113799 |

Tabelle 7.6.: Generierte Code-Zeilen der SKilL-Anbindungen für IML nach Programmiersprache laut `cloc`.

Ferner bieten alle APIs leicht unterschiedliche Möglichkeiten mit Daten zu interagieren. So wird z.B. in Scala Code benötigt, um Patternmatching zu implementieren. In Ada wird eine Vielzahl an Hilfsfunktionen generiert, die Typkonversionen, insbesondere zwischen Zeigertypen, bieten.

### 7.1.5. Testdaten

In diesem Abschnitt werden kurz die verwendeten Testdaten aufgelistet. Es handelt sich hierbei um mehr oder minder aktuelle C-Programme. Die meisten davon sind als Open Source erhältlich.

Zwei der kleineren Testdaten sind handgeschriebene Programme. *empty.iml* ist der Übersetzung einer leeren C-Datei[1] entsprungen. *simple.iml* ist ein sehr einfacher Test, der ursprünglich für die Datarace Analyse vorgesehen war. Beide wurden aufgenommen, um den Bereich sehr kleiner Eingabedaten mit abzudecken, da diese gerade als Tests in der Entwicklung auftreten.

Zwei Werkzeuge liegen jeweils in zwei Versionen vor. Zum einen wurde *bash* neben einer alten Version um die Version 4.3 erweitert. Zum anderen wurde *php* zunächst in Version 5 übersetzt, da sich Version 7 mit Bauhaus nicht direkt verarbeiten ließ. Außerdem handelt es sich hierbei um leicht un-

[1] Das ist immer noch größer als ein Graph über ein leeres Programm, da hier etwas Standardcode enthalten ist. Dieser Standardcode umfasst beispielsweise Initialisierungs- und Finalisierungsroutinen. Diese enthalten zwar keine Anweisung, aber dennoch beispielsweise die Information zu welcher Übersetzungseinheit sie gehören. Es wäre für unsere Zwecke auch möglich mit einer leeren Datei zu messen. Diese Messung wäre aber nicht praxisrelevant und hätte den Nachteil, einige Invarianten von Bauhauswerkzeugen zu verletzen.

terschiedliche aber relativ große und damit für uns interessante Datensätze, weswegen jeweils beide Versionen verwendet wurden.

Da es sich bei unserer Messung nicht um eine syntaktische Analyse der Quellprogramme sondern um eine Untersuchung der Repräsentation des Analysegraphen handelt, ist die Zahl der Quellzeilen zunächst nicht von Belang. Um dem Leser ein Gefühl für die Größe des tatsächlich analysierten Programms zu geben, wurde dennoch für alle verwendeten Graphen die Zahl der referenzierten Quellzeilen (USLoC)[1] gemessen und neben der Dateigröße des Graphen nach Front-End ohne Analyseergebnisse als `.sf`-Datei aufgetragen (siehe Tabelle 7.7).[2] Diese Darstellung hat den Vorteil, dass sie auch verwendeten Code aus Bibliotheken mitberücksichtigt. Ein Nachteil ist allerdings, dass Code-Zeilen, die keinen Anfang eines Knotens enthalten, nicht berücksichtigt werden, was im Wesentlichen schließende Klammern von Blöcken sind. Daher sind die Daten nur bedingt mit rein syntaktischen Verfahren wie etwa `cloc`[Dan13] vergleichbar.

Weiterhin soll gezeigt werden, dass SKilL auch Analyseergebnisse neben dem Resultat des Binders effizient speichert. Dafür wurde zunächst die auf Andersen basierende Zeigeranalyse verwendet[And94; Fro06]. Darauf basierend wurde anschließend ein Kontrollflussgraph aufgebaut. Beide Werkzeuge arbeiten selbst in Bauhaus auf IML, die resultierenden Graphen werden nach Abschluss der Analyse nach SKilL konvertiert. Die Knoten- und Kantenzahlen der resultierenden Graphen sind in den Tabellen 7.8 und 7.9 dargestellt. Leider stellte sich heraus, dass die verfügbare Hardware bei einigen der großen Graphen nicht in der Lage ist, die Analyse erfolgreich durchzuführen, was in der Tabelle durch einen Strich dargestellt ist.

Um die Anwendbarkeit von SKilL auch auf Graphgrößen, bei denen manche Analyseimplementierungen schon versagen, demonstrieren zu können, wurde eine Analyse gesucht, die auch auf den größten Testdaten noch Ergeb-

[1] Die Berechnungsmethode von USLoCs ist in §7.5.1 samt berechnendem Werkzeug dargestellt. Die überlebenden Zeilen sind dabei abhängig von Entscheidungen, welche das Erstellungssystem trifft. Die vermessenen Austauschdateien sind in der digitalen Abgabe zu dieser Arbeit enthalten um eine Reproduzierbarkeit der Ergebnisse zu gewährleisten.

[2] Die Tabelle ist nach der Zahl der Knoten geordnet, um mit folgenden Tabellen vergleichbar zu sein.

| Dateiname | USLoC | Größe |
|---|---|---|
| empty | 3 | 0,017 |
| simple | 77 | 0,025 |
| sgfgen | 154 | 0,039 |
| bashversion | 266 | 0,055 |
| psize.aux | 519 | 0,073 |
| mksyntax | 367 | 0,073 |
| mksignames | 483 | 0,082 |
| mkeyes | 602 | 0,128 |
| gethost | 1066 | 0,139 |
| bf_test | 614 | 0,140 |
| time | 834 | 0,159 |
| mkbuiltins | 1128 | 0,218 |
| aget | 1420 | 0,278 |
| tstuu | 1779 | 0,312 |
| darkhttpd | 2154 | 0,425 |
| doc2gih | 4388 | 0,519 |
| dc | 2841 | 0,577 |
| units | 2889 | 0,643 |
| concepts | 3451 | 0,733 |
| uuname | 3522 | 0,767 |
| Astro | 2117 | 0,911 |
| smtprc | 3597 | 1,106 |
| mkpat | 4958 | 1,176 |
| gnuplot_x11 | 5711 | 1,207 |
| uulog | 5525 | 1,313 |
| joseki | 4926 | 1,394 |
| bc | 5412 | 1,294 |
| uuchk | 5816 | 1,415 |
| trueprint | 6652 | 1,434 |
| uupick | 6095 | 1,469 |

| Dateiname | USLoC | Größe |
|---|---|---|
| nano | 7259 | 1,713 |
| uuconv | 8270 | 1,948 |
| uucp | 8214 | 2,072 |
| uux | 8426 | 2,093 |
| grep | 8160 | 2,261 |
| make.new | 9256 | 2,267 |
| uustat | 9303 | 2,285 |
| sed | 9673 | 2,433 |
| uuxqt | 9792 | 2,450 |
| pgen | 9277 | 2,850 |
| cu | 12659 | 2,941 |
| bison | 15723 | 5,210 |
| uucico | 23197 | 5,887 |
| cook | 32121 | 6,722 |
| screen | 27299 | 7,190 |
| tcsh | 29991 | 7,842 |
| bash | 54740 | 14,428 |
| bluefish | 35863 | 16,970 |
| gnuplot | 52111 | 15,932 |
| bash43 | 68099 | 18,476 |
| gqview | 59579 | 23,531 |
| gnugo | 105407 | 46,811 |
| _freeze_importlib | 152843 | 58,946 |
| python | 156673 | 65,063 |
| _testembed | 156713 | 65,067 |
| php-cgi | 405659 | 277,570 |
| php | 407378 | 278,650 |
| php7-cgi | 405972 | 372,756 |
| php7 | 408485 | 374,168 |
| php7dbg | 415169 | 391,423 |

Tabelle 7.7.: Benutzte Testdateien nach Knotenzahl geordnet. Die Größe der `.sf`-Datei ist in Megabyte angegeben.

nisse produziert. In Bauhaus existiert eine Steensgaard-Implementierung [Ste96], die diese Anforderung erfüllt. Die Größen der entsprechenden Graphen sind in den Tabellen 7.10 und 7.11 dargestellt.

| Name | Gebunden | | + Andersen | | + CFG | |
|---|---|---|---|---|---|---|
| | Knoten | Kanten | Knoten | Kanten | Knoten | Kanten |
| empty | 732 | 200 | 745 | 216 | 762 | 302 |
| simple | 1.409 | 1.818 | 1.434 | 1.895 | 1.467 | 2.263 |
| sgfgen | 2.515 | 4.810 | 2.535 | 4.855 | 2.583 | 6.047 |
| bashversion | 3.578 | 7.597 | 3.619 | 7.717 | 3.712 | 9.690 |
| psize | 4.783 | 9.505 | 4.802 | 9.546 | 4.829 | 9.942 |
| mksyntax | 5.055 | 12.146 | 5.177 | 12.867 | 5.301 | 16.463 |
| mksignames | 5.793 | 13.076 | 5.910 | 13.508 | 5.973 | 16.177 |
| mkeyes | 8.911 | 23.548 | 8.974 | 23.758 | 9.146 | 31.040 |
| gethost | 9.772 | 22.606 | 9.841 | 22.858 | 9.988 | 26.900 |
| bf_test | 10.313 | 28.187 | 10.513 | 29.394 | 10.758 | 37.699 |
| time | 11.144 | 30.727 | 11.346 | 31.601 | 11.635 | 40.898 |
| mkbuiltins | 15.690 | 45.869 | 16.166 | 48.367 | 16.734 | 65.570 |
| aget | 20.687 | 58.232 | 20.910 | 59.166 | 21.289 | 73.313 |
| tstuu | 23.657 | 66.887 | 24.031 | 68.528 | 24.744 | 91.512 |
| darkhttpd | 30.821 | 89.841 | 31.408 | 93.579 | 32.343 | 125.253 |
| doc2gih | 32.489 | 84.193 | 40.544 | 124.392 | 40.654 | 151.099 |
| dc | 41.233 | 127.322 | 42.455 | 174.511 | 44.213 | 225.052 |
| units | 45.718 | 137.633 | 46.674 | 144.704 | 48.246 | 198.036 |
| concepts | 51.121 | 159.088 | 52.676 | 167.447 | 54.473 | 225.798 |
| uuname | 53.362 | 169.583 | 54.465 | 175.465 | 56.159 | 222.291 |
| Astro | 58.559 | 175.504 | 59.712 | 182.599 | 60.844 | 242.913 |
| smtprc | 73.805 | 229.443 | 74.524 | 233.008 | 76.215 | 313.839 |
| mkpat | 78.631 | 240.379 | 80.388 | 250.274 | 83.159 | 343.186 |
| gnuplot_x11 | 80.787 | 245.837 | 82.424 | 256.062 | 84.846 | 342.545 |
| uulog | 87.985 | 286.004 | 90.288 | 304.254 | 93.121 | 391.412 |
| joseki | 89.020 | 298.366 | 91.075 | 312.139 | 95.431 | 442.448 |
| bc | 91.118 | 274.868 | 92.864 | 348.028 | 95.718 | 453.037 |
| uuchk | 93.892 | 307.706 | 96.340 | 329.079 | 99.481 | 427.632 |
| trueprint | 96.904 | 298.396 | 99.132 | 311.871 | 103.198 | 431.305 |
| uupick | 98.186 | 319.061 | 100.680 | 338.501 | 103.817 | 434.610 |

Tabelle 7.8.: Graphstruktur der verwendeten Testdaten (Andersen, kleine Graphen). Kanten sind alle Referenzen ohne *null*, Knoten ist alles, was an Kanten beteiligt ist.

| Name | Gebunden | | + Andersen | | + CFG | |
|---|---|---|---|---|---|---|
| | Knoten | Kanten | Knoten | Kanten | Knoten | Kanten |
| nano | 113.722 | 353.835 | 115.551 | 367.410 | 119.798 | 508.452 |
| uuconv | 129.229 | 415.540 | 132.339 | 444.888 | 136.873 | 591.492 |
| uucp | 137.127 | 444.963 | 140.404 | 469.208 | 144.789 | 606.495 |
| uux | 138.603 | 449.065 | 141.903 | 474.514 | 146.465 | 616.006 |
| grep | 142.828 | 469.718 | 145.401 | 498.040 | 152.825 | 722.286 |
| make | 146.705 | 465.879 | 150.801 | 552.702 | 157.437 | 750.575 |
| uustat | 150.466 | 487.647 | 154.128 | 520.051 | 159.224 | 677.750 |
| sed | 155.149 | 503.120 | 158.339 | 527.324 | 164.966 | 739.652 |
| uuxqt | 160.933 | 521.446 | 164.729 | 550.707 | 169.980 | 716.758 |
| pgen | 173.611 | 584.255 | 175.341 | 607.039 | 178.363 | 694.595 |
| cu | 191.014 | 615.394 | 195.818 | 648.927 | 202.553 | 858.416 |
| bison | 307.683 | 1.036.575 | 316.142 | 1.365.968 | 328.280 | 1.796.351 |
| uucico | 360.224 | 1.162.287 | 369.260 | 1.825.995 | 381.884 | 2.231.901 |
| cook | 417.869 | 1.314.645 | 432.939 | 7.554.127 | 448.136 | 8.045.157 |
| screen | 441.453 | 1.397.653 | – | – | – | – |
| tcsh | 477.925 | 1.531.306 | 488.700 | 2.471.655 | 508.890 | 3.081.442 |
| bash | 857.432 | 2.786.583 | 890.217 | 49.767.574 | 934.185 | 50.961.942 |
| bluefish | 935.816 | 3.130.796 | 948.236 | 3.193.372 | 959.395 | 3.850.182 |
| gnuplot | 940.825 | 3.043.739 | 969.892 | 6.514.440 | 1.005.685 | 7.781.244 |
| bash43 | 1.094.515 | 3.567.206 | 1.140.355 | 136.864.136 | 1.198.355 | 138.420.240 |
| gqview | 1.357.794 | 4.542.444 | 1.381.766 | 5.214.757 | 1.404.992 | 6.158.210 |
| gnugo | 2.758.624 | 9.192.389 | 2.829.084 | 9.800.647 | 2.876.332 | 13.275.682 |
| _freeze_importlib | 3.297.987 | 11.543.649 | – | – | – | – |
| python | 3.572.725 | 12.357.817 | – | – | – | – |
| _testembed | 3.572.874 | 12.358.310 | – | – | – | – |
| php-cgi | 14.607.239 | 48.612.980 | – | – | – | – |
| php | 14.658.935 | 48.794.576 | – | – | – | – |
| php7-cgi | 18.736.674 | 64.381.245 | – | – | – | – |
| php7 | 18.805.521 | 64.610.412 | – | – | – | – |
| php7dbg | 19.609.485 | 67.500.947 | – | – | – | – |

Tabelle 7.9.: Graphstruktur der verwendeten Testdaten (Andersen, große Graphen). Kanten sind alle Referenzen ohne *null*, Knoten ist alles, was an Kanten beteiligt ist.

| Name | Gebunden | | + Steensgaard | | + CFG | |
|---|---|---|---|---|---|---|
| | Knoten | Kanten | Knoten | Kanten | Knoten | Kanten |
| empty | 732 | 200 | 747 | 228 | 764 | 314 |
| simple | 1.409 | 1.818 | 1.443 | 1.916 | 1.476 | 2.284 |
| sgfgen | 2.515 | 4.810 | 2.570 | 4.987 | 2.618 | 6.179 |
| bashversion | 3.578 | 7.597 | 3.648 | 7.803 | 3.741 | 9.776 |
| psize | 4.783 | 9.505 | 4.821 | 9.609 | 4.848 | 10.005 |
| mksyntax | 5.055 | 12.146 | 5.176 | 12.544 | 5.300 | 16.140 |
| mksignames | 5.793 | 13.076 | 5.896 | 13.482 | 5.959 | 16.151 |
| mkeyes | 8.911 | 23.548 | 9.088 | 24.348 | 9.260 | 31.630 |
| gethost | 9.772 | 22.606 | 9.859 | 22.902 | 10.006 | 26.944 |
| bf_test | 10.313 | 28.187 | 10.595 | 29.141 | 10.840 | 37.446 |
| time | 11.144 | 30.727 | 11.370 | 31.498 | 11.659 | 40.795 |
| mkbuiltins | 15.690 | 45.869 | 16.103 | 47.415 | 16.671 | 64.618 |
| aget | 20.687 | 58.232 | 21.090 | 59.630 | 21.469 | 73.777 |
| tstuu | 23.657 | 66.887 | 24.055 | 68.286 | 24.768 | 91.270 |
| darkhttpd | 30.821 | 89.841 | 31.542 | 92.652 | 32.477 | 124.326 |
| doc2gih | 32.489 | 84.193 | 32.585 | 84.503 | 32.695 | 111.210 |
| dc | 41.233 | 127.322 | 42.667 | 132.708 | 44.425 | 183.249 |
| units | 45.718 | 137.633 | 46.826 | 142.020 | 48.398 | 195.352 |
| concepts | 51.121 | 159.088 | 53.042 | 166.565 | 54.839 | 224.916 |
| uuname | 53.362 | 169.583 | 54.734 | 174.517 | 56.428 | 221.343 |
| Astro | 58.559 | 175.504 | 60.240 | 181.530 | 61.372 | 241.844 |
| smtprc | 73.805 | 229.443 | 74.756 | 234.337 | 76.447 | 315.168 |
| mkpat | 78.631 | 240.379 | 80.600 | 248.546 | 83.371 | 341.458 |
| gnuplot_x11 | 80.787 | 245.837 | 82.630 | 252.764 | 85.052 | 339.247 |
| uulog | 87.985 | 286.004 | 90.274 | 294.943 | 93.107 | 382.101 |
| joseki | 89.020 | 298.366 | 91.932 | 310.675 | 96.288 | 440.984 |
| bc | 91.118 | 274.868 | 92.899 | 282.220 | 95.753 | 387.229 |
| uuchk | 93.892 | 307.706 | 96.275 | 317.459 | 99.416 | 416.012 |
| trueprint | 96.904 | 298.396 | 98.819 | 305.991 | 102.885 | 425.425 |
| uupick | 98.186 | 319.061 | 100.749 | 328.945 | 103.886 | 425.054 |

Tabelle 7.10.: Graphstruktur der verwendeten Testdaten (Steensgaard, kleine Graphen). Kanten sind alle Referenzen ohne *null*, Knoten ist alles, was an Kanten beteiligt ist.

| Name | Gebunden | | + Steensgaard | | + CFG | |
|---|---|---|---|---|---|---|
| | Knoten | Kanten | Knoten | Kanten | Knoten | Kanten |
| nano | 113.722 | 353.835 | 116.011 | 362.421 | 120.258 | 503.463 |
| uuconv | 129.229 | 415.540 | 132.124 | 427.810 | 136.658 | 574.414 |
| uucp | 137.127 | 444.963 | 140.531 | 458.004 | 144.916 | 595.291 |
| uux | 138.603 | 449.065 | 141.982 | 462.013 | 146.544 | 603.505 |
| grep | 142.828 | 469.718 | 145.902 | 481.834 | 153.326 | 706.080 |
| make | 146.705 | 465.879 | 149.818 | 479.312 | 156.454 | 677.185 |
| uustat | 150.466 | 487.647 | 154.180 | 501.919 | 159.276 | 659.618 |
| sed | 155.149 | 503.120 | 159.405 | 520.885 | 166.032 | 733.213 |
| uuxqt | 160.933 | 521.446 | 164.890 | 536.770 | 170.141 | 702.821 |
| pgen | 173.611 | 584.255 | 182.368 | 612.714 | 185.390 | 700.270 |
| cu | 191.014 | 615.394 | 195.808 | 633.842 | 202.543 | 843.331 |
| bison | 307.683 | 1.036.575 | 317.690 | 1.077.621 | 329.828 | 1.508.004 |
| uucico | 360.224 | 1.162.287 | 369.114 | 1.197.234 | 381.738 | 1.603.140 |
| cook | 417.869 | 1.314.645 | 430.378 | 1.362.298 | 445.575 | 1.853.328 |
| screen | 441.453 | 1.397.653 | 449.613 | 1.437.329 | 471.127 | 2.048.906 |
| tcsh | 477.925 | 1.531.306 | 488.873 | 1.570.958 | 509.063 | 2.180.745 |
| bash | 857.432 | 2.786.583 | 879.177 | 2.865.956 | 923.145 | 4.060.324 |
| bluefish | 935.816 | 3.130.796 | 948.461 | 3.178.382 | 959.620 | 3.835.192 |
| gnuplot | 940.825 | 3.043.739 | 959.082 | 3.111.218 | 994.875 | 4.378.022 |
| bash43 | 1.094.515 | 3.567.206 | 1.121.409 | 3.667.725 | 1.179.409 | 5.223.829 |
| gqview | 1.357.794 | 4.542.444 | 1.380.469 | 4.625.598 | 1.403.695 | 5.569.051 |
| gnugo | 2.758.624 | 9.192.389 | 2.834.603 | 9.487.329 | 2.881.851 | 12.962.364 |
| _freeze_importlib | 3.297.987 | 11.543.649 | 3.392.646 | 11.907.571 | 3.528.515 | 16.185.009 |
| python | 3.572.725 | 12.357.817 | 3.666.940 | 12.717.213 | 3.802.335 | 17.275.100 |
| _testembed | 3.572.874 | 12.358.310 | 3.667.106 | 12.717.749 | 3.802.519 | 17.275.999 |
| php-cgi | 14.607.239 | 48.612.980 | 14.891.852 | 49.746.098 | 15.355.596 | 67.418.937 |
| php | 14.658.935 | 48.794.576 | 14.944.702 | 49.932.972 | 15.411.229 | 67.681.875 |
| php7-cgi | 18.736.674 | 64.381.245 | 19.230.268 | 66.281.411 | 20.166.601 | 94.969.925 |
| php7 | 18.805.521 | 64.610.412 | 19.300.921 | 66.518.171 | 20.240.881 | 95.310.597 |
| php7dbg | 19.609.485 | 67.500.947 | 20.131.373 | 69.508.045 | 21.124.588 | 99.645.467 |

Tabelle 7.11.: Graphstruktur der verwendeten Testdaten (Steensgaard, große Graphen). Kanten sind alle Referenzen ohne *null*, Knoten ist alles, was an Kanten beteiligt ist.

### 7.1.6. Wahl des Regressionsmodells

Die erwartete Ressourcennutzung der in diesem Kapitel vermessenen Werkzeuge ist grundsätzlich linear oder quasi[1] linear abhängig von der Größe[2] des Graphen. Daher wird ein lineares Regressionsmodell verwendet, um diese Erwartungen zu prüfen. Da die hier verwendeten Messgrößen nicht negativ sein können und einen erwartbaren konstanten Anteil haben, werden die Regressionen im Normalfall mit Konstante durchgeführt. Enthält das Ergebnis einer Regressionsanalyse einen negativen konstanten Anteil, so ist dies entweder auf einen Fehler, oder auf eine irgendwie geartete superlineare Ressourcennutzung zurückzuführen.

Die Darstellung der Ergebnisse der Regressionsanalyse erfolgt in Tabellenform. Da zumeist mehrere Alternativen verglichen werden, werden diese in Spalten dargestellt. Die erste Zeile enthält die linear abhängige Komponente, eine Fehlerabschätzung sowie eine Einschätzung der Signifikanz des berechneten Wertes. Die zweite Zeile enthält dieselben Informationen für die konstante Komponente. Darunter folgen zur Beurteilung der Regression jeweils die Zahl der verwendeten Datenpunkte, ein normiertes Korrelationsmaß[3] ($R^2$), und ein Test der prinzipiellen Zulässigkeit[4] der Regression (F Statistic). Die Erzeugung der Regressionstabellen erfolgt mithilfe der GNU R Bibliothek stargazer[5] und kann nur bedingt beeinflusst werden.

### 7.1.7. Die Ausreißer – Bash unter Andersen

Bei aufmerksamer Betrachtung von Tabelle 7.9 stellt man fest, dass sowohl *bash* als auch *bash43* eine für ihre Knotenzahl ungewöhnlich hohe Kanten-

[1] Es werden beispielsweise hashbasierte Container verwendet, deren erwartete konstante Zugriffskosten in der Praxis zu zeigen wären.

[2] Die Größe entspricht hier zunächst der Summe aus Knoten und Kanten. Diese beiden Größen sind in unseren Graphen aber weitestgehend austauschbar, da sie sich zumeist nur um einen kleinen konstanten Faktor unterscheiden.

[3] Der Wert liegt zwischen 0 und 1, wobei 1 einer perfekten Korrelation entspricht.

[4] Etwas vereinfacht dargestellt, müsste man das Regressionsmodell prüfen, würde dieser Wert unter 2 liegen. Da wir eine große Bandbreite an Messpunkten haben sind die Regressionen hier stets zulässig.

[5] Hlavac, Marek (2015). stargazer: Well-Formatted Regression and Summary Statistics Tables. R package version 5.2. http://CRAN.R-project.org/package=stargazer

zahl aufweisen. Die meisten Graphen haben ein Kanten/Knoten-Verhältnis zwischen 2 und 4. Bei *bash* hingegen liegt es bei über 50 und bei *bash43* sogar bei über 100. In folgenden Analysen zeigt sich, dass diese Austauschdateien bei nahezu allen Messungen die einzigen Ausreißer darstellen. Diese sind leicht zu erkennen, da die meisten Diagramme über die Knotenzahl aufgetragen sind und sie daher aus einem oft linearen Verhalten weit ausbrechen. Ursächlich hierfür ist, dass die meisten Messungen tatsächlich linear in der Menge des verwendeten Hauptspeichers oder der Dateigröße sind, welche bei einer stark gestiegenen Kantenzahl offensichtlich nicht mehr ähnlich zu den Daten vergleichbarer Knotenzahlen ist.

Würde man die Diagramme stattdessen über die Dateigröße auftragen, so würden die Ausreißer zwar weitgehend verschwinden, das Verhalten wäre aber praxisfremd. Tatsächlich interessiert sich ein Werkzeugnutzer in der Regel auch dann für die Laufzeit, wenn er etwas Ungewöhnliches macht. Zudem sieht man deutlich, dass SKilL keine Hürde ist, etwas Ungewöhnliches zu machen, was für eine breite Anwendbarkeit des Verfahrens spricht. Dass dadurch bei den meisten Messungen Korrelation und Konfidenzintervalle verwaschen werden, darf kein Argument sein, die Ausreißer zu entfernen. Es ist durchaus zu erwarten, dass man bei der Entwicklung z.B. durch Programmierfehler oder eine fragwürdige[1] Modellierung der Zwischendarstellung entsprechende Graphen erhält.

### 7.1.8. Messfehler und Messmethoden

In diesem Abschnitt wird kurz begründet wie und warum bestimmte Messverfahren und Auswertungsmethoden verwendet werden. Zur Illustration des gewählten Vorgehens wird die in Abschnitt 7.3 durchgeführte Messung des Verhältnisses zwischen Lesezeit und Schreibzeit mithilfe des Werkzeugs `recode` für die C++-Implementierung statt den üblichen 10 mal 1000 mal

[1] Hierbei handelt es sich beim Gros der Kanten um Zwischenergebnisse der Zeigeranalyse, die man nicht serialisieren sollte. Offenbar besteht hier weiterer Forschungsbedarf im Rahmen der Darstellung der Andersen-Zeigeranalyse in Bauhaus.

wiederholt.[1] Als Testdatensatz wird `aget` verwendet, da es sich hierbei um eine kleines nicht-triviales Programm handelt (siehe Tabelle 7.7). Die in diesem Abschnitt erhobenen Daten wurden Anfang September 2016 mit etwas älteren Versionen von SKilL und Bauhaus erhoben. Da es sich hierbei nicht um eine Untersuchung des entwickelten Systems, sondern der Messmethodik selbst handelt und eine vollständige Neuerhebung Hardwareanpassungen des Messsystems erfordert hätte, wurde auf eine Neuerhebung der Zwischenschritte verzichtet.

Zunächst wurden Ausführungszeiten mittels des Bash-Werkzeugs `time` ermittelt, da hierdurch der Gesamtaufwand konservativ abgeschätzt werden kann. Ferner lassen sich so z.B. die Kosten von Objektallokation nicht verstecken, indem man Objekte in einem globalen Cache vorallokiert.

Da es sich bei den ausgewählten Daten um echte Datensätze handelt, führen ihre individuellen Eigenheiten zu einer ganz natürlichen Streuung der Messwerte. Obwohl sich um eine möglichst gute Abschottung der Testsysteme gegen äußere Störungen bemüht wurde, können diese nicht ganz ausgeschlossen werden. Die Störungen haben dabei keinen erkennbar strukturierten Effekt auf die Messung, sprich die Messungen folgen keiner erkennbaren Verteilung. Die Histogramme in Abbildung 7.1 zeigen, dass sich etwa die Hälfte der Messungen eng um den Mittelwert gruppieren und der Rest vergleichsweise gleichmäßig und weit streut. Bei der Betrachtung des Histogramms sei angemerkt, dass die Ränder kaum zu erreichen sind, da z.B. ein Wert von 0.909 bedeutet, dass das Lesen einer Austauschdatei 10 mal länger benötigt hat, als das Schreiben eines Äquivalents. Ein Wert von 0.5 bedeutet, dass beide Operationen gleich schnell waren. Eine naheliegende Interpretation des Histogramms ist, dass es sich bei den streuenden Werten um irgendwie geartete Unterbrechungen des gemessenen Prozesses handelt. Für diese These spricht die Verteilung der Ausführungszeiten über die Verteilung des lesenden Anteils, die in Abbildung 7.2 dargestellt ist.

In dieser Abbildung kann man an der Zeilenbildung der Messpunkte erkennen, dass man für kleine Eingaben das Problem der Auflösung des

[1] Außerdem wurde auf die hier ohnehin unnötige Verifikation jedes einzelnen Datenpunkts verzichtet, um die Laufzeit im Rahmen zu halten.

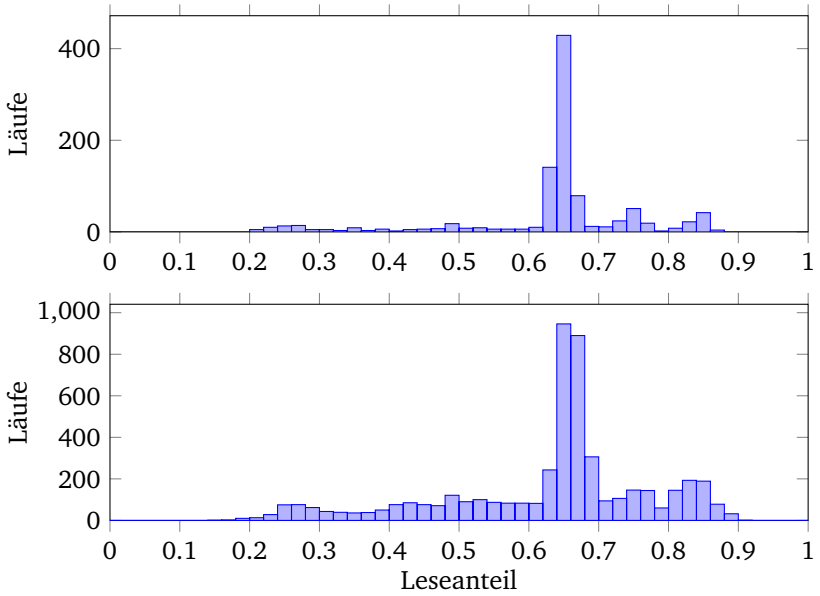



Abbildung 7.1.: `recode`-Messungen, 1000 Wiederholungen, Histogramm über den Anteil der zum Lesen verwendeten Zeit. Oben nur `aget` Front-End Daten (in Summe 1000 Läufe), unten inklusive Analyseergebnisse (in Summe 5000 Läufe).

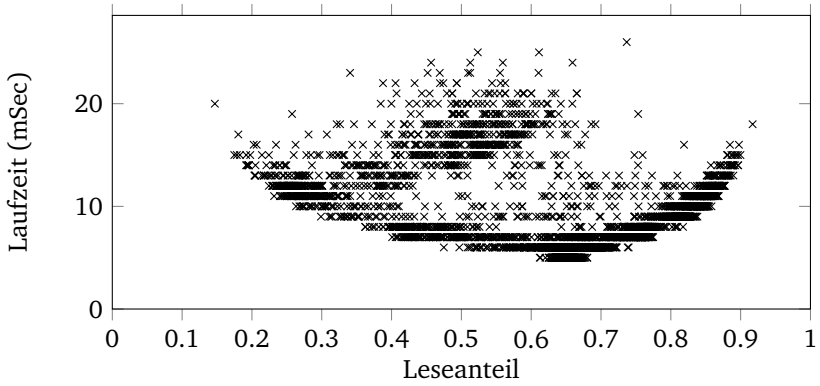



Abbildung 7.2.: `recode`-Messungen, 1000 Wiederholungen, Ausführungszeit über Anteil der Leseoperation. `aget` inklusive Derivate.

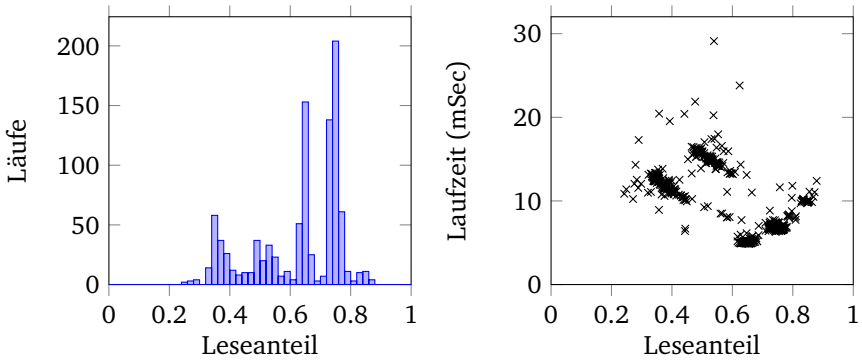



Abbildung 7.3.: `recode`-Messungen, ohne XServer. Links Histogramm der Verteilung (1000 Läufe), rechts Laufzeit über Verteilung.

Messwerkzeugs hat, welche sich in einer sichtbaren Diskretisierung der Ausführungszeiten niederschlägt. Um dem Problem zu begegnen, wurde zunächst ein neues Zeitmesswerkzeug `etime` implementiert, welches die Zeit auch im Mikrosekundenbereich darstellt und verwendet wird, falls die Messergebnisse eine entsprechende zeitliche Auflösung erfordern. Die Details finden sich in Anhang B.[1]

Da die Messung noch zu viele Messfehler enthält, wurde diese zunächst auf die gebundenen Daten von `aget` beschränkt. Die erste untersuchte vermeintliche Fehlerquelle sind andere Programme. Um dieser Quelle zu begegnen, wurde die graphische Oberfläche des Systems beendet und die Messung in einem Terminal durchgeführt. Der Effekt ist in Abbildung 7.3 dargestellt. Gegenüber der naiven Messung sind die Daten deutlich weniger verschmiert. Obwohl die Daten erkennbare Cluster bilden, die zudem noch relativ viele Datenpunkte enthalten, kann an dieser Stelle nicht aufgehört werden.

Der einzige verbleibende Prozess auf dem laufenden System ist die Messumgebung. Daher kann der Messcode so umgestaltet werden, dass die Messungen über `ssh` von einem anderen Rechner ausgeführt werden. Das Ergeb-

[1] Das Problem ist, dass time zu stark rundet. Dadurch kann ein Aufruf auch *gemessene* 0.0 Sekunden dauern, was uns vor gewisse Schwierigkeiten stellt.

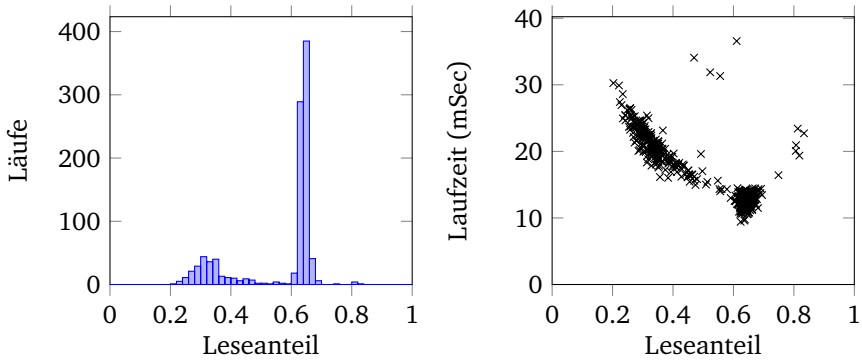



Abbildung 7.4.: `recode`-Messungen, ohne XServer und über SSH. Links Histogramm der Verteilung, rechts Laufzeit über Verteilung.

nis ist in Abbildung 7.4 dargestellt. Insbesondere wenn man das Histogramm betrachtet, könnte man sich leicht von der phänomenalen Messschärfe beeindrucken lassen. Das Problem besteht hier allerdings darin, dass die vorige Messung gezeigt hat, dass die optimale Ausführungszeit bei etwa 5ms liegt. In diesem Messaufbau liegt die beste Ausführungszeit bei etwa 10ms. Folglich muss davon ausgegangen werden, dass jede Messung falsch ist. An dieser Stelle wurden verschiedene Experimente mit verschiedenen Konfigurationon der `ssh`-Session und auch einer Variante von `etime`, welche die CPUs aufweckt, getestet. Letzteres basierte auf der Theorie, dass sich CPUs zu tief schlafen legen, wenn zwischen den Messungen Zeit vergeht, da neue `ssh`-Sessions erzeugt werden. In dieser Richtung konnten jedoch keine verwertbaren Erkenntnisse gewonnen werden, sodass dieser Weg nicht weiter verfolgt wurde.

Aus diesem Grund wurde die Messung wieder auf die ausführende Maschine verschoben. Da die Frage nach den schlafenden CPUs jedoch noch im Raum stand, wurde diese ebenfalls untersucht. Die Präzision wurde jedoch durch das Abschalten des entsprechenden Effekts schlechter, was die Frage nach thermischen Problemen der CPU aufwirft. Folglich wurden die manuelle Lüfterkontrolle auf automatisch gestellt. Das Ergebnis der Messungen

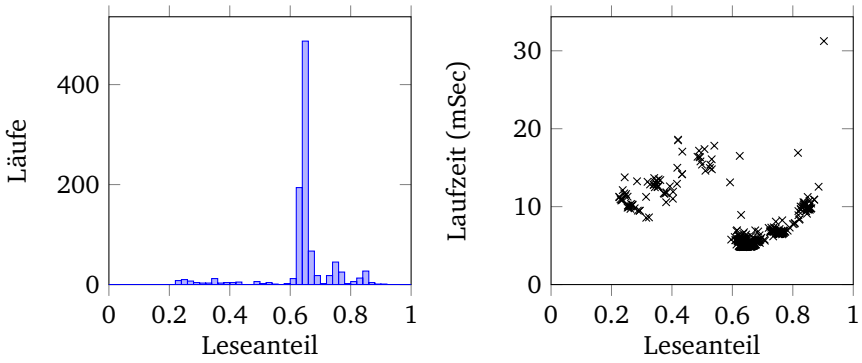



Abbildung 7.5.: `recode`-Messungen, ohne XServer und automatische Lüfterkontrolle. Links Histogramm der Verteilung, rechts Laufzeit über Verteilung.

mit automatischer Lüfterkontrolle findet sich in Abbildung 7.5. Wie man im Histogramm deutlich erkennen kann, sind fast die Hälfte der Daten in einem 2% breiten Bereich angesiedelt. Zudem befinden sie sich, was im Scatterplot gut zu sehen ist, in der Nähe der minimalen Ausführungszeit.

Die Frage, die sich an dieser Stelle natürlich stellt, ist, ob die restlichen Ausreißer auch verschwinden, wenn der Lüfter aggressiver eingestellt wird. Dies ist jedoch nicht der Fall, was die in dieser Arbeit unbeantwortete Frage nach dem tatsächlichen Grund aufwirft. Ferner wurde ein größerer Lüfter installiert, um das entstandene Geräuschproblem zu beseitigen. Hierdurch scheint sich der Messfehler geringfügig reduziert zu haben. Davon abgesehen konnten keine weiteren Erfolge in der Reduktion der Messfehler erzielt werden. Nichtsdestotrotz ergibt sich hierdurch ein grundlegender Testaufbau, der keine allzu großen Messfehler verursacht. Dieser wird verwendet, um die `recode`-Messung noch einmal mit der aktuellen Implementierung auf alle `aget` Derivate anzuwenden. In Abbildung 7.6 kann man sehen, dass diese Derivate auf die Messung keine nennenswerte Auswirkung haben.

Die folgenden Messungen sind allesamt so aufgebaut, dass alle Messungen auf einem Datenpunkt durchgeführt werden und dann der nächste Datenpunkt vermessen wird. Dadurch wird durch die zeitliche Nähe der

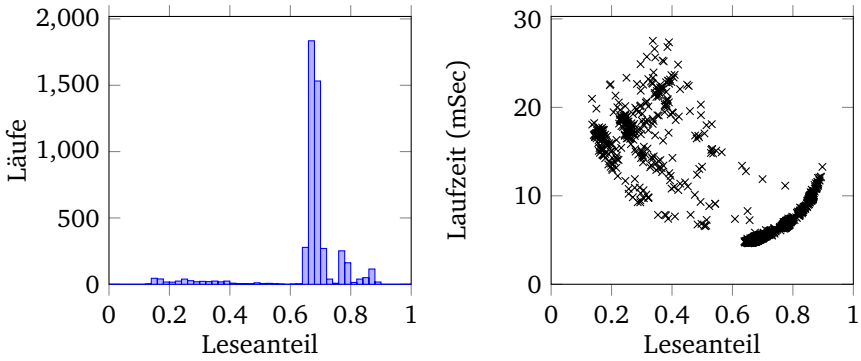



Abbildung 7.6.: `recode`-Messungen für `aget` Derivate (5000 Läufe) – finale Messmethode mit aktueller Implementierung.

Messung verglichener Implementierungen pro Datenpunkt eine gewisse Vergleichbarkeit gewährleistet, selbst wenn Umgebungsvariablen wie etwa die Raumtemperatur einen Einfluss auf das Ergebnis hätten.

### 7.1.9. Systematische Risiken

In diesem Abschnitt wird kurz auf systematische Risiken der hier durchgeführten Evaluation hingewiesen und diese werden kurz bewertet. Ein offensichtliches Risiko ist, dass die Messungen an sich fehlerhaft sein könnten. Um diesem Risiko zu begegnen, enthält jeder Messaufbau eine Maßnahme, um die Resultate zu validieren. Daher ist diese Gefahr eher gering.

Ein gewisses Risiko ergibt sich aus der Zahl der Messwiederholungen. Hohe Wiederholungsraten führen zu gut abschätzbaren tatsächlichen Werten des Gemessenen. Andererseits sind die Ursachen der Messfehler eine gewisse Unberechenbarkeit der genutzten Testumgebung, welche man als Werkzeugnutzer im realen Betrieb genauso erfahren würde. Würde man Messungen ohne Wiederholung durchführen, oder das Messsystem nach jeder Messung neu starten, so ergäben sich eventuell Ergebnisse, die einer Nutzung durch einen Anwender besser abbilden. Die systematische Auswertung dieser Daten wäre aber schwieriger, da es weniger Datenpunkte gäbe, was zu

einer größeren Unsicherheit führt. Auch verlängert ein Systemneustart die Gesamtdauer einer Messung erheblich, was negative Konsequenzen auf die Reproduzierbarkeit der Ergebnisse hat. Simuliert man einen Systemneustart, indem man alle Caches leert, erhält man zwar eine gute Worstcaseabschätzung, das Verhalten ist aber wenig praxisnah. Daher verwenden die hier dargestellten Experimente, falls sie nicht ohnehin deterministisch[1] sind, so viele Wiederholungen, dass sie eine Gesamtausführungszeit von zwei Tagen nicht wesentlich überschreiten.

Ein weiteres Risiko besteht in der Wahl der tatsächlich verwendeten Implementierungen von SKilL. Zum einen gibt es große Freiheitsgrade bei der Realisierung der IR-Spezifikation, zum anderen sind die Implementierungen unterschiedlich vollständig. Alle Implementierungen bilden dabei den Kern von SKilL vollständig[2] ab und sind in diesem Bereich durch automatische Tests getestet. Alle vermessenen Implementierungen eigenen sich prinzipiell als Grundlage weiterer Bauhauswerkzeuge, sodass keine Gefahr für die Validität der Resultate besteht. Eine gewisse Unsicherheit besteht, will man von der aktuellen Performanz auf die Performanz einer Implementierung schließen, die SKilL in jedem Detail vollständig implementiert. Eine erhebliche Unsicherheit besteht, wenn man versuchen würde, aus den Performanzunterschieden einzelner Implementierungen Rückschlüsse auf die Performanz der genutzten Programmiersprache zu ziehen. Dies liegt vor allem daran, dass die Implementierungen nicht bis ins letzte Detail optimiert sind, was für die Betrachtungen in dieser Arbeit jedoch keine Rolle spielt.

Zu guter Letzt könnte die Wahl der Testdaten nicht ausreichend repräsentativ sein. Dabei ist die Wahl der verwendeten Programme selbst kaum ein Problem. Es wurde eine Vielzahl echter Programme unterschiedlicher Größe verwendet, sodass das Verhalten über einen weiten Bereich gut einschätzbar sein sollte. Die Beschränkung auf die Bauhaus-Spezifikation ist hier vielmehr eine Risikoquelle, die aber nicht allzu groß sein dürfte. Eine Beschränkung

[1] Dateigrößen und Graphstruktur können z.B. präzise gemessen werden, ebenso z.B. die Kompression durch ein deterministisches Kompressionsverfahren.

[2] Von der bereits erwähnten Unzulänglichkeit der C++-Implementierung in Verbindung mit verteilten Feldern in Kombination mit Subtyping und ungünstigen Blöcken, welche nachträglich beseitigt wurde, einmal abgesehen. Siehe Anhang C.

der Analysen sowie der analysierten Programmiersprache stellt vermutlich kein eigenes Risiko dar, da sich Variationen dieser Eigenschaft in einer anderen IR-Spezifikation und Graphgröße niederschlagen würden. Es ist nicht erkennbar, warum sich die Ergebnisse dadurch prinzipiell ändern sollten.

## 7.2. Frühere Evaluationen

Zu Beginn des Projekts wurden einige, auf synthetischen Graphen basierende, Evaluationen durchgeführt [Fel14; Prz14; Rot15]. Diese basieren alle auf Architekturen, die nicht weiter verfolgt wurden und sind alleine deswegen für die Betrachtung des aktuellen Zustands von SKilL nur bedingt relevant. Eine interessante Erkenntnis, die sich vor allem aus den Vergleichen in [Prz14; Rot15] gewinnen lässt, ist, dass die Wahl der Datenstrukturen um Objekte im Speicher zu repräsentieren die Laufzeit so stark dominiert, dass sie zunächst einen bedeutend größeren Einfluss auf das Gesamtverhalten hat als die Wahl der Programmiersprache oder sonstige Entscheidungen, die dem Binärformat zugrunde liegen.

## 7.3. Lesen und Schreiben

Bei diesem Experiment wird ein Graph in den Hauptspeicher gelesen und anschließend in eine neue `.sf`-Datei geschrieben.[1] Dabei werden im Wesentlichen die folgenden Hypothesen verfolgt:

- Die Lesegeschwindigkeit entspricht grob der Schreibgeschwindigkeit.
- Erzielte Lese- und Schreibraten sind für alle SKilL-Anbindungen akzeptabel.
- Ergebnisse sind auf andere Testsysteme übertragbar.

**Testaufbau** Um diese Hypothesen zu untersuchen, wird in jeder untersuchten Anbindung ein Werkzeug implementiert, das `recode` genannt wird. Das Werkzeug verwendet zwei Parameter als Eingabe. Der erste ist der Eingabepfad, der zweite der Ausgabepfad. Dabei wird eine vollständige SKilL Spezifikation ohne *ondemand* oder *distributed* Hints verwendet, damit das Öffnen einer `.sf`-Datei zu einem vollständigen Lesen führt. Der resultierende Zustand bekommt den Ausgabepfad und schreibt mit einer Flush-Operation auf den gewünschten Pfad. Das Werkzeug misst die Dauer der beiden Operationen und gibt deren Verhältnis auf der Standardausgabe aus. Die Gesamtausführungszeit des Werkzeugs wird von außen gemessen.

Die Messung wird für alle Testdaten durchgeführt. Das Werkzeug wird zunächst einmal ungemessen ausgeführt und danach zehnmal hintereinander vermessen. Es ist im Folgenden zu bedenken, dass sich hierdurch aus den insgesamt 282 Testgraphen 2820 Datenpunkte pro Anbindung und Maschine ergeben. Außerdem sind die Eingabedaten nicht gleichmäßig verteilt, sodass leicht unterschiedliche Eingabedaten den optischen Eindruck eines Messfehlers hervorrufen können.

[1] Keine der Implementierungen kopiert dabei Daten direkt von einer `.sf`-Datei in die nächste, was eine denkbare, aber nicht verfolgte Optimierung ist. Diese würde jedoch die folgende Messung von ihrem Sinn befreien.

**Validierung** Die Validierung der Ergebnisse erfolgt mittels strukturellem Vergleich von Eingabe und Ausgabe (siehe Anhang A). Ein Vergleich des Bytestroms ist leider unmöglich, da die Kodierung Freiheitsgrade enthält, die von Implementierungen ausgenutzt werden, um eine bessere Performanz zu erreichen. Da der Vergleich selbst einen erheblichen Ressourcenbedarf verursacht, wird er zum einen nicht während der eigentlichen Messung durchgeführt[1] und zum anderen nur auf *Workstation*. Die Ausführungszeit der Validierung beträgt mit 10 Wiederholungen etwa zwei Tage, die Wiederholungen wurden aber dennoch durchgeführt, um ein sporadisches Versagen erkennen zu können. Es wurde kein fehlerhaftes Verhalten festgestellt.[2]

Um sich vor sporadischem Versagen durch zu hohen Speicherverbrauch zu schützen, ist die Ausgabe des Verhältnisses zwischen Lesen und Schreiben die letzte Aktion des Programms. Ist diese vorhanden, so ist aufgrund der imperativen Natur der Implementierungen davon auszugehen, dass die Messung insgesamt erfolgreich verlaufen ist.

### 7.3.1. Ausführungszeit: Lesen ≈ Schreiben?

Um diese Hypothese zu untersuchen, wird das Verhältnis zwischen Lesen zur Gesamtlaufzeit und Schreiben zur Gesamtlaufzeit über alle Anbindungen und Eingaben betrachtet. Da es einen Unterschied in der Skalierbarkeit von Lese- und Schreiboperation geben könnte, wird zudem untersucht, wie sich dieses Verhältnis entwickelt. Die Ergebnisse sind zunächst als Histogramm (Abb. 7.7) sowie als doppelt logarithmisches Diagramm (Abb. 7.8) dargestellt.

**Interpretation** Zunächst ist in Abbildung 7.8 erkennbar, dass es kein wirklich scharfes Ergebnis gibt. Zwar sinkt die Streuung tendenziell deutlich mit der Dateigröße, sie nimmt aber punktuell wieder zu. Weiterhin ist zu erkennen, dass bei großen `.sf`-Dateien das Phänomen entsteht, dass Scala

[1] Das hätte vermutlich verfälschende Konsequenzen, die mit der Messung über `ssh` vergleichbar wären.

[2] Diese Prüfung fand mit dem hier beschriebenen Testaufbau vom 3.4. bis zum 6.4.17 statt.

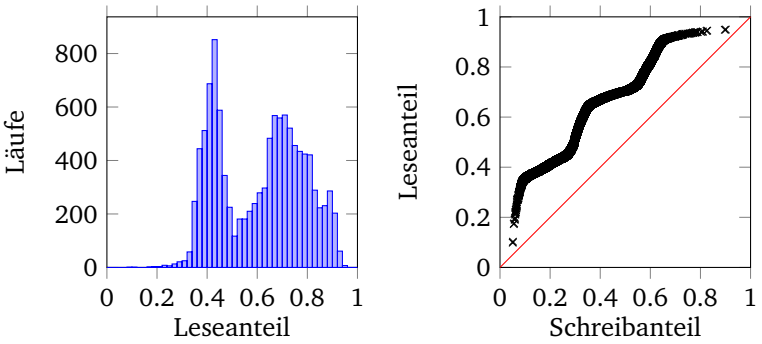



Abbildung 7.7.: Histogramm in 2%-Schritten über die Leseanteile(links). Sortierte Lese- über Schreibanteile (rechts).

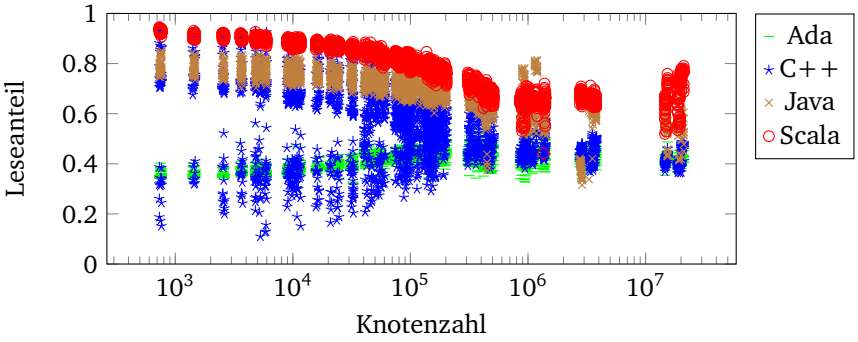



Abbildung 7.8.: Verteilung der bei Neukodierung ermittelten Leseanteile abhängig zur Graphgröße.

länger liest, wogegen Java länger schreibt. Hier gibt es offensichtlich noch Optimierungspotential in mindestens einer der beiden Implementierungen. Die scheinbar fallenden oder steigenden Trends, die man im linken Teil der Abbildung sehen kann, entstehen durch leicht unterschiedliche konstante Kosten beim Lesen und Schreiben.

In Abbildung 7.7 ist klar erkennbar, dass die Daten nicht normalverteilt sind und es insgesamt einen Hang zum längeren Lesen gibt. Um das

| | $\mathcal{Q}_{0.025}$ | $\mathcal{Q}_{0.5}$ | Mittelwert | $\mathcal{Q}_{0.975}$ |
|---|---|---|---|---|
| Untergrenze | 0.3736 | 0.6144 | 0.5945 | 0.8455 |
| Obergrenze | 0.3804 | 0.6282 | 0.6009 | 0.8509 |

Tabelle 7.12.: 95%-Intervalle für Charakteristika der Verteilung der Leserate.

wahrscheinliche Verhalten abschätzen zu können, werden mittels *bootstrapping*[Efr79][1] 0.025-, 0.5- und 0.975-Perzentile sowie der Mittelwert berechnet. Um eine halbwegs zuverlässige Antwort für große Graphen zu erhalten, wurde die Berechnung auf Resultate von Graphen mit über 22000 Knoten beschränkt. Bei der Verarbeitung von Graphen unterhalb dieser Grenze ist eine hohe IO-Performance ohnehin kein entscheidendes Argument, da die Ausführungszeiten hier im niedrigen Millisekundenbereich und damit vernachlässigbar[2] sind. Die Ergebnisse sind in Tabelle 7.12 dargestellt.

Die statistische Analyse zeigt, dass 95% der Messungen bei einer Konfidenz von 95% zwischen 37% und 85% liegen. Dies entspricht an der Untergrenze einer 1,7-fachen Schreibdauer und einer 5,6-fachen Lesedauer an der Obergrenze. Es ist dabei zu erwarten, dass einen Faktor 1,5 länger gelesen als geschrieben wird. Beschränkt man sich bei unseren Daten und Implementierungen auf die Untersuchung von nur einem von beiden, so sollte man sich tendenziell auf das Lesen beschränken. Eine tendenziell längere Lesedauer ist aufgrund der notwendigen Objektallokation prinzipiell zu erwarten.

### 7.3.2. Ausführungszeit – Anbindungsabhängigkeit

Dieser Abschnitt beschäftigt sich mit den absoluten Ausführungszeiten für Lesen und Schreiben gesondert nach Zielsprache einzelner SKilL-Anbindungen. Dafür werden aus der Gesamtzeit und dem Verhältnis zwischen Lesen und Schreiben die jeweiligen absoluten Zeiten errechnet. Das Ergebnis ist in Abb. 7.9 und 7.10 dargestellt. Um die Skalierbarkeit zu quantifizieren, wurden Regressionsanalysen zu den abgebildeten Daten durchgeführt, welche in Tabelle 7.13 dargestellt sind. Da es nur einen indirekten Zusammenhang zwi-

[1] Tatsächlich kam das GNU R Paket `boot` zum Einsatz.
[2] Man bedenke, dass man in unserem Modell im Wesentlichen einen Graphen bearbeitet.

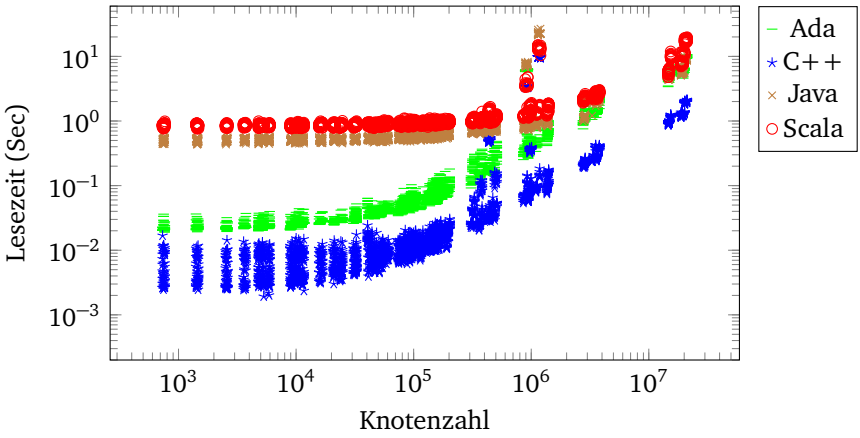



Abbildung 7.9.: Absolute Ausführungszeit für Lesen nach Anbindung.

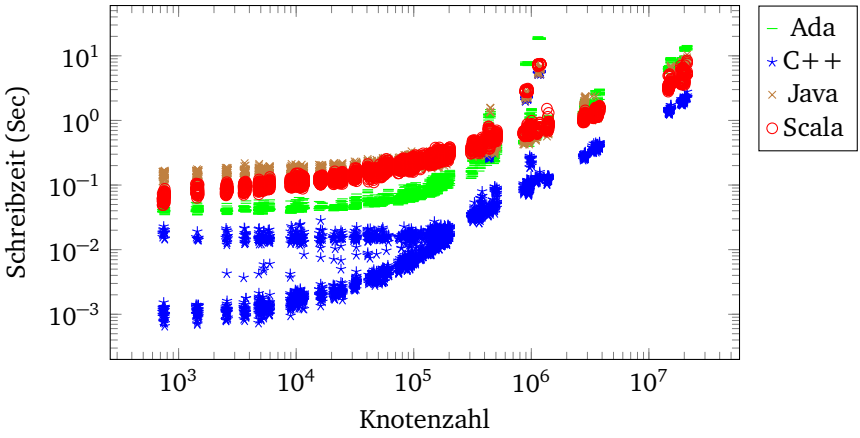



Abbildung 7.10.: Absolute Ausführungszeit für Schreiben nach Anbindung.

| | Ada read | C++ read | Java read | Scala read |
|---|---|---|---|---|
| $10^6$ Knoten | 0.356*** | 0.072*** | 0.321*** | 0.529*** |
| | (0.007) | (0.004) | (0.009) | (0.006) |
| Constant | 0.201*** | 0.106*** | 0.740*** | 0.943*** |
| | (0.028) | (0.017) | (0.038) | (0.026) |
| Observations | 2820 | 2820 | 2820 | 2820 |
| $R^2$ | 0.511 | 0.104 | 0.307 | 0.726 |
| F Statistic (df = 1; 2818) | 2940.052*** | 327.214*** | 1245.739*** | 7477.172*** |

| | Ada write | C++ write | Java write | Scala write |
|---|---|---|---|---|
| $10^6$ Knoten | 0.498*** | 0.100*** | 0.273*** | 0.241*** |
| | (0.008) | (0.002) | (0.003) | (0.003) |
| Constant | 0.239*** | 0.057*** | 0.278*** | 0.285*** |
| | (0.033) | (0.009) | (0.012) | (0.013) |
| Observations | 2820 | 2820 | 2820 | 2820 |
| $R^2$ | 0.593 | 0.419 | 0.767 | 0.689 |
| F Statistic (df = 1; 2818) | 4111.929*** | 2030.210*** | 9276.481*** | 6252.118*** |
| *Note:* | | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.13.: Regression über Ausführungszeiten in Abhängigkeit der Knotenzahl. Konstanter Anteil in Sekunden und Steigung in Sekunden pro $10^6$ Knoten.

schen Knotenzahl und Dateigröße gibt, die Ausführungszeit aber maßgeblich von der Dateigröße abhängt, findet sich in Tabelle 7.14 die Regression der Laufzeit in Abhängigkeit der Dateigröße.

**Interpretation** Zunächst fällt auf, dass bis auf die konstanten Kosten beim Lesen in C++ in Abhängigkeit der Dateigröße alle Regressionen signifikant sind. Ferner sind die $R^2$-Werte beim Lesen deutlich geringer als beim Schreiben. Hierfür sind hauptsächlich nicht-lineare Konstruktoren von Containern verantwortlich. Diese Eigenschaft kann bei sehr großen Containern die Laufzeit sichtbar beeinflussen und ist der tatsächlich verwendeten Containerimplementierung geschuldet. Sollte sich hieraus in der Praxis ein Problem ergeben, wären in Implementierungen die genutzten Containerklassen durch

| | Ada read | C++ read | Java read | Scala read |
|---|---|---|---|---|
| $10^9$ Byte | 19.786*** | 6.123*** | 20.291*** | 26.072*** |
| | (0.163) | (0.142) | (0.279) | (0.125) |
| Constant | 0.062*** | 0.014 | 0.540*** | 0.837*** |
| | (0.016) | (0.014) | (0.027) | (0.012) |
| Observations | 2820 | 2820 | 2820 | 2820 |
| $R^2$ | 0.839 | 0.398 | 0.652 | 0.939 |
| F Statistic (df = 1; 2818) | 14676.500*** | 1861.032*** | 5285.807*** | 43713.730*** |

| | Ada write | C++ write | Java write | Scala write |
|---|---|---|---|---|
| $10^9$ Byte | 26.523*** | 5.824*** | 13.098*** | 12.043*** |
| | (0.172) | (0.063) | (0.061) | (0.068) |
| Constant | 0.080*** | 0.010* | 0.233*** | 0.232*** |
| | (0.017) | (0.006) | (0.006) | (0.007) |
| Observations | 2820 | 2820 | 2820 | 2820 |
| $R^2$ | 0.894 | 0.751 | 0.942 | 0.918 |
| F Statistic (df = 1; 2818) | 23679.200*** | 8496.841*** | 45755.540*** | 31431.120*** |
| *Note:* | | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.14.: Regression über Ausführungszeiten in Abhängigkeit der Dateigröße. Konstanter Anteil in Sekunden und Steigung in Sekunden pro $10^9$ Byte.

effizientere Pendants zu ersetzen. Diese Eigenschaft hat mit SKilL an sich wenig zu tun, da die konkrete Implementierung nicht festgelegt wird.

Des Weiteren fällt auf, dass Java beim Lesen deutlich schneller als Scala ist, dieses jedoch beim Schreiben signifikant bessere Ergebnisse liefert. Da es sich bei beiden Programmiersprachen um JVM-Sprachen handelt, die sich sogar in Teilen eine Implementierung teilen, ist zu erwarten, dass es möglich wäre, auf die jeweils bessere Implementierung zurückzugreifen. Ebenso ist überraschend, dass sich Ada und C++ sehr stark unterscheiden.

Die bedeutend höheren Fixkosten der JVM-Sprachen gegenüber den na-

tiven Programmiersprachen sind erfahrungsgemäß[1] dem JIT-Compiler geschuldet. Insgesamt überrascht, dass sich der Einsatz von JVM-Sprachen in puncto Serialisierung der IR bei großen Graphen zeitlich kaum auswirkt.

Bemerkenswert ist, dass in C++ sowohl die Zeit zum Lesen, als auch die Zeit zum Schreiben selbst bei den größten `.sf`-Dateien nur im einstelligen Sekundenbereich liegt. Diese Beobachtung wird später bei einem Vergleich mit anderen Verfahren noch weiter verfolgt.

### 7.3.3. Ausführungszeit – Anbindungs- und Maschinenabhängigkeit

Da bereits eine erhebliche Anbindungsabhängigkeit festgestellt wurde, wird die Maschinenabhängigkeit für jede Anbindung gesondert betrachtet. Dies geschieht analog zur Betrachtung der Anbindungsabhängigkeit und ist in Abb. 7.11 nach Anbindung und Operation aufgeschlüsselt dargestellt.

**Interpretation** Die Unterschiede zwischen den Maschinen fallen erkennbar geringer aus als die Unterschiede zwischen den einzelnen Implementierungen. Aufgrund der geringen Zahl gemessener Maschinen ist es schwer, zuverlässige Erkenntnisse über das Verhalten der Implementierungen zu gewinnen. Die Performance der *Workstation* ist in der Regel besser als die des *Laptops*, welcher wiederum zumeist erheblich besser ist als der *Server*. Hieraus können wir unmittelbar schließen, dass weder die Anzahl der Prozessoren, noch die Blocktransferraten des Hintergrundspeichers die Ausführungszeiten alleine dominierend beeinflussen. Allerdings zeigt *Server2* bei näherer Betrachtung der Java-Schreibzeit eine bemerkenswert hohe Skalierbarkeit (siehe Regression in Tab. 7.16). Betrachtet man Scala, so fällt auf, dass die *Laptop* Kurven für kleine Daten näher an *Workstation* sind und für große Daten näher an *Server*. Hier zeigt *Server2* insbesondere beim Schreiben ein entgegengesetztes Verhalten, was vermuten lässt, dass die Schreiboperationen in dieser Implementierung tatsächlich relativ gut parallelisieren.

[1] Diese Einschätzung basiert auf Beobachtung beim Debuggen und Profilen während der Entwicklung. Wie man hierzu ein zuverlässiges Experiment gestaltet ist unmittelbar nicht ersichtlich.

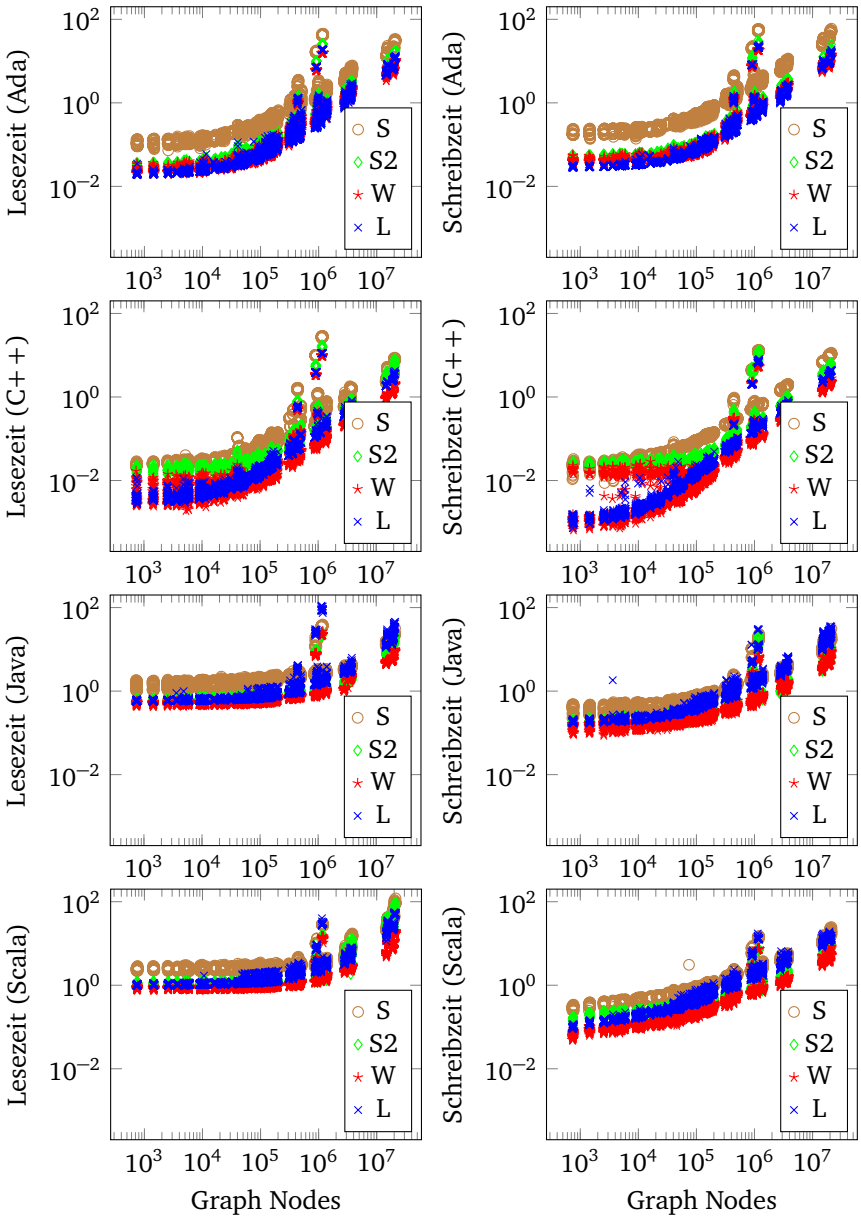



Abbildung 7.11.: `recode`-Maschinenabhängigkeit. Alle Zeiten in Sekunden. Legende: **S**erver, **S**erver**2**, **W**orkstation, **L**aptop

| | Workstation | Laptop | Server | Server2 |
|---|---|---|---|---|
| $10^6$ Knoten | 0.356*** | 0.427*** | 1.141*** | 0.606*** |
| | (0.007) | (0.008) | (0.018) | (0.011) |
| Constant | 0.201*** | 0.211*** | 0.669*** | 0.284*** |
| | (0.028) | (0.033) | (0.075) | (0.045) |
| Observations | 2820 | 2820 | 2820 | 2820 |
| $R^2$ | 0.511 | 0.512 | 0.598 | 0.536 |
| F Statistic (df = 1; 2818) | 2940.052*** | 2952.800*** | 4184.850*** | 3249.117*** |
| *Note:* | | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.15.: Regression zu Ada-Lesezeit.

| | Workstation | Laptop | Server | Server2 |
|---|---|---|---|---|
| $10^6$ Knoten | 0.273*** | 1.057*** | 0.773*** | 0.329*** |
| | (0.003) | (0.009) | (0.008) | (0.006) |
| Constant | 0.278*** | 0.444*** | 0.801*** | 0.349*** |
| | (0.012) | (0.038) | (0.033) | (0.027) |
| Observations | 2820 | 2820 | 2820 | 2820 |
| $R^2$ | 0.767 | 0.830 | 0.779 | 0.477 |
| F Statistic (df = 1; 2818) | 9276.481*** | 13754.450*** | 9921.400*** | 2565.194*** |
| *Note:* | | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.16.: Regression zu Java-Schreibzeit.

Im Kontext der Parallelität ist es vermutlich ratsam, das Spektrum in große und kleine `.sf`-Dateien aufzuteilen und diese gesondert zu untersuchen. Beschäftigt man sich mit Java, so fällt auf, dass hier *Laptop* manchmal von *Server* geschlagen wird.

Die nahezu identischen Ergebnisse von *Laptop* und *Workstation* für Ada sind ausgesprochen überraschend, da sich hier kaum Erklärungsmöglichkeiten bieten. Den optischen Eindruck untermauert eine Regressionsanalyse, welche in Tabelle 7.15 dargestellt ist. Die einzige Hardwareeigenschaft,

die hier zugrunde gelegt werden kann ist die CAS Latenz mit 13.5ns (Laptop) bzw. 13.35ns (Workstation). Dies würde jedoch bedeuten, dass die Ausführungszeit durch vom Cache nicht abfangbare Speicherzugriffe beschränkt wird, was insofern wenig plausibel ist, als die vergleichbare C++ Implementierung dieses Verhalten nicht aufzeigt, die Caches aufgrund der unterschiedlichen Kernzahlen unterschiedlich oft vorhanden sind und der L3-Cache mit 3 respektive 15 MB unterschiedlich groß ausfällt. Hinzu kommt, dass diese Größe auch bei *Server* ähnlich ist. Es ist daher nicht möglich, aus den gegebenen Daten Schlüsse über wahrscheinliches Verhalten auf anderen Hardwarekonfigurationen zu ziehen. Da sich die Daten in der Tendenz kaum unterscheiden, werden weitere Messungen auf *Workstation* beschränkt.

### 7.3.4. Plattformunabhängigkeit

Um die Platformunabhängigkeit zu demonstrieren, wird der Testaufbau leicht modifiziert. Es wird auf *Laptop* ein Windows gestartet und die in Java implementierte Variante von `recode` sowie alle einfachen `.sf`-Dateien von *Workstation* kopiert. Um Ressourcen zu sparen und den Testaufbau einfach zu halten,[1] beschränkt sich das Experiment auf `.sf`-Dateien die kleiner als 100 MB sind. Unter Windows werden die `.sf`-Dateien mittels `recode` gelesen und geschrieben. Die Ergebnisse werden wieder auf *Workstation* kopiert und dort mittels `imlDiff`, d.h. wie bei der Validierung der übrigen auf `recode` basierten Messungen, gegen die Originale verglichen. Alle Ausgabeverzeichnisse werden vor der Ausführung geleert.

Dieses Experiment lässt sich beliebig oft wiederholen, ohne dass es einen Unterschied zwischen den Graphen gäbe. Mit der angegebenen Version wurde das Experiment am 6.4.2017 erfolgreich durchgeführt. Daher ist es offensichtlich kein Problem ein Werkzeug auch auf einem anderen Betriebssystem auszuführen. Dieses Experiment ließe sich natürlich auch für die übrigen Implementierungen durchführen. Es wäre dann aber notwendig,

[1] Heaps über 2GB sind unter Windows nach wie vor ein Problem, dem hier nicht begegnet wurde, da es für die Aussage insgesamt keine Rolle spielt.

die entsprechenden Binaries neu zu übersetzen.[1]

### 7.3.5. Zusammenfassung

In diesem Kapitel wurde gezeigt, dass Lesen tendenziell etwas teurer ist als Schreiben. Es wurde erkannt, dass sich die Laufzeiten von SKilL-Implementierungen je nach Implementierungssprache stark unterscheiden. Außerdem stellte sich heraus, dass sich die Laufzeiten auf den verfügbaren Testsystemen nicht wesentlich unterscheiden. Ferner konnte demonstriert werden, dass alle Implementierungen an sich vertretbare Laufzeiten aufweisen. Dabei zeigte sich, dass die Validierung von Messungen von Schreiboperationen durch Freiheitsgrade zu einer teuren Operation wird (vergleiche Anhang A). Daher werden weitere Messungen, soweit möglich, auf lesende Operationen und leicht validierbare Ergebnisse zurückgreifen.

[1] Die derzeitige Implementierung des Tokenstroms in C++ und Ada verwendet die `mmap`-Header [POS16], d.h. es ist eine Übersetzungsumgebung zu wählen, welche diese Header ebenso bereitstellt, oder diese sind zu ersetzen. Eine Ersetzung involviert die Anpassung von etwas unter 100 Codezeilen, da die Zugriffe strikt gekapselt sind.

## 7.4. Verzögerte Serialisierung in PTA-Prototypen

Dieser Abschnitt untersucht die Anwendung bedarfsorientierter Serialisierung auf die von Matthias Harrer[Har16] entwickelten prototypischen Implementierungen von Zeigeranalysewerkzeugen. Hierbei handelt es sich zunächst um fünf Werkzeuge: Jeweils eines für eine Zeigeranalyse nach Steensgaard, Das und Andersen. Zudem ein Werkzeug, welches aus IML die verwendete abstrakte Speicherstruktur extrahiert, sowie ein weiteres Werkzeug, das zwei berechnete Speicherstrukturen vergleichen kann. In diesem Abschnitt werden nur das Werkzeug zur Erzeugung der Speicherstruktur (`heap-structure`) und die Implementierung der Zeigeranalyse nach Steensgaard (`steensgaardV2`) verwendet. Die Originalwerkzeuge wurden an die in dieser Arbeit verwendete Spezifikation angepasst (siehe §7.5.3). Für dieses Experiment wurden einige, für studentische Arbeiten übliche, kleinere Fehler beseitigt. Beispiele hierfür sind der Aufbau nicht benötigter Datenstrukturen oder die Wahl falscher Datenstrukturen, wie etwa HashMaps mit dichten Integer-Keys anstelle von Arrays. Zudem wurde dafür gesorgt, dass die resultierenden Daten deterministisch sind, um diese über einen strukturellen Vergleich validieren zu können.

Eine Quantifizierung der Auswirkungen bedarfsorienteren Lesens im Allgemeinen ist nicht das Ziel dieser Arbeit. Eine solche findet sich für XML in [NSL02]. Da die konkreten Effekte stark von der Qualität der Implementierung dieses Aspekts abhängen und in diesem Bereich nur das Allernötigste investiert wurde, wären gewonnene Daten voraussichtlich schnell überholt. Es ist zu erwarten, dass mit der von Jonathan Roth entwickelten Objektrepräsentation[Rot15] deutliche Verbesserungen zu erreichen sind, wenn man diese lediglich für verteilte Daten verwendet. Es wird offen gelassen, ob sich die für SKilL getroffene Entscheidung, das *!onDemand !distributed* impliziert, in der Praxis bewährt. Ferner lässt sich SKilL so implementieren,[1] dass der Wechsel zwischen *!onDemand* und direktem Auslesen das öffentliche API nicht verändert. Es lässt sich also vergleichsweise günstig durch Neugenerie-

[1] Vorausgesetzt, das Objekt über eine Fabrikmethode des besitzenden Zustands oder Pools erzeugt werden, da verteilte Felder eine Referenz zum Pool erfordern.

rung der SKilL-Anbindung zwischen den beiden Modi wechseln, sodass in der Praxis einfach geprüft werden kann, ob es günstiger ist *!onDemand* zu verwenden.

Im Rahmen dieser Arbeit werden die Kosten und Nutzen der Umstellung existierender Werkzeuge auf bedarfsorientiertes Lesen und Anhängen untersucht. Dabei geht es im einzelnen um die folgenden Hypothesen:

- Den vom Anwendungscode nicht genutzten Teilgraph beim Schreiben nach Bedarf zu lesen ist in etwa so schnell, wie ihn direkt zu lesen.
- Der ungenutzte Teilgraph wird beim bedarfsorientierten Lesen nicht gelesen, wenn er beim Anhängen nicht benötigt wird.
- Partielle Spezifikationen verringern die Ausführungszeit und die Zahl generierter Code-Zeilen rein lesender Analysewerkzeuge.

### Messung der Ausführungszeit

Im Folgenden sollen die Auswirkung verschiedener IR-Spezifikationen auf die Laufzeit einer echten Werkzeugkette vermessen werden, um das Nutzen von bedarfsorientiertem Lesen und Anhängen zu evaluieren. Da die Werkzeuge ihre Eingabedatei mit Informationen anreichern, wird vor jedem Testlauf eine Kopie der Eingabe erstellt, auf der anschließend gearbeitet wird. Diese Kopie wird zunächst an das Werkzeug `heap-structure`, welches den abstrakten Heap erzeugt, und danach an das Werkzeug `steensgaardV2` übergeben, welches die eigentliche Zeigeranalyse durchführt. Das Resultat wird zur Validierung gegen einen Graph, der durch vollständiges Lesen und Schreiben erzeugt wurde, strukturell abgeglichen.

Da es sich hierbei um Prototypen handelt, ergeben sich bei deren Nutzung unangenehme Einschränkungen. Die Darstellung der Heapstruktur ist weniger kompakt, als es bei der äquivalenten Bauhausstruktur der Fall ist, damit sie unverändert auch für eine Analyse nach Das oder Andersen eingesetzt werden kann, was in Bauhaus nicht möglich ist. Infolgedessen kommt es zu zwei Problemen, die eine Messung für manche Testdaten verhindern. Die fünf *php*-Austauschdateien lassen sich zwar analysieren, aber nicht schreiben,

da das Resultat größer als 2 GB wäre und es dadurch zu einem Integerüberlauf beim Zugriff auf den Bytestrom[1] kommt. Ein weiteres Problem ergibt sich für die Analyse von *gnugo*. Hier reichen selbst 24GB Heap nicht, um die Analyse durchzuführen, was für eine Messung auf *Workstation* problematisch ist.[2] Aus Gründen der Praktikabilität wurden zudem Testdaten entfernt, deren Laufzeit im zweistelligen Minutenbereich liegt. Hierbei handelt es sich um `_freeze_importlib`, `_testembed` und `python`. Diese würden die Gesamtausführungszeit des Experiments um mehr als einen Tag verlängern, was hier nicht gerechtfertigt scheint. Wie in früheren Abschnitten bereits gezeigt wurde, ist bei derart langen Ausführungszeiten die Laufzeit von Lesen und Schreiben erwartungsgemäß ohnehin vernachlässigbar. Ferner wird der verfügbare Heap auf 16GB verdoppelt, um durch den Garbage Collector (GC) verursachte Schwankungen der Ausführungszeit zu reduzieren.

**Vollständiger Graph** Da `heap-structure` und `steensgaardV2` angepasst wurden, können alte Laufzeitmessungen nicht wiederverwendet werden. Um einen Ausgangspunkt für die Auswirkungen von partiellem Lesen und Anhängen zu haben, wird deren Ausführungszeiten neu vermessen. Das Vorgehen ist dabei analog zu den übrigen Laufzeitmessungen in diesem Kapitel und wird auf *Workstation* durchgeführt. Da das Werkzeug dieselbe `.sf`-Datei als Ein- und Ausgabe verwendet, wird eine neue Kopie vor jeder Ausführung erzeugt.

**Bedarfsorientiert Lesen + Schreiben** Um die Werkzeuge bedarfsorientiert lesend vermessen zu können, benötigt man zunächst entsprechende IR-Spezifikationen. Da es derzeit keine Werkzeugunterstützung[3] hierfür

[1] Dieses Problem ließe sich zwar prinzipiell lösen, es ist in diesem Kontext aber zu aufwändig. Würde man in der Entwicklung auf dieses Problem stoßen, so würde man natürlich zunächst die Repräsentation der gespeicherten Daten optimieren.

[2] Hier auf *Server* auszuweichen und einen noch größeren Heap zu verwenden, erscheint wenig zielführend, da man aller Wahrscheinlichkeit nach ohnehin an der 2 GB Grenze scheitern würde. Außerdem haben Messungen auf *Server* den Nachteil, dass sie erheblich schwerer ohne Messfehler durchführbar sind, da es sich um eine geteilte Ressource handelt.

[3] Würde es sich nur um ganze Typdefinitionen handeln, so könnte man sich etwa an den Importstatements für generierte Typen orientieren.

gibt, werden die beiden Werkzeugspezifikationen gekürzt, neu übersetzt und auf Korrektheit geprüft. Dies wird solange wiederholt, bis keine weiteren irrelevanten Definitionen mehr gefunden werden. Dieser Vorgang hat neben dem Zeitaufwand den Nachteil, dass die IR-Spezifikation anstelle von mit `!onDemand` markierten Felddefinitionen nichts spezifiziert. Es wird also eine partielle IR-Spezifikation vermessen. Die dabei entstehenden IR-Spezifikationen sind jeweils bedeutend größer als die nahezu minimale IR-Spezifikation in Abschnitt 7.8.3.

**Bedarfsorientiert Lesen + Anhängen** Um aus einer bedarfsorientiert lesenden und vollständig schreibenden Implementierung eine, die lediglich anhängt, zu erzeugen, muss nur beim Erstellen des `SkillStates` der entsprechende Modus gewählt werden.

**Validierung** Die Messungen werden validiert, indem zunächst die vollständig serialisierende Lösung einmal ausgeführt wird. Das Ergebnis wird separiert und nach der Ausführung der beiden Werkzeuge mittels SKilL-Diff (siehe Anhang A) verglichen. Da SKilL/Scala von der Möglichkeit, unbenutzte Typ- und Felddeklarationen beim Serialisieren zu entfernen, je nach Schreiben und Anhängen unterschiedlich Gebrauch macht, werden alle Meldungen zu fehlenden Typen und Feldern ignoriert.[1] Da es sich um Prototyp handelt, die nicht ausführlich getestet wurden, wird die Validierung ungeachtet der Konsequenzen auf den Messfehler nach jeder Messung durchgeführt.

**Messungen** Die Ergebnisse der Messungen sind getrennt nach Werkzeug in Abbildung 7.12 dargestellt. Um das asymptotische Verhalten zu quantifizieren, findet sich das Ergebnis einer Regressionsanalyse in den Tabellen 7.17 und 7.18. Eine Übersicht der benötigten Code-Mengen für die Realisierung der Werkzeuge ist in Tabelle 7.19 zu finden. Die Implementierung von

[1] Diese konnten bei Erstellung des Testaufbaus eindeutig auf Typen und Felder eingeschränkt werden, die bei bestimmten `.sf`-Dateien nicht vorkommen und unterscheiden sich folglich auch je nach Testdatum.

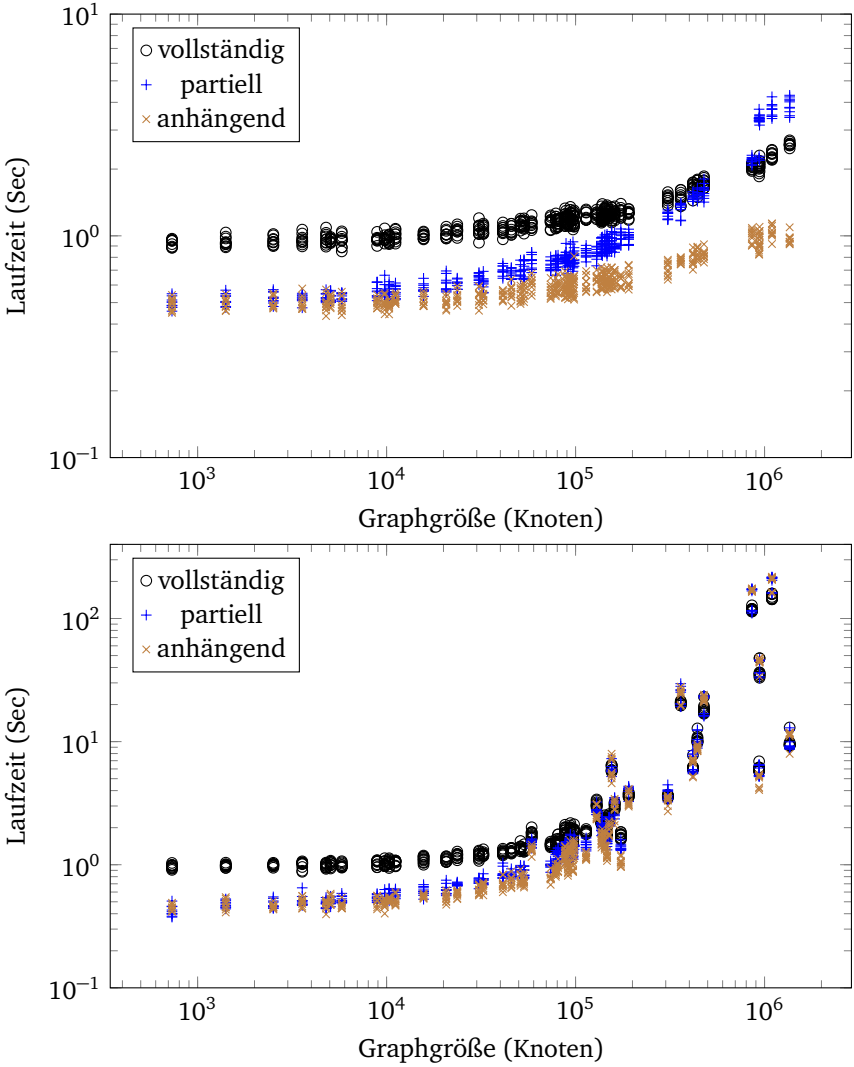



Abbildung 7.12.: Laufzeit nach Lese- und Schreibmethode (*vollständig* lesend und schreibend; *partiell* lesend und vollständig schreibend; partiell lesend und *anhängend*). Oben `heap-structure`, unten `steensgaardV2`.

| | vollständig | partiell | anhängend |
|---|---|---|---|
| $10^6$ Knoten | 1.194*** | 2.555*** | 0.445*** |
| | (0.012) | (0.024) | (0.009) |
| Constant | 1.035*** | 0.531*** | 0.539*** |
| | (0.004) | (0.009) | (0.003) |
| Observations | 510 | 510 | 510 |
| Adjusted $R^2$ | 0.950 | 0.959 | 0.827 |
| F Statistic (df = 1; 508) | 9711.089*** | 11773.780*** | 2425.932*** |
| *Note:* | | *p<0.1; **p<0.05; ***p<0.01 | |

Tabelle 7.17.: Regression der Laufzeit nach Knoten für `heap-structure`.

| | vollständig | partiell | anhängend |
|---|---|---|---|
| $10^6$ Knoten | 55.723*** | 69.153*** | 75.003*** |
| | (2.944) | (3.885) | (4.162) |
| Constant | −2.152** | −3.553** | −4.096*** |
| | (1.067) | (1.408) | (1.508) |
| Observations | 510 | 510 | 510 |
| Adjusted $R^2$ | 0.412 | 0.383 | 0.389 |
| F Statistic (df = 1; 508) | 358.353*** | 316.858*** | 324.819*** |
| *Note:* | | *p<0.1; **p<0.05; ***p<0.01 | |

Tabelle 7.18.: Regression der Laufzeit nach Knoten für `steensgaardV2`.

`heap-structure` benötigt im partiellen Fall weniger Code, da ein großer Patternmatching-Ausdruck vereinfacht werden kann, wenn nur relevante Typen existieren. Im Fall von *anhängend* benötigt man zwei weitere Imports, um die Parameter beim Öffnen der `.sf`-Datei setzen zu können.

**Interpretation** Zunächst fällt auf, dass die partiellen Implementierungen mit nahezu 100k Zeilen weniger Code auskommen. Obwohl es sich dabei überwiegend um Code handelt, den der Werkzeugbauer nicht direkt sieht,

| | heap-structure | | | steensgaardV2 | | |
|---|---|---|---|---|---|---|
| | Anbindung | Analyse | Skill | Anbindung | Analyse | Skill |
| vollständig | 119k | 385 | 5164 | 119k | 532 | 5168 |
| partiell | 19k | 371 | 747 | 8084 | 532 | 278 |
| anhängend | 19k | 373 | 747 | 8084 | 534 | 278 |

Tabelle 7.19.: Code-Menge für die Implementierung der beiden Werkzeuge nach Serialisierungsstrategie. *Anbindung* und *Analyse* entspricht dem von `cloc` angegebenen *code*. *Skill*-Spezifikation entspricht `wc -l`.

bedeutet ein Wegfall von grob 80% des Codes eine spürbare Verbesserung der Compilezeit. Ebenso wirkt sich weniger Code positiv auf die Laufzeit von Classloader und JIT aus.

`heap-structure` zeigt klar das konstruktionsbedingt zu erwartende Verhalten. Das Werkzeug erzeugt mit nachweislich linearem Aufwand aus dem eingelesenen Graphen eine Datenstruktur für die nachfolgenden Zeigeranalysen. Die konstanten Kosten sind dabei für partielle Ansätze deutlich geringer als für vollständige. Ebenso skaliert die partiell schreibende Variante erkennbar schlechter und die anhängende erkennbar besser.

Für `steensgaardV2` ist deutlich zu erkennen, dass die Laufzeit des Anwendungscodes und damit des Werkzeugs superlinear ist. Hier zeigt sich bis etwa 100k Knoten dasselbe Verhalten wie beim ersten Werkzeug. Danach werden die Schwankungen der Laufzeit schnell größer als der Einfluss der Serialisierungsmethode auf die Gesamtlaufzeit. Beschäftigt man sich näher mit den Testdaten, so fällt auf, dass die Regression hier von *bash* und *bash43* dominiert wird. Streicht man diese, so sind die drei Steigungen nahezu identisch. Im Vergleich bleibt die vollständige Kurve aber auch dann noch etwas flacher. Ferner fällt bei der Betrachtung der konkreten Laufzeiten der drei langsamsten Läufe (*bash*, *bash43* und *gnuplot*) auf, dass die vollständige Implementierung nur um wenige Sekunden schwankt. Die beiden bedarfsorientierten Implementierungen dagegen haben quasi zwei Zustände: Einen, in dem sie sich im Wesentlichen gleich wie die vollständige Implementierung verhalten, und einen anderen, in dem sie einen zweistelligen Sekundenbe-

reich länger laufen. Dieses Verhalten ist in Abb. 7.12 ohne Zuhilfenahme einer Lupe für beide Messpunkte oben rechts erkennbar. Eine definitive Ursache hierfür konnte nicht ermittelt werden.

Im Sinne der in diesem Abschnitt aufgestellten Hypothesen hätte man natürlich `heap-structure` akzeptieren oder ein `steensgaardV3` implementieren können, das einen klar linearen Ressourcenverbrauch aufweist. Dies hätte zudem den Vorteil gehabt, dass man am Anfang des Abschnitts die Wahl der Testdaten nicht hätte einschränken müssen. Es wäre aber ebenso langweilig wie unsachlich. Betrachtet man die Ausführungszeiten von `heap-structure`, so fällt auf, dass der Break-even-Punkt von vollständigem und bedarfsorientiertem Lesen erst bei etwa einer Million Knoten liegt. Die anhängende Variante ist lediglich dreimal so schnell. Dies legt nahe, dass es global gesehen möglicherweise effizienter wäre, das Format und vor allem die Implementierung zu vereinfachen, indem man Anhängen von der Anforderungsliste streicht und die dadurch frei gewordenen Ressourcen für eine performantere Implementierung von bedarfsorientiertem Lesen verwendet. Ferner wären in diesem Fall alle gelesenen Indexbereiche zusammenhängend. Dadurch könnte ohne großen Aufwand in jedem Fall eine Repräsentation verteilter Felder durch ein Array anstelle einer HashMap gewählt werden.

Nichtsdestotrotz ist klar erkennbar, dass alle drei Varianten unmittelbar anwendbar sind und auch hier SKilL die Skalierbarkeit der vermessenen Werkzeuge nicht einschränkt.

### Zusammenfassung

In diesem Abschnitt konnte gezeigt werden, dass bedarfsorientiertes Lesen sowie Anhängen wie spezifiziert funktionieren und partielle IR-Spezifikationen die Gesamtperformance steigern können, falls die Laufzeit des Anwendungscodes entsprechend kurz ist und Resultate an die bestehende Austauschdatei angehängt werden.

## 7.5. Benutzbarkeit

Im Bereich der Benutzbarkeit sind als Evaluation zunächst die auf SKilL aufbauenden studentischen Arbeiten anzuführen [Kai15; BDF+16; Har16; Prz16].

Daneben gibt es einige Konzepte, die offensichtlich die Benutzbarkeit verbessern. So entspricht die Typisierung der IR-API in Programmiersprachen, die es unterstützen, der Typisierung aus der IR-Spezifikation.

Kommentare werden exportiert und so formatiert, dass das Dokumentationssystem der Programmiersprache damit umgehen kann. Dies ermöglicht beispielsweise in Java MouseOverDocumentation in IDEs. Kommentare können zusammen mit der IR-Spezifikation nach Doxygen exportiert werden und so in eine gut navigierbare Form gebracht werden. Obwohl das natürlich in weiten Teilen eine Leistung von Doxygen [vHee13] darstellt, ist der Komfortgewinn und die Navigierbarkeit durch die IR-Spezifikation ein erhebliches Plus gegenüber der reinen Textform. Ein Beispiel dieser Darstellung findet sich in Abb. 7.13.

Ein vergleichbares Werkzeug für Daten wurde von Moritz Rathgeber im Rahmen seiner Diplomarbeit geschaffen [Rat17]. Dieses verwendet die von SKilL durch das API bereitgestellte Typinformation um eine navigierbare Darstellung des in der `.sf`-Datei enthaltenen Graphen zu ermöglichen. Ein Beispiel der Darstellung findet sich in Abb. 7.14. Wie für studentische Arbeiten üblich, ist hier trotz erkennbarem Mehrwert auch direkt ein gewisses Verbesserungspotenzial erkennbar.

Ferner ist die Zustandsverwaltung und API aktueller SKilL-Implementierungen so gestaltet, dass man problemlos mehrere `.sf`-Datei gleichzeitig bearbeiten und modifizieren kann. Dies ist für eine Kapselung in Bibliotheken und für die Verarbeitung in Threads unerlässlich.

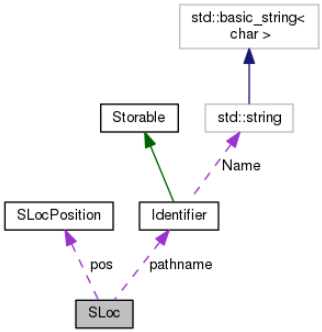
Die Möglichkeit, Teilspezifikationen zu verwenden, verbessert die Benutzbarkeit, da sich hierdurch die Zahl der zu betrachtenden Typen und Felder reduziert.

Auch das Aufteilen von IR-Spezifikationen in mehrere Spezifikationsdateien scheint eine offensichtliche Verbesserung zu sein, die sich etwa auch

### SLoc Klassenreferenz

`#include <SLoc.h>`

Zusammengehörigkeiten von SLoc:




#### Öffentliche Attribute

| | |
|---|---|
| **Identifier** | **pathname** |
| **SLocPosition** | **pos** |

#### Ausführliche Beschreibung

An object of this class is used to store source-line-information. Such objects are used in all nodes of the IML-Graph.

Abbildung 7.13.: Mittels SKilL/doxygen und doxygen generierte navigierbare Darstellung der IR-Spezifikation aus Listing 7.1 für SLoc.

in XSD findet, in IML jedoch fehlt. Hier ist schlussendlich IDE-Support erforderlich, da das Auffinden von Typdefinitionen sonst unnötig erschwert wird. Dieser IDE-Support wurde für SKilL in Form eines Studienprojekts [BDF+16] von Studenten entwickelt und ist zumindest in diesem Punkt sehr gut zu benutzen.

Darüber hinaus werden hier zwei kurze Beispiele angeführt, die das zugrundeliegende Benutzbarkeitskonzept illustrieren und dessen Wirksamkeit untermauern sollen.

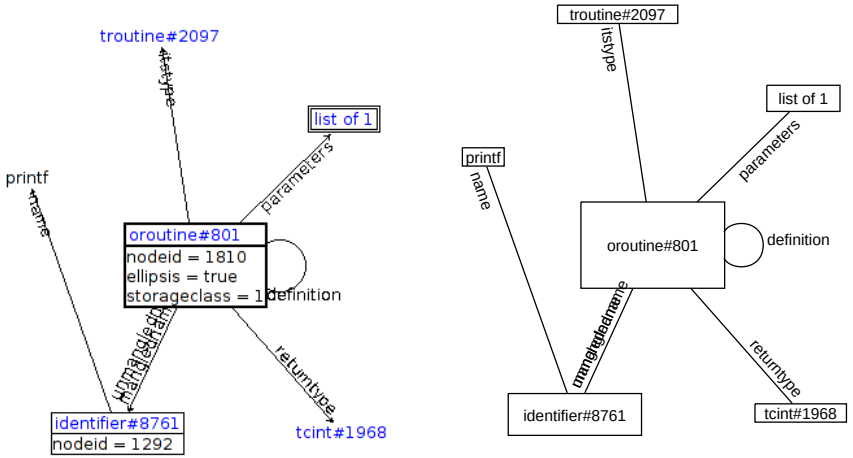



Abbildung 7.14.: Mittels Graphvisualisierung von Moritz Rathgeber erstellte Ansicht des Knotens ORoutine Nummer 801 in `aget`. Links: Screenshot der navigierbaren Darstellung. Rechts: Darstellung derselben Ansicht im exportierten PS.

### 7.5.1. Used Source Lines of Code

In diesem Abschnitt wird ein recht eingängiges Beispiel der Lavaflow-basierten Entwicklung [Fel15] demonstriert. Hierfür wird ein kleines Werkzeug mit dem Ziel implementiert, für die in dieser Arbeit verwendeten Testdateien die tatsächlich genutzten Quellcodezeilen (USLoC) zu berechnen. Bauhaus enthält ein solches Werkzeug bereits (Name: `stats`). Dieses hat aber Abhängigkeiten an `libConcurrency`, was bedeutet, dass dessen Ausführung eine Threadanalyse erfordert, welche auf den hier verwendeten Daten nicht durchgeführt wurde und zum Teil auch nicht durchführbar ist.

Nun scheint das Aufsammeln von Quellcodeverweisen keine besonders

schwere Aufgabe zu sein. Tatsächlich ist es ausreichend, die IML-Spezifikation so auf `SLocs` zu beschränken, dass nur noch Dateiname und Zeile übrig bleiben. Werden diese dann in einem Set gesammelt, so ist die Größe des Sets das gewünschte Resultat, vorausgesetzt dass Zeilen in gleicher Datei mit gleicher Nummer für das Set gleich wären. Dabei werden zwar potentiell Zeilen ignoriert, die z.B. nur eine schließende Klammer enthalten. Diese wären aber aufgrund der vorliegenden Daten nicht zuverlässig erkennbar, da in Bauhaus an einzelne Knoten ein Anfang, aber nur selten ein Ende angehängt wird. Kommentare tragen jeweils zur Zahl der Quellzeilen bei, im Fall mehrzeiliger Blockkommentare aber unter Umständen nicht jede betroffene Zeile. Ebenso würde, falls ein mehrzeiliger String vorhanden wäre, dieser nur eine Zeile beitragen. Die Abweichungen von rein syntaktischen Verfahren sind hier nicht relevant.

Der erste Schritt besteht darin, die IR-Spezifikation entsprechend zu kürzen – das Ergebnis findet sich in Listing 7.1. Hätte man Zugriff auf eine funktionierende SKilL-IDE, so wäre dieser Schritt mit dem Setzen zweier Häkchen und dem Exportieren der IR-Spezifikation oder einer direkt generierten API beendet.

Als nächstes muss eine Programmiersprache gewählt und das eigentliche Werkzeug implementiert werden. Da die Ausführungszeit irrelevant, die Entwicklungszeit aber von Bedeutung ist,[1] wurde Scala gewählt. Der vollständige Code für die erste Version[2] des Werkzeugs findet sich in Listing 7.2. Hier muss man der Fairness halber dazu sagen, dass es durchaus längere Versionen in Zwischenschritten gab, die der Prüfung der Ergebnisse dienten, da diese im ersten Moment unplausibel erschienen. Es hat sich aber schnell gezeigt, dass dieses Verfahren auch genutzte Standardbibliotheksfunktionen mit zählt, was im vorgesehenen Anwendungsfall aber akzeptabel scheint.

[1] Das Werkzeug wird zur Aktualisierung der Daten eine handvoll Ausführungen erfahren und dann vergessen werden. Hier etwas Effizientes zu entwickeln hält nur vom Schreiben ab.

[2] Die tatsächlich erste Version wurde gegen eine IR-Spezifikation geschrieben, in der `pathname` ein `string` und kein `Identifier` ist, was die Implementierung drei Zeilen kürzer macht, da `toPair` nicht erforderlich ist. Die Darstellung wurde hier nachträglich angepasst, da sich hierdurch die Kernaussage nicht verändert und die gegebene Darstellung im Sinne der Vergleichbarkeit und Nachvollziehbarkeit erwartungsgemäß den größeren Mehrwert bietet.

```
typedef natural
 @min(0) v64;

/** Position of a sloc. Taken from Ada source. */
SLocPosition {
  natural line;
}

/**
 * An object of this class is used to store source-line-
     information.
 * Such objects are used in all nodes of the IML-Graph.
 */
SLoc {
  SLocPosition pos;
  Identifier pathname;
}

Identifier : Storable {
  string Name;
}

@abstract
Storable {}
```

Listing 7.1: IR-Spezifikation für USLoC counting

Ob der Einfachheit der Aufgabe und der Nutzbarkeit der SKilL-Infrastruktur, lässt sich mühelos in sehr kurzer Zeit eine erste Implementierung schreiben. Zur Publikation des Resultats in dieser Arbeit benötigt man die Ausgabe aber in einer Form, die sich gut in ein LaTeX-Dokument übernehmen lässt. Außerdem wäre es reine Zeitverschwendung und eine sinnlose Fehlerquelle, würde man alle Daten von Hand vermessen. Daher wurde das Werkzeug noch ein paar Minuten weiterentwickelt, um diese Prozesse zu automatisieren. Wenn alle relevanten Testdateien vermessen werden sollen, so ist es hilfreicher, das Verzeichnis zu übergeben und die relevanten Austauschdateien heraus zu filtern. Dies ist per Konvention über die

```
import iml.api.SkillFile
import iml.SLoc

object Main {
  def main(args : Array[String]) {
    val file = SkillFile.open(args(0))
    val lines = file.SLoc.map(toPair).toSet.size
    println(s"${args(0)}: $lines")
  }
  def toPair(x : SLoc) =
   if (null == x.pathname) (null, 0)
   else (x.pathname.Name, x.pos.line.toInt)
}
```

Listing 7.2: USLoC counting: Erste Implementierung

Dateiendung möglich. Das Ergebnis wird in einem sequentiellen Container zwischengespeichert, um es später in eine Beschreibung einer LaTeX-Tabelle auf die Standardausgabe zu drucken, damit es der Werkzeugnutzer nur noch in das Dokument kopieren muss. Um die Tabellen deutlich lesbarer zu gestalten, wird die Sequenz sortiert. Diese Sortierung erfolgte zunächst nach Code-Zeilen, wurde aber später der Vergleichbarkeit mit anderen Tabellen halber durch eine Sortierung nach Knotenzahl ersetzt. Zu guter Letzt wird noch die ursprüngliche Dateigröße gemessen, um dem Leser einen Vergleichswert zu bieten. Außerdem erhält der ausgegebene Tabelleninhalt eine Kopfzeile. Das Ergebnis findet sich in Listing 7.3. Man beachte, dass eine Wiederverwendung des Werkzeugs nicht vorgesehen ist.

Die tatsächliche Darstellung der Tabelle 7.7 erfordert noch eine Bildunterschrift und ein Aufsplitten, da die Tabelle für eine Seite zu lang ist. Dieser Schritt ist nicht sinnvoll automatisierbar.

Bemerkenswert ist, dass die Gesamtlaufzeit des Werkzeugs auf *Workstation* nur etwa 16 Sekunden beträgt. Das ist insofern ein faszinierender Wert, als die Bauhaus-`function_names`-Implementierung schon für das größte Beispiel alleine über 20 Sekunden benötigt. Ähnliches würde auch für `stats`

```
import java.io.File
import java.nio.file.Files
import iml.api.SkillFile
import iml.SLoc

object Main {
 def main(args : Array[String]) {
  println((for (f <- new File(args(0)).listFiles if f.
      getName.endsWith(".iml.sf")) yield {
   val file = SkillFile.open(f.getAbsolutePath)
   val lines = file.SLoc.map(toPair).toSet.size
   val nodes = file.filter(_.superName.isEmpty).map(_.size)
       .sum + file.String.size
   (nodes, f.getName.replace(".iml.sf", ""), lines, "%.3f".
       format(Files.size(f.toPath) * 1e-6f))
  }).toSeq.sortBy(_._1).map(_.productIterator.toSeq.tail.
      mkString(" & ")).mkString("File Name & USLoC & File
      Size (MB)\\\\\\hline\n", "\\\\\\n", ""))
 }
 def toPair(x : SLoc) =
  if (null == x.pathname) (null, 0)
  else (x.pathname.Name, x.pos.line.toInt)
}
```

Listing 7.3: USLoC counting: Finale Implementierung

gelten, wenn es denn anwendbar wäre, da IML immer den kompletten Graphen aufbaut.

### 7.5.2. SLoC Unification

Ein zweites, vergleichbares Experiment befasst sich mit der Frage, ob SLoCs unifiziert werden können, um `.sf`-Dateien kleiner zu machen. Für eine praktische Umsetzung einer Unifikation müsste sichergestellt werden, dass es kein Werkzeug gibt, welches ausnutzt, dass SLoCs nicht unifiziert sind.

Um entscheiden zu können, ob sich der Aufwand überhaupt rechnen würde, wird das Einsparpotential untersucht. Verwendet man hier Scala,

```scala
import scala.collection.mutable.HashMap
import iml.SlocPosition
import iml.api.SkillFile

object Main {
 def main(args : Array[String]) {
  val sf = SkillFile.open(args(0))

  println(s"found: ${sf.SlocPosition.size} positions")
  val positions = new HashMap[(Int, Int), SlocPosition]
  for (p <- sf.SlocPosition)
    positions.getOrElseUpdate((p.Line.toInt, p.Column.toInt
        ), p)

  println(s"found: ${positions.size} unique positions")

  for (s <- sf.Sloc if s.Pos != null; p = s.Pos)
    s.Pos = positions((p.Line.toInt, p.Column.toInt))

  val used = positions.values.toSet
  for (p <- sf.SlocPosition if !used(p))
    sf.delete(p)

  sf.close
 }
}
```

Listing 7.4: Unifikation von SlocPositions

so kann man analog zum vorangegangen Abschnitt vorgehen. Hierbei wird relativ schnell klar, dass sich in den meisten `.sf`-Dateien keine einzige SLoC einsparen lässt, da diese zu viel Kontextinformation tragen. Dies gilt jedoch nicht für deren Positionsknoten, die lediglich Zeile und Spalte enthalten.

Da die tatsächlich eingesparten Werte für diese Arbeit nicht relevant sind, werden hier nur die Zahlen für `bash43.iml.sf` verwendet, einen Graphen mittlerer Größe ohne Analyseergebnisse. Werden die Positionen nach Zeile und Spalte unifiziert, so kommt man schnell zu dem Ergebnis,

dass von 226747 Knoten nur 105653 benötigt werden. Das ist angesichts der insgesamt etwa 860k Knoten zunächst ein erheblicher Teil. Es ist aber zu bedenken, dass es sich um relativ kleine Knoten handelt. Um die Auswirkung auf die Dateigröße zu messen, können die überflüssigen Knoten einfach ersetzt, gelöscht und die `.sf`-Datei neu geschrieben werden. Dies ist in Listing 7.4 implementiert. Nach der Ausführung ist die Dateigröße von vormals 18,5 auf 18,2 MB gesunken (etwa 2 %). Es ist zu erwarten, dass Graphen, die viele Analyseergebnisse enthalten, diesen Wert verschwindend klein werden lassen, sodass eine Umstellung einen geringen Effekt hätte.

Eine übersichtliche Darstellung in Abhängigkeit der Graphgröße, sowie eine statistische Evaluation, wie es in anderen Abschnitten üblich ist, wäre kein großer Arbeitsaufwand. Eine ausführliche Betrachtung dieser Frage würde im Rahmen dieser Arbeit aber zu weit führen.

### 7.5.3. Erkennung inkompatibler Änderungen

In diesem Abschnitt wird untersucht, inwieweit inkompatible Änderungen von Werkzeugen automatisch erkannt werden und wie aufwändig die Migration auf eine logisch kompatible Werkzeugspezifikation ist. Hierfür wird zunächst das Verhalten der PTA-Prototypen von Matthias Harrer [Har16] angewandt auf die hier genutzten Testdaten untersucht. Danach werden die in den Werkzeugen verwendeten IR-Spezifikationen aktualisiert und der Anpassungsaufwand untersucht.

Um dies zu erreichen wurde zunächst eine Kopie[1] der Testdaten ohne Analyseinformationen angelegt (siehe §7.1.5). Führt man die PTA-Prototypen auf den Testdaten aus, so erhält man für jedes Eingabedatum die folgende Fehlermeldung:

```
de.ust.skill.common.scala.api.TypeSystemError: the opened
file contains a type IMLRoot with super type attributable,
but none was expected
```

[1] Im Gegensatz zu den existierenden Bauhauswerkzeugen entsteht bei den Prototypen nicht zwingend eine neue `.sf`-Datei und eine Fernwirkung auf andere Messungen soll vermieden werden.

Hier hat der Parser korrekt erkannt, dass sich die Typhierarchie verändert hat und daher die Verarbeitung wie spezifiziert verweigert. Der neue Supertyp wurde eingeführt, um Typen korrekt darzustellen, welche nicht in der IML-Spezifikationsdatei dargestellt, aber in der IML-IR enthalten sind.

Um die IR-Spezifikation zu aktualisieren, muss zunächst entschieden werden, welche Teile gegenüber der Originalspezifikation verändert wurden. Ein Diff zwischen den Versionen der Spezifikation zeigt, dass neben einer neu eingefügten Spezifikationsdatei, die unverändert übernommen werden kann, noch ein Interface eingeführt wurde, mittels dessen alle spezifizierten IML-Knoten einen Verweis auf ihr Heap-Objekt untergemischt bekommen. Diese IR-Spezifikation mag ungewöhnlich erscheinen, ist jedoch leicht anzupassen. Inklusive neuer Generierung der Implementierung und erstem Compilerlauf beläuft sich hier der Arbeitsaufwand auf etwa 15 Minuten – das Upgrade auf den aktuellen Quellcode-Generator nicht mitberechnet.[1]

Es zeigt sich, dass der Code nicht übersetzt, da es zwei inkompatible Änderungen gab. Zum einen werden `Identifier` nicht mehr direkt auf Strings, sondern auf einen eigenen `Identifier`-Knoten abgebildet. Zum anderen wurde ein Feld minimal umbenannt. Beides lässt sich ohne nennenswerten Aufwand beheben und das Werkzeug kann danach wie gehabt verwendet werden. Wären die fünf Werkzeuge nicht jeweils gegen die komplette IR-Spezifikation, sondern gegen das von ihnen verwendete Subset geschrieben, so wären die Änderungen an dieser Stelle abgeschlossen. Stattdessen muss dieser Schritt für jedes der Werkzeuge wiederholt werden.

Da die Testdaten mit `Locations` etwa 10% größer sind, als aufgrund der Erfahrung mit der Bauhausimplementierung zu erwarten war, wurde versucht, das vom Prototyp eingeführte Interface vom Wurzeltyp nach unten in die relevanten Knoten zu verschieben. Entfernt man die Vererbungsbeziehung zwischen dem Wurzeltyp und dem Interface und generiert das API neu, so ergeben sich nach Neugenerieren fünf Gruppen von Übersetzungsfehlern

[1] Es hat sich zwischen Beginn der Arbeit und Durchführung der Evaluation das Kommandozeilen-Interface des Quellcode-Generators geändert. Die damit verbundene Anpassung der Erstellungsskripte (englisch build script) ist kein erwartbarer Aufwand. Mit einer groben Viertelstunde fällt dieser aber auch nicht hoch aus.

im Anwendungscode. Diese wurden benutzt, um die Typen zu identifizieren, welche das Interface tatsächlich implementieren sollten.

### 7.5.4. Zusammenfassung

In diesem Abschnitt wurde gezeigt, dass bei der Implementierung von SKilL diverse Konzepte umgesetzt wurden, um die Benutzbarkeit zu erhöhen. Außerdem wurde gezeigt, dass es mit geringem Aufwand möglich ist, auch große Werkzeugketten um nützliche neue Werkzeuge geringer Komplexität zu erweitern. Ferner wurde gezeigt, dass SKilL/Scala wie spezifiziert auf Veränderungen der IR-Spezifikation reagiert. Außerdem können diese Veränderungen auch in praxisnahen Szenarien ohne großen Aufwand bearbeitet werden.

## 7.6. Einfluss unterschiedlicher Feldrepräsentationen

In diesem Abschnitt wird der Einfluss auf die Ressourcennutzung von verschiedenen Feldrepräsentationsstrategien anhand eines realen Beispiels untersucht. Die dabei untersuchte Hypothese ist, dass die Nutzung von Daten in *ondemand*-Feldern einen messbaren negativen Einfluss auf die Ressourcennutzung hat, während die Nichtbenutzung dieser Felder die Ressourcennutzung positiv beeinflusst.

Um diese Messung durchzuführen wird eine einfache Implementierung der zyklomatischen Komplexität [McC76] verwendet. Dabei handelt es sich um eine einfache Analyse, die einen Großteil der strukturellen Informationen der Zwischendarstellung benötigt. Eine Implementierung existiert in Bauhaus in Form des Werkzeugs `imlmetrics`. Das Verhalten dieses Werkzeugs wurde eins zu eins, also inklusive aller Bugs, nachgebaut.[1] Der exakte Nachbau gestattet es, alle Messungen durch zeilenweisen Vergleich der Ausgaben mit der Referenzimplementierung zu validieren. Der Algorithmus arbeitet in beiden Fällen mithilfe einer rekursiven Exploration der von Funktionsrümpfen aus erreichbaren Teilgraphen. Die Bauhausimplementierung verwendet dabei einen Besucher (engl. Visitor), der der Struktur des Programms folgt. Die Neuimplementierungen (`mccabe`) verwendet stattdessen SKilL-Reflection. Ferner wird SKilL/Scala verwendet, da es sich hierbei um die einzige Quellcode-Generator-Back-End handelt, welches bei der Generierung der Anbindung `ondemand`-Hints berücksichtigt.

Um der Frage nachzugehen inwieweit das Laufzeitverhalten von der IR-Spezifikation abhängt, wurde diese in mehreren Abstufungen auf bedarfsorientiert lesende Felder umgestellt. Die Nulllinie stellt dabei die volle IR-Spezifikation dar (im Folgenden *Voll*). Die erste Abstufung besteht in der Entfernung aller Typ- und Felddefinitionen, bei denen aufgrund des Programmverhaltens nicht zu erwarten ist,[2] dass diese gelesen werden (im Folgenden *Partiell*). Die zweite Abstufung besteht in der Reduktion der IR-

[1] Dieser Arbeitsschritt hat etwa zwei Stunden für die korrekte Berechnung und zwei Arbeitstage für die Integration aller Bugs samt Tests erfordert.

[2] Eine automatische Berechnung dieser IR-Spezifikation ist mit der aktuellen Implementierung nicht möglich.

Spezifikation auf das vom Werkzeug statisch verwendete API (im Folgenden *Benutzt*). Hierbei werden insbesondere Subtypen benutzter Typen entfernt, deren Instanzen garantiert verwendet werden. Als letzter Schritt werden alle verbleibenden Felder mit `!ondemand` Hints versehen (im Folgenden *Hints*). Dadurch werden hier alle Felder bei Bedarf nachgelesen. Insbesondere ist der Zugriff auf spezifizierte Felder architekturbedingt ineffizienter als bei nicht spezifizierten Feldern (siehe §6.3.5). Für die Vermessung der neuen Implementierung werden die Parameter `-Xmx8G` und `-Xss4M` verwendet. Letzterer ist erforderlich, da die Standardstackgröße für die erhebliche Rekursionstiefe auf großen Graphen nicht ausreicht. Es werden alle verfügbaren Testdaten vermessen, um den Einfluss von eventuell vorhandenen Analyseergebnissen mit einzubeziehen.

### 7.6.1. Ausführungszeit

Für die Evaluation der Ausführungszeit wurde die bestehende Implementierung als Kontrolllinie verwendet. Die Ergebnisse sind in Abb. 7.15 dargestellt.

Bei kleinen Graphen zeigt sich das bereits bekannte Verhalten, bei dem die konstanten Kosten der vermessenen Anbindung ausschlaggebend sind. Diese fallen umso kleiner aus, je intensiver bedarfsorientiert gelesen wird. Dieser Effekt ist aber verglichen mit dem Einfluss der Programmiersprache erkennbar weniger gravierend. Die Skalierbarkeit wirkt insgesamt relativ vergleichbar. Man sieht zudem die von `bash` verursachten Ausreißer an gewohnter Stelle bei den beiden vollständig lesenden Implementierungen. Um den subjektiven Eindruck zu untermauern wurde eine Regression durchgeführt (siehe Tab. 7.20). Das relativ starke Ausreißen von imlmetrics für `bash` ist maßgeblich für das geringe $R^2$ verantwortlich und lässt einen direkten Vergleich der Steigung fragwürdig erscheinen. Die Verbesserung von *Voll* auf *Partiell* fällt mit einem Faktor 1,2 relativ gering aus. Dass die Veränderung von *Partiell* auf *Benutzt* vergleichbar ausfällt ist reiner Zufall.[1] Der Unter-

[1] Würde man die zyklomatische Komplexität mehrfach berechnen um dadurch die Zahl der Zugriffe auf Laufzeitobjekte zu erhöhen, würden sich diese Verhältnisse erwartungsgemäß unterschiedlich verschieben. Ein Ausbau der Messung würde hier aber zu weit führen.

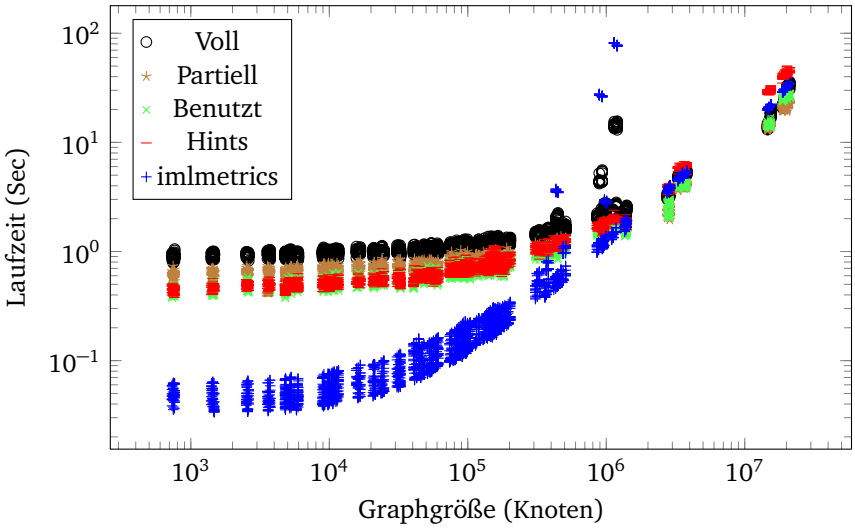



Abbildung 7.15.: Laufzeit der unterschiedlichen *mccabe*-Implementierungen.

schied zwischen *Partiell* und *Hints* fällt mit einem Faktor von etwa 2 zwar deutlich aber dennoch etwas geringer als erwartet aus.

### 7.6.2. Speicherverbrauch

In diesem Abschnitt wird die Hauptspeichernutzung untersucht. Hier zählt weniger der Bedarf des Prozesses, weil dieser wesentlich von den übergebenen Startparametern abhängt. Was hier entscheidend ist, ist im Wesentlichen die Belastung des GCs. Hierdurch wird die Performance der entsprechenden Lösung stark beeinflusst. Folglich werden drei Größen gemessen: Die maximale Größe des genutzten Heaps (in MB), die Menge des wieder eingesammelten Speichers (in MB) und die Zeit, welche vom GC verbraucht wird (in %). Die initiale Heapgröße wird der Vergleichbarkeit halber auf den Standardwert[1] auf *Workstation* gesetzt. Dies geschieht durch Hinzunahme

[1] Bestimmt mittels `java -XX:+PrintFlagsFinal -version`.

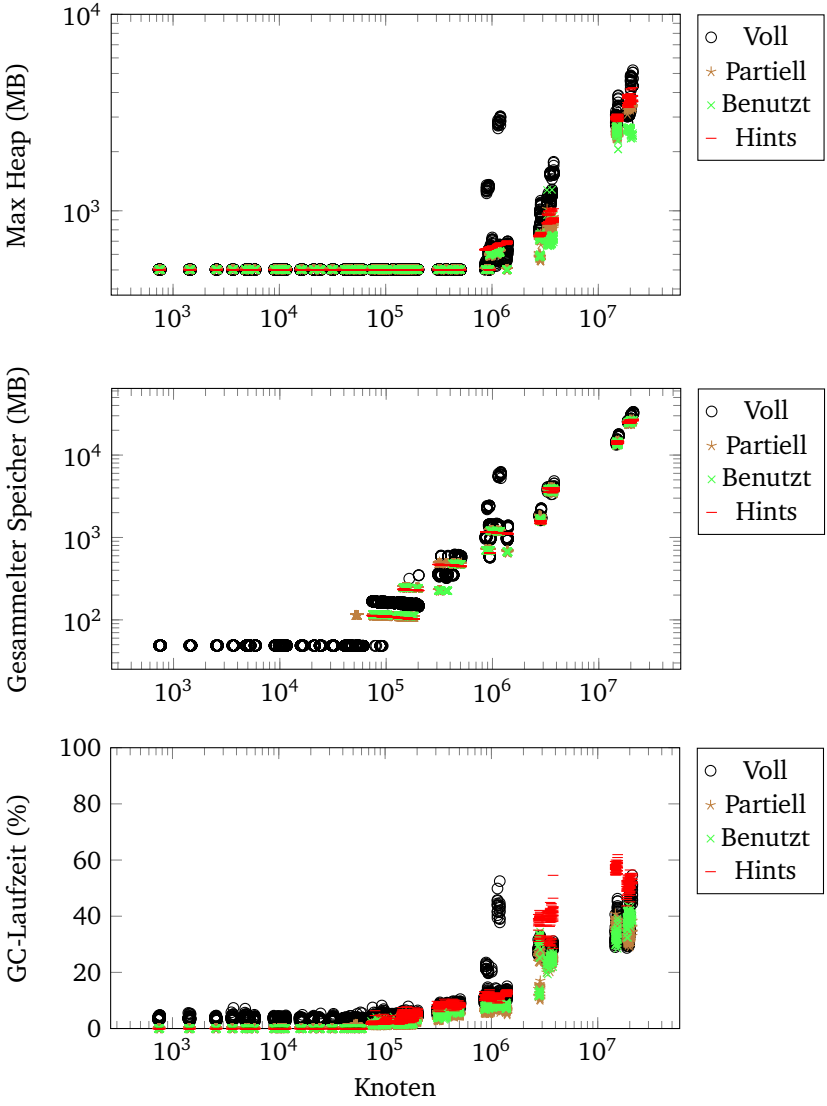



Abbildung 7.16.: Auslastung des GCs diverser `mccabe`-Implementierungen relativ zur Knotenzahl.

| | Voll | Partiell | Benutzt |
|---|---|---|---|
| $10^6$ Knoten | 1.182*** | 0.992*** | 1.209*** |
| | (0.006) | (0.002) | (0.004) |
| Constant | 1.046*** | 0.647*** | 0.436*** |
| | (0.027) | (0.008) | (0.016) |
| Observations | 2'820 | 2'820 | 2'820 |
| $R^2$ | 0.922 | 0.991 | 0.974 |
| F Statistic (df = 1; 2818) | 33'310.700*** | 307'113.200*** | 106'333.600*** |

| | Hints | IML |
|---|---|---|
| $10^6$ Knoten | 2.053*** | 1.526*** |
| | (0.004) | (0.032) |
| Constant | 0.280*** | 0.726*** |
| | (0.015) | (0.135) |
| Observations | 2'820 | 2'820 |
| $R^2$ | 0.992 | 0.446 |
| F Statistic (df = 1; 2818) | 338'418.100*** | 2'266.148*** |
| *Note:* | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.20.: Regression der Laufzeit von `mccabe` in Abhängigkeit der Knotenzahl.

des Startparameters `-Xms526385152`.

Um an die gesuchten Informationen zu gelangen, wurden gegenüber der reinen Laufzeitmessung die Startparameter außerdem um `-XX:+PrintGC` und `-Xloggc:gc.log` erweitert. Dadurch werden im Fall einer Garbage Collection in eine Logdatei Informationen geschrieben, aus denen die hier relevanten Informationen leicht abgeleitet werden können, wenn gleichzeitig die Gesamtlaufzeit gemessen wird. Es stellt sich dabei heraus, dass GCs die sehr unintuitive Eigenschaft besitzen, dass sie reproduzierbar negativ viel Speicher freigeben können. Ferner ergibt sich in der Darstellung ein Plateau für alle Messpunkte, bei denen keine Garbage Collection stattgefunden hat.

Die Ergebnisse dieser Messung sind in Abb. 7.16 dargestellt.

**Interpretation** Zunächst fällt auf, dass die bedarfsorientiert lesenden Implementierungen im Gegensatz zu *Voll* für einen weiten Teil der kleinen Graphen keine Collection verursachen. Insgesamt fällt der Speicherverbrauch von *Voll* höher aus, da hier auch ungenutzte Daten erzeugt werden. Überraschend ist, dass die Speichernutzung von *Benutzt* teilweise geringer ausfällt als die von *Partiell*. Dies kann zusätzlich mithilfe einer Regression nachgewiesen werden (siehe Tab. 7.21). Der Unterschied von etwa 27 Byte pro Objekt lässt sich intuitiv nicht erklären. Bei näherer Betrachtung der Rohdaten der Messung zeigt sich schnell, dass es Datenpunkte gibt, bei denen jeweils eine der beiden Implementierungen klar besser ist während bei manchen Datenpunkten keine konstant bessere Lösung auszumachen ist. Außerdem zeigt sich, dass die Messungen punktuell erheblich schwanken können. Betrachtet man nur die Ergebnisse für große Graphen bei denen der systematische Fehler durch spontane unnötige Allokation von Heapspace naturgemäß geringer ist, so zeigt sich die Regression jedoch in der Tendenz bestätigt. Die schlechtesten Ergebnisse über alle Läufe, des größten Datenpunkts, sind 4236 MB (Hints), 2483 MB (Benutzt), 3409 MB (Partiell) sowie 5193 MB (Voll). Eine Erklärung in Gestalt einer zu großen IR-Spezifikation für *Partiell* bietet sich insofern nicht an, als die Laufzeit von *Partiell* besser ist.

Die bedarfsorientierten Implementierungen verhalten sich beim wieder eingesammelten Speicher insgesamt relativ ähnlich. Auffällig ist hingegen, dass *Hints* bei großen Graphen einen erheblichen Anteil an GC-Laufzeit verursacht. Generell würden sich in dieser Richtung weitere Experimente anbieten. Die Auswirkungen des bedarfsorientierten Lesens sind insgesamt allerdings wenig erforscht. Weil die Untersuchung der Auswirkungen des bedarfsorientierten Lesens im Rahmen dieser Arbeit erschöpfend abgeschlossen ist, werden grundsätzlich, bis auf den nachfolgenden Abschnitt, keine weiteren Experimente dazu durchgeführt. Würde man sich intensiv mit diesem Thema beschäftigen, so wäre zu erwarten, dass dies auch eine Anpassung der Architektur der Anbindung nach sich ziehen würde.

| | Voll | Partiell | Benutzt | Hints |
|---|---|---|---|---|
| $10^6$ Knoten | 166.866*** | 140.492*** | 113.048*** | 161.744*** |
| | (1.094) | (0.303) | (0.402) | (0.275) |
| Constant | 504.088*** | 469.867*** | 479.543*** | 475.526*** |
| | (4.623) | (1.281) | (1.698) | (1.161) |
| Observations | 2'820 | 2'820 | 2'820 | 2'820 |
| $R^2$ | 0.892 | 0.987 | 0.966 | 0.992 |
| F Statistic (df = 1; 2818) | 23'271.750*** | 215'045.000*** | 79'212.090*** | 346'496.800*** |
| *Note:* | | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.21.: Regression Max Heap in Abhängigkeit der Knotenzahl.

### 7.6.3. `recode` gegenüber bedarfsorientiertem Lesen

Die Untersuchung von bedarfsorientiertem Lesen anhand eines realen Beispiels hat zwar den Vorteil der Praxisnähe, es bleibt aber eine gewisse Unsicherheit, die durch die Ausführung der eigentlichen Analyse entsteht. Um dem zu begegnen, wurde das Neukodierungswerkzeug aus §7.3 gegen eine leere IR-Spezifikation implementiert. Um diese Messung prinzipiell mit der Vorherigen vergleichen zu können, wird ebenfalls Scala verwendet. Es wird nun die Laufzeit des ursprünglichen Werkzeugs mit der des Neuen verglichen. Der erhöhte Speicherbedarf verteilter Felder erfordert die Durchführung der Messung mit 16 anstelle der üblicherweise verwendeten 8 GB Heap. Davon abgesehen entspricht die Messung der in §7.3. Die Validierung wurde mit lediglich zwei Wiederholungen am 3.5.2017 durchgeführt, da zum einen aufgrund der Validierung in §7.3 keine Fehler zu erwarten sind und zum anderen die Ausführungszeit erheblich ist. Tatsächlich diente die Validierung hauptsächlich der Bestimmung einer ausreichenden Heapgröße für alle Testdaten.

Die Ergebnisse sind in Abb. 7.17 dargestellt. Dabei führt das Fehlen der IR-Spezifikation aufgrund geringerer konstanter Kosten bei kleinen Graphen zu erkennbar besseren Ausführungszeiten. Bei großen Graphen dreht sich

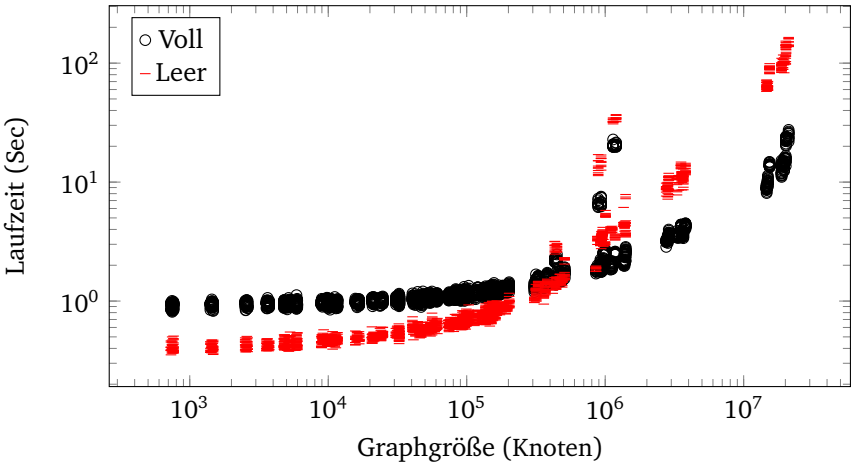



Abbildung 7.17.: Laufzeit der *recode*/Scala-Implementierungen.

dieses Verhältnis deutlich. Da es keinen Anwendungscode gibt, handelt es sich hierbei um eine Worst-Case-Abschätzung.

## 7.7. Beispiele der Änderungstoleranz

In diesem Abschnitt wird gezeigt, wie sich die in §2.2 dargestellten Änderungsszenarien und Anwendungsfälle in den vorangegangenen Abschnitten der Evaluation und der Testsuite des Quellcode-Generators (siehe §7.1) wiederfinden. Änderungen von Typen oder Feldern welche das Laufzeittypsystem betreffen, die aber keine in der Werkzeug-Spezifikation vorhandenen Typen, Vererbungsbeziehungen oder Felder tangieren, werden immer toleriert und daher im Folgenden nicht weiter berücksichtigt.

### 7.7.1. Veränderung des Typs eines Feldes

Dieser Fall tritt im Rahmen der Anpassung der PTA-Prototypen in §7.5.3 auf. Betroffen sind alle Felder vom Typ `Identifier`. Ein vergleichbares Beispiel ist innerhalb der generierten Tests der Testsuite die Kombination der Spezifikation `restrictionsCore.skill` mit dem Testdatum `partial.sf`. Hier wird für alle vermessenen Quellcode-Generator-Back-Ends zur Spezifikation ein Testprogramm generiert, welches das Testdatum liest und eine Fehlermeldung erwartet. Diese Tests sind in den erfolgreich ausgeführten Tests enthalten.

### 7.7.2. Hinzufügen eines Feldes

Hier seien wieder die PTA-Prototypen angeführt. Das interessante Verhalten tritt im positiven Sinne in §7.4 auf. Es gibt in dieser Arbeit kein Werkzeug, welches fälschlicherweise Instanzen von Typen erzeugt, deren Felder nicht vollständig in der Werkzeugspezifikation enthalten sind. In diesem Fall würden die in SKilL spezifizierten Standardwerte entstehen. Führt man den in §7.4 verwendeten PTA-Prototyp `steensgaardV2` ohne die davor erforderliche Vorverarbeitung durch `heap-structure` aus, so geschieht einfach nichts, da das Werkzeug davon ausgeht, dass es keine zu analysierende Nutzung von Zeigern gibt.

### 7.7.3. Entfernen eines Feldes

Diese Situation besteht bei allen evaluierten Werkzeugen mit partiellen Spezifikationen. Die Effekte werden am ausführlichsten in §7.6 beschrieben.

### 7.7.4. Veränderung einer Supertyp-Beziehung

Dieser Fall tritt im Rahmen der Anpassung der PTA-Prototypen in §7.5.3 auf. Ebenso tritt der Fall in der Testsuite bei der Spezifikation `graphInterface` auf. Diese definiert einen Typ `Node`, welcher von `AbstractNode` erbt. Es gibt diverse Testdaten, welche einen Typ `Node` ohne Supertyp verwenden. Folglich wird dieses Problem von allen mithilfe der hier verwendeten Quellcode-Generator-Back-Ends erzeugten Anbindungen korrekt erkannt. Der Fall, in dem die Testdaten einen Supertyp spezifizieren, tritt in der Testsuite unter der Spezifikation `advancedFancyTypes` für das Lesen von `localBasePoolOffset.sf` auf.

### 7.7.5. Hinzufügen oder Entfernen eines Typs

Das Werkzeug `heap-structure` (siehe §7.4) fügt in allen drei Inkarnationen den Typ `Location` und diverse Subtypen dessen der IR-Spezifikation hinzu. Die Werkzeuge zur Berechnung der zyklomatischen Komplexität in §7.4 arbeiten teilweise mit Instanzen unbekannter Typen mit bekannten Supertypen (siehe Beschreibung des dort als *Benutzt* bezeichneten Werkzeugs). Da die Entfernung der Typen konform zur Verwendung durch den Anwendungscode ist, verhalten sich die Werkzeuge alle funktional gleich.

### 7.7.6. Neues Werkzeug am Ende einer Werkzeugkette

Die meisten hier vermessenen Werkzeuge sind neue Werkzeuge am Ende der Werkzeugkette. Gute Beispiele sind die Berechnung der zyklomatischen Komplexität in §7.4 oder die Berechnung der USLoCs für die Testdaten in §7.5.1. Die Werkzeuge haben, wie erwartet, keine Auswirkungen auf andere Werkzeuge.

### 7.7.7. Neues Analysewerkzeug innerhalb der Werkzeugkette

Die PTA-Prototypen sind neue Werkzeuge innerhalb der Kette. Die Anhängenden bzw. partiell lesenden Werkzeuge sind dabei neue Werkzeuge, die die Semantik der alten erhalten und daher, wie demonstriert (siehe §7.4) getauscht werden können. Bezieht man die bestehende Bauhaus-Werkzeugkette mit ein, so sind diese keine neuen Werkzeuge innerhalb der Kette, da ihre Resultate von keinem bestehenden Werkzeug der Bauhaus-Werkzeugkette verarbeitet werden können.

### 7.7.8. Neues Transformationswerkzeug innerhalb der Werkzeugkette

In dieser Arbeit wurden keine Transformationswerkzeuge verwendet. In der Testsuite gibt es einige handgeschriebene Tests, die Transformationen durchführen. Diese finden insbesondere im Rahmen des Tests der korrekten Objektlöschung Anwendung.

## 7.8. Vergleich mit existierenden Lösungen

In diesem Kapitel wird SKilL direkt mit anderen Lösungen verglichen. Für den Vergleich wird für alle verwendeten Lösungen eine der Bauhaus-IML-Spezifikation entsprechende Spezifikation verwendet, damit Datensätze verwendet werden können, welche denen aus §7.1.5 entsprechen. Die Konvertierung der Datensätze in die jeweiligen Formate erfolgte einmalig vor Durchführung aller Messungen.

Zunächst war geplant neben der original Bauhaus-IML-Implementierung (in folgenden Messungen als IML bezeichnet) und nativer Java-Serialisierung auch EMF zu untersuchen. Dabei wurde der Eclipse 4.6.0 beiliegenden Generator verwendet. Es ist jedoch nicht gelungen, die entsprechenden Daten mit EMF zu erzeugen, da sich beim Konvertieren großer Datensätze Laufzeiten im zweistelligen Stundenbereich ergaben und diese bei den hier genutzten Testdaten zu einem nicht vertretbaren Aufwand geführt hätten. Einen Vergleich zwischen nativer Java- und C#-Serialisierung, sowie XML wurde bereits von Hericko et al. durchgeführt [HJR+03] und kann als Grundlage einer Abschätzung des zu erwartenden Verhaltens verwendet werden.

Um die Ansätze vergleichen zu können, wurde ein kleines, leicht implementierbares und leicht verifizierbares Werkzeug implementiert, welches die Namen aller in der Zwischendarstellung implementierter Funktionen ausgibt. Neben dem Ressourcenverbrauch der Ausführung wird davon unabhängig auch die Dateigröße der Zwischendarstellung sowie die Komprimierbarkeit derselben untersucht. Dabei werden im Wesentlichen die folgenden Hypothesen verfolgt:

- Dateigröße und Komprimierbarkeit von SKilL sind konkurrenzfähig
- Die Laufzeit mit SKilL ist verglichen mit IML und Java konkurrenzfähig
- Falls nur ein kleiner Teilgraph zu lesen ist, ist SKilL deutlich überlegen
- Der Speicherverbrauch einer SKilL-Lösung ist konkurrenzfähig

**Systematische Risiken** Bei der Abbildung zwischen verschiedenen Darstellungsformen ergeben sich Freiheitsgrade, die die Ergebnisse zugunsten einer Lösung beeinflussen können. So könnte man z.B. die in IML verwendeten Identifier direkt durch String-Referenzen repräsentieren, was bei SKilL und Java zu einer Effizienzsteigerung führen würde. Ebenso stellt sich die Frage, wie man einen kompletten, wurzelfreien Graphen in Java repräsentiert. Hier wurde für jeden Basistyp ein Array aller Instanzen serialisiert. Dies ist durch den Wunsch motiviert, bei der Deserialisierung eine Zugriffsmethode zu haben, die mit IML und SKilL vergleichbar ist. Das hiervon ausgehende Risiko sieht eher gering aus, da die Einflüsse bei großen Graphen nicht erheblich sind.

### 7.8.1. Dateigröße – Unkomprimiert

Das erste Experiment untersucht zunächst die Dateigröße, weil diese einen unmittelbaren Einfluss auf die IO-Performance hat. Sie hat aber auch Konsequenzen für die Benutzbarkeit, wenn es darum geht, Datensätze weiterzugeben. Dieser Punkt wird unter dem Aspekt Komprimierbarkeit noch entscheidender sein, wenngleich man bei komprimierten Datensätzen immer auch die Laufzeit für (De-)Kompression mitberücksichtigen muss.

Bei diesem Experiment wird das Verhältnis von Datei- und Graphgröße untersucht. Eine Validierung ist dabei nicht erforderlich, da die Messung exakt ist und die vermessenen Datensätze von anderen Experimenten stammen und dort validiert werden. Die Ergebnisse sind in Abbildung 7.18 dargestellt.

**Interpretation** Zunächst ist zu erkennen, dass sich IML und SKilL weitgehend ähnlich verhalten, wogegen Java erkennbar größere Austauschdateien erzeugt. Es ist ferner zu erkennen, dass die unterschiedlichen konstanten Kosten der Formate schon bei relativ kleinen Graphen nahezu vernachlässigbar sind. Dass SKilL die größten konstanten Kosten verursacht, ist darauf zurückzuführen, dass SKilL als einziges Format das gesamte Typsystem der enthaltenen Austauschdatei mitführt.

Hieraus ergibt sich auch als Korollar, dass die von der Serialisierung des

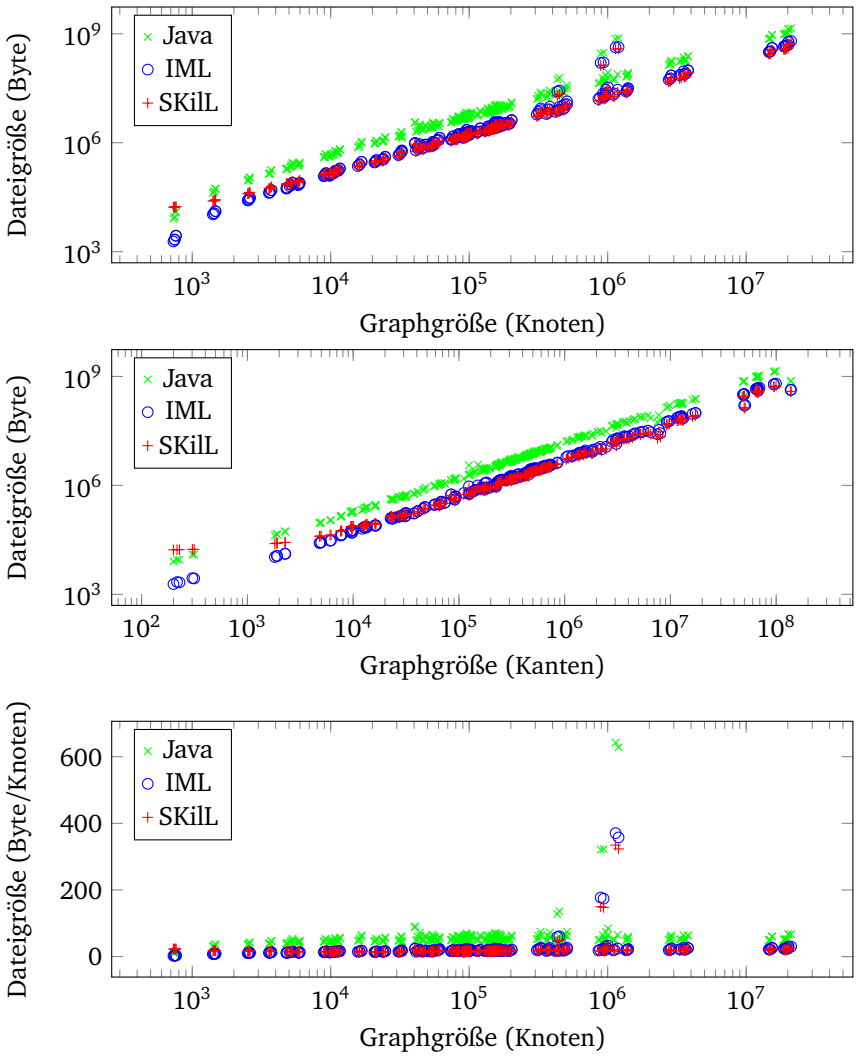



Abbildung 7.18.: Dateigrößen verschiedener Formate aufgetragen gegen Graphgröße.

| | IML | SKilL | Java |
|---|---|---|---|
| Knoten | 13.660*** | 10.893*** | 35.499*** |
| | (0.145) | (0.118) | (0.387) |
| Kanten | 3.040*** | 2.745*** | 5.362*** |
| | (0.030) | (0.024) | (0.079) |
| Observations | 282 | 282 | 282 |
| Adjusted $R^2$ | 0.997 | 0.997 | 0.996 |
| F Statistic (df = 2; 280) | 49643.100*** | 54850.060*** | 32940.620*** |
| *Note:* | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.22.: Regression der Dateigröße nach Knoten und Kanten mit erzwungenem Nulldurchgang.

| | IML | Java |
|---|---|---|
| SKilL | 1.172*** | 2.534*** |
| | (0.002) | (0.013) |
| Observations | 282 | 282 |
| $R^2$ | 0.999 | 0.992 |
| F Statistic (df = 1; 281) | 307973.700*** | 36101.920*** |
| *Note:* | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.23.: Dateigröße relativ zu SKilL.

gesamten Typsystems verursachten Mehrkosten nicht mehr ins Gewicht fallen, falls die serialisierten Graphen mehr als etwa 10000 Knoten haben.

Der intuitive Eindruck wird durch eine Regression der Dateigröße als Funktion der Knoten und Kanten für jedes untersuchte Format in Tabelle 7.22 untermauert. Da die konstanten Anteile insgesamt nicht signifikant sind, wurden der besseren Vergleichbarkeit halber die konstanten Kosten auf Knoten und Kanten umgelegt. Dies bedeutet, dass die Regression mit einem erzwungenen Durchgang durch den Nullpunkt durchgeführt wurde. Die Ergebnisse der Regressionsanalyse sind alle signifikant.

Dabei ist zu bemerken, dass in Bauhaus jedes Objekt eine sogenannte `Node_ID` hat. Hierbei handelt es sich um einen Integer, welcher sich in etwa wie eine Objekt-ID in SKilL verhält, welche aber von SKilL und Java wie reine Daten serialisiert werden. Würde man `Node_IDs` nicht in andere Formate übertragen, so würde man sich etwa 4 Byte pro Knoten sparen, könnte die Daten aber nicht mehr für manche existierende Bauhauswerkzeuge einsetzen. Für den Vergleich zwischen SKilL und Java ist dieser Umstand offensichtlich nicht signifikant. Da SKilL Knoten nicht direkt speichert und damit auch keine nennenswerten Kosten pro Knoten verursacht, die Regression in diesem Punkt aber signifikant ist, ist davon auszugehen, dass es sich bei den mittleren Kosten pro Knoten um weitere flache Nutzdaten, wie Zahlen, handelt.

Im Gegensatz zu der auf grobe Übersicht ausgelegten Abbildung, ist in der Regression klar zu erkennen, dass SKilL etwa 10% effizienter ist als IML und grob doppelt so effizient im Vergleich zu Java. Um diesen ungefähren Eindruck zu untermauern, haben wir eine Regression der Dateigröße anderer Formate in Abhängigkeit von SKilL durchgeführt (siehe Tabelle 7.23). Auch hier erzwingen wir einen Durchgang durch den Nullpunkt. Es ist hier leicht erkennbar, dass IML um einen Faktor von etwa 1,1 größere Austauschdateien verursacht. Bei Java beträgt der Faktor sogar etwa 2,5. Zusammenfassend gilt also, dass SKilL im Punkt Dateigröße das effizienteste untersuchte Format ist und, von extrem kleinen Graphen abgesehen, konkurrenzfähige und vor allem zufriedenstellende Resultate liefert.

### 7.8.2. Dateigröße – Komprimiert

In diesem Abschnitt wird die Komprimierbarkeit der untersuchten Serialisierungsformate analysiert. Dabei soll zum einen ausgeschlossen werden, dass eine geringere Dateigröße einzig durch strukturagnostische Komprimierung erreicht wurde, zum anderen soll die Eignung für Archive und für den Transfer über sehr langsame Übertragungsmedien untersucht werden.

Dabei wird zunächst auf das weit verbreitete `zip`-Format[Inf08] zurückgegriffen. Dieses wird in der Form *beste Laufzeit* und *beste Kompression* vermessen und nach den Parametern 1 (schnell) respektive 9 (kompakt) ge-

nannt. Für noch bessere Kompression wurde `7z`[Pav10] verwendet, welches nur im Modus beste Kompression (Parameter `-mx=9`) vermessen wurde. Da es sich hierbei im Rahmen der Arbeit nur um eine Randfrage handelt, wäre eine größere Auswahl an Kompressionswerkzeugen unverhältnismäßig.

Gemessen wird einerseits die Laufzeit der Kompressionsverfahren und andererseits die Kompression selbst. Die Laufzeitmessung erfolgt auf *Workstation* mit dem üblichen Testaufbau bei 3 Wiederholungen.[1] Die Messung der Kompression selbst ist exakt, weil Wiederholungen das Ergebnis nicht beeinflussen. Da die verwendeten Werkzeuge weit verbreitet sind ist eine Validierung nicht erforderlich.

Die Ausführungszeiten sind in Abbildung 7.19 dargestellt. Als Vergleichswert[2] wurde die Schreibzeit von C++ aus Abb. 7.10 übernommen. Hierdurch können die Auswirkungen einer standardmäßigen Kompression auf die Laufzeit abgeschätzt werden.

Die Kompression ist in Abbildung 7.20 relativ zur Originaldatei des eigenen Formats dargestellt, um die erzielte Kompression der Verfahren beurteilen zu können. Um der Frage nachgehen zu können, wie das Verhältnis der um Kompression ergänzten Verfahren zur hier vorgeschlagenen Ausgangssituation ist, ist in Abbildung 7.21 der Faktor relativ zur unkomprimierten `.sf`-Datei dargestellt. Hierdurch lässt sich der zu erwartende Kompaktheitsgewinn leicht einschätzen.

**Interpretation** Zunächst ist klar erkennbar, dass die Ausführungszeit der Kompressionswerkzeuge auf nahezu allen Datenpunkten deutlich über der Ausführungszeit der unkomprimierten Schreibimplementierung liegt. Der

[1] Das Experiment wurde ursprünglich mit 10 Wiederholungen durchgeführt, die Ausführungszeit ist mit etwa drei Tagen aber so hoch, dass dies die Reproduzierbarkeit negativ beeinflusst. Da die Ergebnisse selbst kaum schwanken, erscheinen drei Wiederholungen ausreichend.

[2] Eine Darstellung relativ zur Schreibzeit wurde evaluiert, führt aber insgesamt zu einer weniger intuitiven Darstellung ohne dabei die Übersichtlichkeit zu erhöhen. In den Rohdaten ist erkennbar, dass die Ausführungszeiten nur bezogen auf SKilL über einen weiten Teil der Testdaten je nach Methode um einen Faktor zwischen etwa 4 und etwa 60 zunehmen würde. Da es hier aber primär um die Frage des *ob* und nicht des *wie viel* geht, wurde entschieden, weiter mit den absoluten Zahlen zu arbeiten.

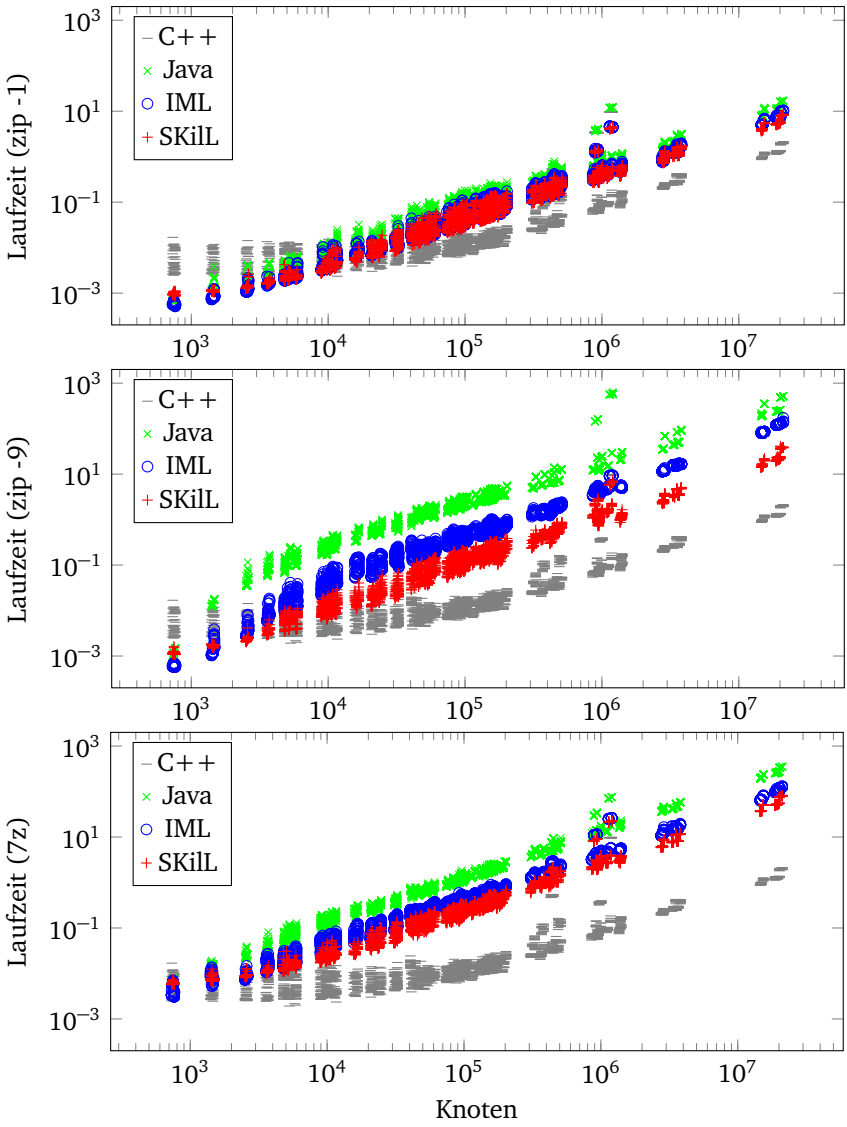



Abbildung 7.19.: Laufzeit der Kompression nach Format. C++ bezeichnet die Schreibzeit für unkomprimiertes SKilL in C++.

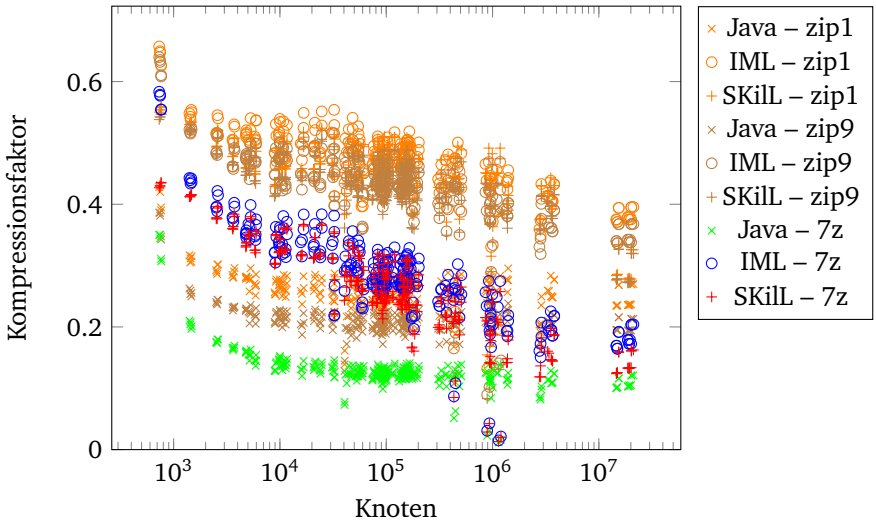



Abbildung 7.20.: Kompression nach Format als Faktor der komprimierten Dateigröße – kleiner ist besser.

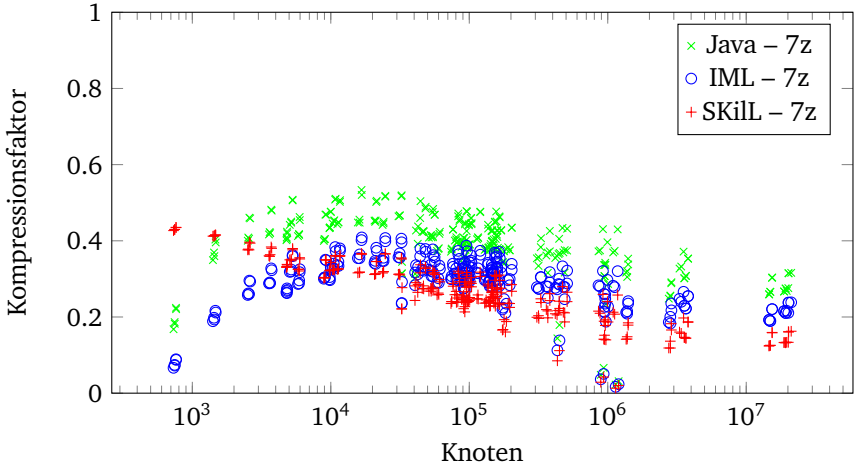



Abbildung 7.21.: Kompression nach Format als Faktor relativ zur unkomprimierten `.sf`-Datei (beste Kompression) – kleiner ist besser.

Versuch, durch stärkere Kompression die Schreibrate zu erhöhen, indem man die Schreibzeit des Hintergrundspeichers durch eine Reduktion des zu Schreibenden senkt, würde bei gegebener Hardware scheitern. Tatsächlich würde sich die Gesamtausführungszeit je nach gewählter Methode um einen ein- bis zweistelligen Faktor erhöhen und die Fähigkeit des bedarfsorientierten bzw. partiellen Auslesens würde verloren gehen. Die formatspezifischen Ausführungszeiten sind im Wesentlichen relativ zur Dateigröße verteilt. Sollte sich das Verhältnis von CPU- und IO-Geschwindigkeit so entwickeln, dass die CPU-Zeit günstiger wird, so ist diese Entscheidung zu überdenken. In diesem Fall gilt, dass man die Kompression direkt in die Datenkodierung in Form einer `@coding`-Restriction einbauen sollte, damit man das bedarfsorientierte Lesen erhält. Dasselbe gilt für Kompressionsverfahren, die schneller als die gemessenen sind. Ein solches existiert möglicherweise in Form von LZ4[1] bereits. Weil `.sf`-Dateien ohnehin sehr kompakt sind, wurde dieser Ansatz noch nicht verfolgt.

Betrachtet man die Kompressionsraten in Abb. 7.20, so überrascht zunächst die enorm starke Kompression der *bash*-Dateien mit Analyseinformationen durch `7z`, unabhängig vom zugrundeliegenden Format. Dies lässt vermuten, dass die in den entsprechenden Graphen enthaltenen Kanten keine echten Informationen enthalten. Ferner fällt auf, dass die Kompression für Java nahezu unabhängig von der Größe ist, wohingegen die Kompression für IML und SKilL mit steigender Knotenzahl besser wird. Dieses Verhalten ist insofern überraschend, als dies zur Folge hat, dass die ohnehin schon kompakteren Austauschdateien bei großen Graphen nahezu gleich stark komprimiert werden. Hierdurch bleibt die relative Ordnung der Ursprungsformate bezogen auf Dateigröße auch in komprimierter Form bestehen.

Betrachtet man die Dateigröße diverser komprimierter Formate relativ zu unkomprimiertem SKilL (siehe Abb. 7.21[2]), so ist zunächst gut zu erkennen, dass die Kodierung des Typsystems von der Graphgröße amortisiert werden muss, was zwischen 10k und 30k Knoten der Fall ist. Ferner zeigt sich, dass

[1] siehe `http://www.lz4.org`

[2] Die Ergebnisse von `zip -1` und `zip -9` wurden der Lesbarkeit halber aus der Darstellung entfernt. Diese verhalten sich sehr ähnlich.

| | IML | SKilL | Java |
|---|---|---|---|
| Knoten | 4.379*** | 2.810*** | 5.745*** |
| | (0.089) | (0.069) | (0.121) |
| Kanten | 0.081*** | 0.073*** | 0.112*** |
| | (0.018) | (0.014) | (0.025) |
| Observations | 282 | 282 | 282 |
| Adjusted $R^2$ | 0.966 | 0.953 | 0.964 |
| F Statistic (df = 2; 280) | 3993.238*** | 2862.026*** | 3741.830*** |
| *Note:* | | *p<0.1; **p<0.05; ***p<0.01 | |

Tabelle 7.24.: Regression komprimierter Dateigröße nach Knoten und Kanten mit erzwungenem Nulldurchgang.

| | IML | Java |
|---|---|---|
| SKilL | 1.503*** | 1.984*** |
| | (0.005) | (0.005) |
| Observations | 282 | 282 |
| $R^2$ | 0.997 | 0.999 |
| F Statistic (df = 1; 281) | 98898.370*** | 188185.300*** |
| *Note:* | *p<0.1; **p<0.05; ***p<0.01 | |

Tabelle 7.25.: Größe komprimierter Austauschdateien relativ zu komprimiertem SKilL.

die scheinbar gute Komprimierbarkeit von Java nur bei Graphen mit unter 200 Knoten ausreicht, um eine Datei zu erzeugen, die kleiner ist als mit äquivalent komprimiertem SKilL. Da sich klar gezeigt hat, dass `7z` von den verwendeten Kompressionsformaten über den gesamten Bereich die besten Resultate liefert, stellt sich die Frage, wie groß ein Graph in dieser Form gespeichert wäre. Um einen Vergleich mit der unkomprimierten Regression herstellen zu können, wird dasselbe Regressionsmodell wie in Tabelle 7.22 verwendet.

Die Regression in Tabelle 7.24 enthält zwei bemerkenswerte Aspekte. Wie

man schon bei Betrachtung von Abb. 7.21 erahnen konnte, hat Java nahezu[1] zu IML aufgeschlossen, der Abstand zu SKilL bleibt aber weiterhin groß. Hier macht sich die fundamental unterschiedliche Darstellung der Felddaten in SKilL bezahlt. Die von der Regression errechneten Abhängigkeitsfaktoren zwischen Dateigröße und Kanten belaufen sich für alle drei Formate auf weniger als ein Bit pro Kante. Obgleich es sich hierbei um mittlere Werte und ein Format handelt, in welchem Kanten ohnehin nicht mehr erkennbar existieren, erscheint dieser Wert unintuitiv niedrig.[2] Es bieten sich hier zwei naheliegende Erklärungen an. Zum einen ist es durchaus möglich, dass alle Graphen eine relativ geringe Entropie aufweisen, die bei der Kompression gut ausgenutzt werden kann. Zum anderen könnte die hohe Korrelation zwischen Knotenzahl und Kantenzahl das Ergebnis deutlich verschlechtern. Letzteres würde auch auf andere Regressionen dieser Art zutreffen. Nichtsdestotrotz ist für SKilL die Kompaktheit grob einen Faktor 1,5 höher als IML und grob einen Faktor 2 höher als Java (siehe Tabelle 7.25).

### 7.8.3. Ausführungszeit

Nun soll die Ausführungszeit eines Werkzeugs gemessen werden, welches die Namen aller in der Austauschdatei implementierter Funktionen auf der Standardausgabe ausgibt. Die Korrektheit ist leicht zu validieren, indem man eine Implementierung als korrekt deklariert und sicherstellt, dass alle anderen Ausgaben identisch[3] dazu sind. Aus Transitivität folgt, dass es unerheblich ist, welche Implementierung gewählt wird. Daher wird die schnellste gewählt, welche, wie sich herausstellen wird, die *SKilL/C++ lazy*-Implementierung ist. Da sich die Ausgaben nicht unterscheiden dürfen,

[1] Die Faktoren belaufen sich noch auf etwa 1,3 (Knoten) bzw. 1,4 (Kanten). Unkomprimiert liegen diese bei 2,6 bzw. 1,8.

[2] Das bedeutet nicht, dass es bei näherer Betrachtung nicht doch plausibel ist. Es erscheint genauso unintuitiv, dass die php-cgi/Steensgaard/CFG-Datei als komprimiertes SKilL lediglich 83,5 MB groß ist. In der verarbeitbaren Form als Graph im Hauptspeicher werden hier jedoch mehrere GB fällig – das Thema RAM-Nutzung wird direkt nach Betrachtung der Ausführungszeit noch genauer diskutiert werden.

[3] Weil IML manchmal Ausgaben in unterschiedlicher Reihenfolge produziert, wird die Ausgabe sortiert.

wird die korrekte Ausgabe vor der eigentlichen Messung ermittelt. Weil die Validierung nicht besonders umfangreich ist, wird hier immer jede einzelne Messung validiert.

In diesem Abschnitt werden die Laufzeiten diverser Implementierungsalternativen untersucht. Es stellt sich bei der nativen Java-Serialisierung heraus, das für große Graphen ein übergroßer Stack erforderlich ist. Daher sind die Parameter für alle Java Virtual Machine (JVM)-Implementierungen `-Xmx8G` und für Java-native Serialisierung zusätzlich `-Xss32M`. Die Messungen werden auf *Workstation* durchgeführt.

Um die Verfahren möglichst neutral miteinander zu vergleichen, werden, neben der in Ada geschriebenen IML-Implementierung und der in Java geschriebenen Java-Implementierung, SKilL-Implementierungen in Ada, Java und C++ vermessen.[1] Hierdurch ist zum einen abschätzbar, wie hoch der Unterschied innerhalb derselben Programmiersprache ausfällt und zum anderen, wie gut eine Implementierung mit den derzeitigen SKilL-Implementierungen sein könnte. In diesem Zuge wird auch untersucht, wie die Ergebnisse ausfallen würden, wenn man bedarfsorientiertes Lesen in SKilL ausnutzt. Dabei ist anzumerken, dass es sich hierbei in gewisser Weise um den Idealfall für ein bedarfsorientiertes Lesen von Felddaten handelt, weil lediglich drei Felder betroffen sind. Eine übersichtliche Darstellung der Laufzeiten findet sich in Abbildung 7.22.

**Interpretation** Im vollständig lesenden Vergleich fällt zunächst auf, dass SKilL/Ada nahezu immer bessere Resultate liefert als das vergleichbare IML. Im Vergleich von SKilL/Java mit nativem Java fällt auf, dass es einen Break-even-Punkt bei etwa 200k Knoten gibt. Hier scheint SKilL zwar deutlich besser zu skalieren, die konstanten Kosten sind aber so viel höher, dass man hier im relevanten Bereich zwischen 10k und 1M Knoten mit Java etwas bessere Resultate erzielt. Ein möglicher Erklärungsversuch hierfür sind die grob 100k Code-Zeilen, welche für die Parametrisierung der Reflection verwendet werden und deren Verwendung den JIT-Compiler und Classloader

[1] Es wird auch eine Implementierung in Scala für einen späteren Vergleich vermessen, die Daten sind hier aber nicht erheblich.

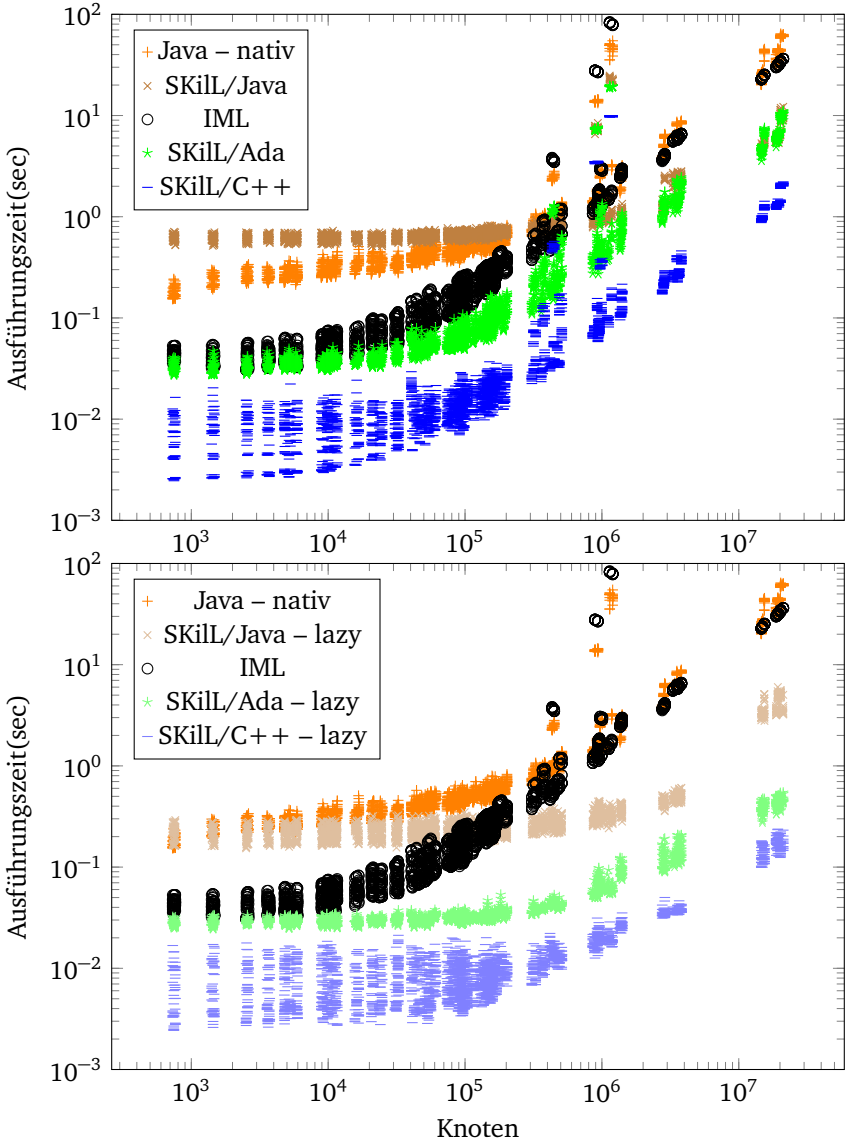



Abbildung 7.22.: Ausführungszeit von `FunctionNames` diverser Implementierungen. Oben komplett lesendes SKilL. Unten mit partiellem Lesen.

stärker beanspruchen, als das bei der nativen Java-Implementierung der Fall ist.[1] Dafür spricht, dass die konstanten Kosten in der Lazy-Implementierung erkennbar geringer ausfallen.

Diese Beobachtungen werden durch die in Tabelle 7.26 dargestellte Regressionsanalyse quantifiziert. Hier fällt zunächst auf, dass trotz statistisch signifikanter Regression, die Korrelation der Geraden vergleichsweise gering ausfällt. Ursächlich ist vermutlich das starke Ausreißen der *bash*-Derivate. Obgleich sich eine signifikantere Regression durch Entfernen dieser oder durch eine Regression in Abhängigkeit der Kantenzahl oder Dateigröße erreichen ließe, scheint dies weder zielführend noch erforderlich, da hier lediglich der visuelle Eindruck bestätigt werden soll, was auch gelingt. Es ist deutlich zu erkennen, dass sich in Ada eine etwa viermal und in Java eine etwa sechsmal bessere Skalierung der SKilL-Variante einstellt.

Wie man in der Grafik auf den ersten Blick erkennt, sind die Möglichkeiten von SKilL hier noch nicht ausgeschöpft. Besteht die Option ein Werkzeug in einer anderen Programmiersprache zu implementieren, so kann man die wesentlich bessere Performanz der C++-Implementierung nutzen. Ist es möglich das Werkzeug mit Lazy-Reads zu implementieren, so ist im unteren Teil von Abbildung 7.22 leicht zu erkennen, dass die Ausführungszeit von SKilL selbst bei derselben Implementierungssprache nahezu immer besser ist, was aufgrund der Ergebnisse der vergangenen Abschnitte aber zu erwarten war. Bevor die Frage nach dem *wie viel?* geklärt wird, sei hier noch auf zwei Implementierungsdetails hingewiesen. Die für *lazy* verwendete IR-Spezifikation enthält tatsächlich nur die verwendeten Definitionen, was sich äußerst positiv auf die Compilezeit auswirkt. Eine solche Anbindung kann via Reflection auf alle in der Austauschdatei vorhandenen Objekte vollständig zugreifen und diese auch schreiben, würde dafür aber mehr Zeit benötigen als eine IR-Spezifikation, welche alle Objekte und Felder kennt. Dabei unterscheidet sich die gewählte Repräsentation nicht entscheidend von der bekannter Felder, die als *!ondemand* markiert wurden. Die

[1] Dem könnte man begegnen, indem man eine Architektur ohne diese aufwändige Parametrisierung entwirft. Da sich hierdurch aber auch die Skalierbarkeit verschlechtern würde, ist der Mehrwert zu gering, um sich ernsthaft damit zu beschäftigen.

| | Skill/Ada | IML | Skill/Java | Java native |
|---|---|---|---|---|
| $10^6$ Knoten | 0.371*** | 1.631*** | 0.336*** | 2.255*** |
| | (0.008) | (0.033) | (0.009) | (0.021) |
| Constant | 0.249*** | 0.793*** | 0.840*** | 0.495*** |
| | (0.034) | (0.137) | (0.038) | (0.087) |
| Observations | 2820 | 2820 | 2820 | 2820 |
| $R^2$ | 0.434 | 0.471 | 0.331 | 0.810 |
| F Statistic (df = 1; 2818) | 2159.629*** | 2513.506*** | 1391.281*** | 12004.970*** |

| | Skill/C++ | Skill/Ada lazy | Skill/Java lazy | Skill/C++ lazy |
|---|---|---|---|---|
| $10^6$ Knoten | 0.076*** | 0.022*** | 0.215*** | 0.008*** |
| | (0.004) | (0.0001) | (0.001) | (0.00003) |
| Constant | 0.109*** | 0.032*** | 0.184*** | 0.008*** |
| | (0.017) | (0.0003) | (0.004) | (0.0001) |
| Observations | 2820 | 2820 | 2820 | 2820 |
| $R^2$ | 0.109 | 0.968 | 0.954 | 0.958 |
| F Statistic (df = 1; 2818) | 345.675*** | 83923.890*** | 58977.160*** | 63848.050*** |
| *Note:* | | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.26.: Regression der Ausführungszeit in Abhängigkeit der Knotenzahl – kleiner ist besser.

Implementierung von bekannten *!ondemand* Feldern ist je nach Anbindung unterschiedlich implementiert, was einen Vergleich hier schwierig macht. Daneben allokieren aktuelle SKilL-Implementierungen immer alle Knoten beim Lesen, selbst solche unbekannten Typs, um Invarianten der Implementierung zu etablieren. Die in Abbildung 7.22 klar erkennbare Korrelation bei großen Austauschdateien ist hierauf zurückzuführen[1] und nicht etwa auf eine Korrelation zwischen Graphgröße und Anzahl der verwendeten Knoten, die es möglicherweise auch gibt.

Interessant wird es, wenn man *Skill/C++ lazy* betrachtet. Hier fällt die Ausführungszeit selbst bei sehr großen Graphen soweit, dass man immer noch nahezu spontane Reaktionen erhält. Dieser Punkt ist vor allem für

[1] Beschäftigt man sich mit Profiling der Implementierungen, so ist dieser Zusammenhang unübersehbar.

eine gute Nutzererfahrung wichtig. Dass konkurrierende Produkte hier fast eine Minute für die Antwort benötigen ist bemerkenswert, selbst wenn es sich bei diesem Test um ein extremes Beispiel handelt. Vergleicht man die Steigerung der Ausführungszeiten in den Tabellen 7.26, so ergibt sich eine Verbesserung der Ausführungszeit gegenüber IML um einen Faktor von etwa 200. Vergleicht man die absoluten Ausführungszeiten von *Skill/C++ lazy* mit nativem Java, so liegt der Faktor in der Größenordnung um 280 und damit noch höher. An der Folgerung, dass SKilL leicht erkennbar die signifikant besser skalierenden Implementierungen liefert, ändert der konkrete Faktor nicht mehr viel. Es ist jedoch wiederholt darauf hinzuweisen, dass es sich hierbei um eine enorm einfache Analyse auf einem kleinen Teilgraphen und damit um die Paradedisziplin bedarfsorientierter Auswertungen handelt.

### 7.8.4. Speicherverbrauch – valgrind

In diesem Abschnitt wird der Speicherverbrauch von nativ übersetzten Implementierungen wie IML und SKilL Ada und C++, jeweils voll und partiell lesend untersucht. Hierfür wird mittels `valgrind`[NWF06] der Hauptspeicherbedarf des Gesamtprozesses ermittelt. Die auf der JVM ausgeführten Lösungen sind hierfür nicht geeignet, da die Nutzung des Hauptspeichers bei diesen vor allem durch die Kommandozeilenparameter und weniger durch das ausgeführte Programm beeinflusst wird.

`Valgrind` wird mit den Parametern `--tool=massif` und `--stacks=yes` ausgeführt. Die Stacknutzung muss mitberücksichtigt werden, da es sonst leicht möglich wäre die Werte zu schönen, indem man die Daten auf dem Stack allokiert. Gemessen wurde auf dem Rechner *Workstation*. Weil die Resultate nur marginal schwanken wurden lediglich drei Wiederholungen durchgeführt.

Eine Validierung der Ergebnisse ist hier nicht erforderlich, da es sich um dieselben Programme und Eingaben wie in Abschnitt 7.8.3 handelt, es keine sporadischen Fehler gibt und die Validierung bereits in besagtem Abschnitt stattgefunden hat.

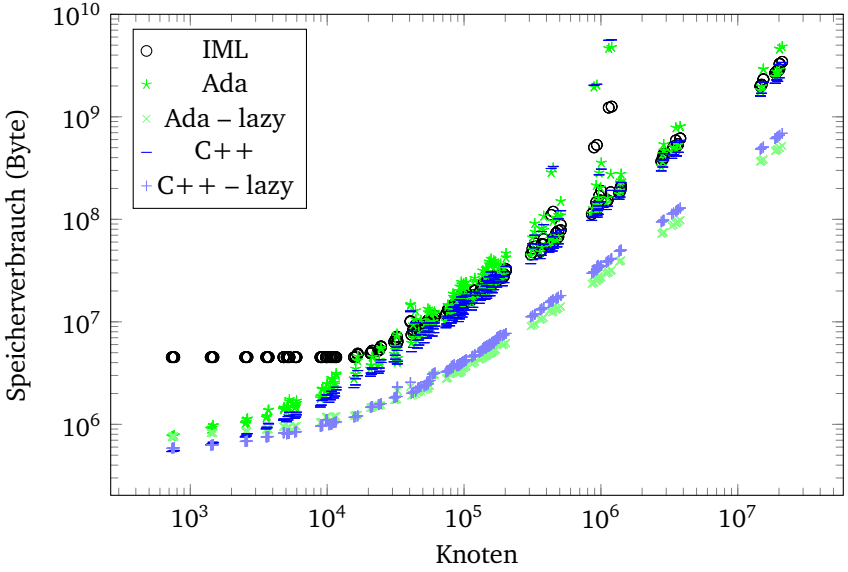



Abbildung 7.23.: RAM-Nutzung relativ zur Graphgröße.

**Interpretation** In Abbildung 7.23 ist der Speicherverbrauch in Abhängigkeit von der Knotenzahl dargestellt. Dabei fallen zunächst drei Dinge auf. Zum einen scheint IML einen relativ großen vorallokierten Puffer zu verwenden, da der Speicherverbrauch bis etwa 10k Knoten nahezu konstant ist bevor er plötzlich linear wird. Dieses Verhalten ist in den SKilL Implementierungen nicht erkennbar. Bemerkenswert ist das Verhalten für die *bash*-Ausreißer. Hier sieht man, dass die verwendeten Hash-Container einen erheblich höheren[1] Speicherverbrauch aufweisen, als es bei den von IML genutzten Containern der Fall ist. Wäre dies ein praktisches Problem, so wäre es erforderlich den Quellcode-Generator so anzupassen, dass auf speichereffizientere Container zurückgegriffen wird. Der Aufwand für die Anpassung des Generators ist dabei unerheblich. Im Allgemeinen sind der

[1] etwa viermal so viel

| | IML | SKilL/Ada |
|---|---|---|
| Knoten | 147.947*** | 163.600*** |
| | (0.889) | (3.834) |
| Constant | 11′775′251.000*** | 48′250′975.000*** |
| | (3′777′102.000) | (16′291′367.000) |
| Observations | 837 | 837 |
| $R^2$ | 0.971 | 0.686 |
| F Statistic (df = 1; 835) | 27’703.220*** | 1’820.895*** |

| | SKilL/Ada Lazy | SKilL/C++ | SKilL/C++ Lazy |
|---|---|---|---|
| Knoten | 24.729*** | 128.523*** | 32.763*** |
| | (0.009) | (4.235) | (0.010) |
| Constant | 1′429′071.000*** | 58′092′043.000*** | 1′230′797.000*** |
| | (39′199.000) | (17′994′622.000) | (40′588.800) |
| Observations | 837 | 837 | 837 |
| $R^2$ | 1.000 | 0.525 | 1.000 |
| F Statistic (df = 1; 835) | 7’186’216.000*** | 921.114*** | 11’764’880.000*** |
| *Note:* | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.27.: Regression des Hauptspeicherverbrauchs in Abhängigkeit der Knotenzahl – kleiner ist besser.

Aufwand für die Anpassung der Implementierung[1] und die Auswirkung auf die Ausführungszeit nicht einschätzbar. Ferner ist erkennbar, dass die *lazy*-Implementierungen erheblich weniger Speicher verbrauchen, obwohl sie auch alle Knoten allokieren um sich beim Referenzieren dieser ungeprüft auf deren Existenz verlassen zu können. Es ist aber zu bedenken, dass die einzelnen Knoten erheblich kleiner sind, da die fehlenden Felder hier keinen Speicher mehr erfordern. Davon abgesehen scheint es keine nennenswerten Unterschiede zu geben.

Bei der in Tabelle 7.27 dargestellten Regression fällt auf, dass die Konstanten alle höher ausfallen, als der Verbrauch auf kleinen Graphen. Diese

[1] Für diesen kleinen Test ist dieser natürlich nicht vorhanden. Für echte Werkzeuge, die die Container allenorts verwenden, ist hier eventuell eine erhebliche Anpassung des Codes erforderlich.

| | SKilL/Ada | IML | SKilL/C++ |
|---|---|---|---|
| Knoten | 167.737*** | 149.144*** | 130.203*** |
| | (6.562) | (1.503) | (3.188) |
| Constant | −55′309′517.000 | −15′487′342.000 | −13′169′822.000 |
| | (79′673′866.000) | (18′251′689.000) | (38′703′920.000) |
| Observations | 102 | 102 | 102 |
| $R^2$ | 0.867 | 0.990 | 0.943 |
| F Statistic (df = 1; 100) | 653.389*** | 9′843.490*** | 1′668.305*** |
| *Note:* | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.28.: Regression des Hauptspeicherverbrauchs in Abhängigkeit der Knotenzahl – nur große Graphen.

Diskrepanz ist bei SKilL stärker als bei IML. Betrachtet man die vergleichsweise geringen $R^2$-Werte der Ada- und C++-Implementierungen, so ist vermutlich die Regression insgesamt nicht ausreichend präzise.[1] Vermutlich müsste man, um hier direkt die Steigungen der Regressionsgeraden vergleichen zu dürfen, zunächst den Speicherverbrauch auf den *bash*-Graphen reduzieren, was das Ergebnis natürlich verändern würde.

Um dennoch eine Aussage über die Skalierbarkeit des Speicherverbrauchs treffen zu können, wird die Regression für Graphen mit über 1,2M Knoten, also Graphen größer als *bash*, wiederholt (siehe Tabelle 7.28). Bei stark gestiegenem $R^2$ ergeben sich hier zwei bemerkenswerte Ergebnisse. Zum einen ist der konstante Speicherverbrauch jetzt negativ, die Null liegt aber immer noch innerhalb der Abweichung. Zum anderen haben sich die Steigungen gegenüber der Regression über den gesamten Wertebereich nicht wesentlich verändert.

Es ergibt sich daraus, dass, bei einer gewissen Restunsicherheit, SKilL/Ada etwa einen Faktor 1.12 mehr Speicher verbraucht als IML, C++ ist dagegen bei einem Faktor von etwa 1.15 etwas effizienter als IML. Dagegen fällt der

[1] Es wären wohl noch mehr unterschiedliche Daten erforderlich, um eine bessere Abbildung des tatsächlichen Verhaltens zu erhalten. Es ist aufgrund der Konstruktion der untersuchten Implementierungen zu erwarten, dass es eine starke Korrelation geben muss, wenn man Knoten und Kanten als Variablen verwendet. Es hat sich aber bereits gezeigt, dass die Daten hierfür zu gleichförmig sind.

Verbrauch der *lazy*-Implementierungen mit einem Fünftel bis einem Sechstel von IML entscheidend geringer aus. Bedenkt man, dass Ada bei *lazy* und C++ beim vollständigen Lesen sparsamer ist, so stellt sich die Frage, ob es nicht möglich wäre, weiter Speicher einzusparen.

Aufgrund von Hauptspeicherkosten von etwa 5 bis 10 €/GB und einem maximalen Verbrauch von etwa 6 GB, muss man zu dem Schluss kommen, dass der RAM-Verbrauch zu ähnlich ist, als dass es sich lohnen würde, hier weitere Untersuchungen anzustellen. Dabei ist der hohe Verbrauch der Hashcontainer zwar negativ, diese Eigenschaft steht aber in gewisser Weise orthogonal auf der Wahl des Serialisierungsformats.

### 7.8.5. Speicherverbrauch – JVM/GC

In diesem Abschnitt wird die Hauptspeichernutzung der JVM-basierten Implementierungen untersucht. Hier zählt weniger der Bedarf des Prozesses, da dieser maßgeblich von den übergebenen Startparametern abhängt. Was hier entscheidend ist, ist im Wesentlichen die Belastung des GCs. Hierdurch wird die Performance der entsprechenden Lösung stark beeinflusst. Folglich werden drei Größen gemessen: Die maximale Größe des genutzten Heaps (in MB), die Menge des wieder eingesammelten Speichers (in MB) und die Zeit, welche vom GC verbraucht wird (in %). Die Untersuchung erfolgt dabei analog zur Messung in §7.6.2.

Eine Validierung der Ergebnisse ist nicht erforderlich, weil es sich um denselben Aufbau wie bei der Messung der Ausführungszeiten handelt.

**Interpretation** Bei den in Abbildung 7.24 dargestellten Messdaten fällt zunächst auf, dass der GC für die kleinere Hälfte der Graphen im Wesentlichen ungenutzt bleibt. Bemerkenswert in dieser Hinsicht ist, dass die *lazy* Scala-Implementierung bis etwa 2M Knoten keinen GC-Lauf erfordert. Eine Regression bietet sich für die Laufzeit erkennbar nicht an. An dieser Stelle gibt es einige, mit etwas Erfahrung leicht erkennbare, diskussionswürdige Effekte.

Zunächst sollen die drei Aspekte für die *lazy*-Implementierungen genauer

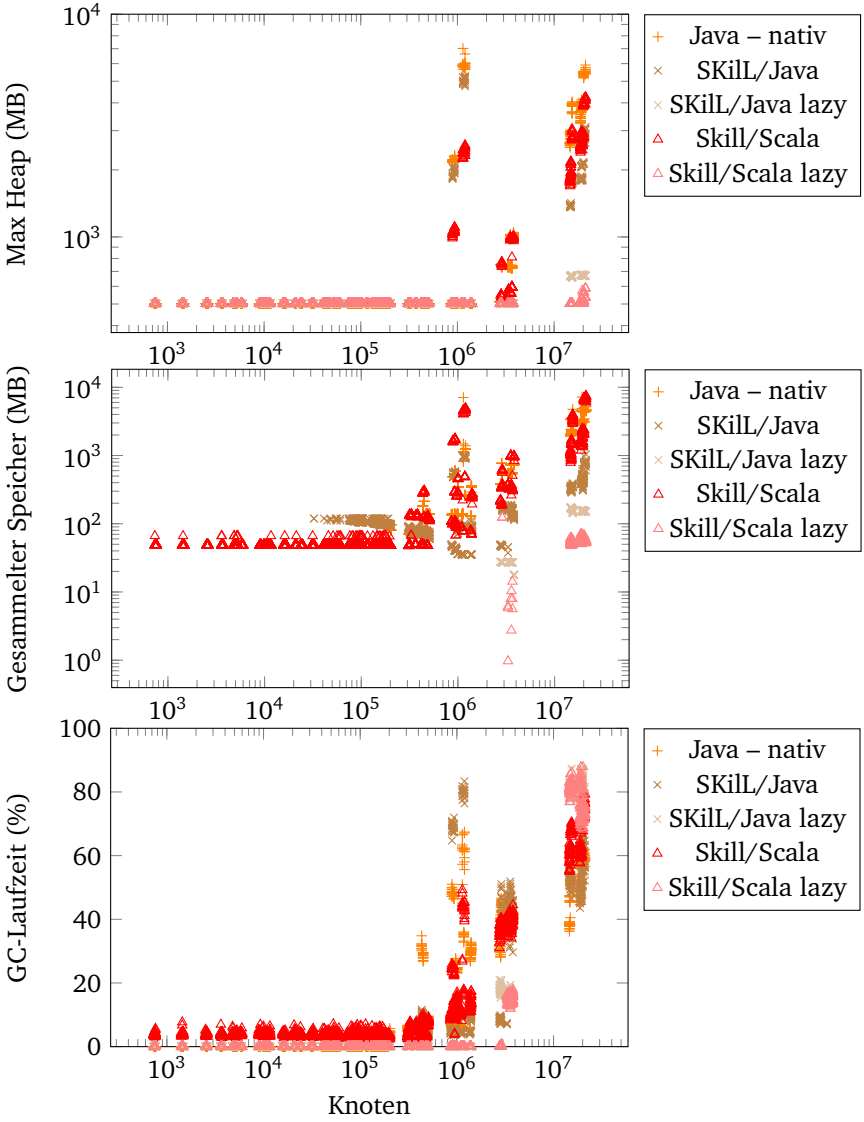



Abbildung 7.24.: Auslastung des GCs für `FunctionNames` diverser Implementierungen relativ zur Knotenzahl.

| | SKilL/Java | Java – Nativ | SKilL/Scala |
|---|---|---|---|
| $10^6$ Knoten | 81.335*** | 188.311*** | 128.085*** |
| | (1.894) | (2.318) | (0.981) |
| Constant | 516.072*** | 500.278*** | 482.019*** |
| | (8.005) | (9.796) | (4.148) |
| Observations | 2'820 | 2'820 | 2'820 |
| $R^2$ | 0.396 | 0.701 | 0.858 |
| F Statistic (df = 1; 2818) | 1'844.438*** | 6'601.069*** | 17'035.260*** |
| *Note:* | | | *p<0.1; **p<0.05; ***p<0.01 |

Tabelle 7.29.: Regression Max Heap in Abhängigkeit der Knotenzahl.

betrachtet werden. Bei der GC-Laufzeit fällt auf, dass es über weite Teile keine GC-Läufe gibt, bei sehr großen Austauschdateien dann allerdings ein enormer Teil hierfür verwendet wird. Gleichzeitig ist zu erkennen, dass sich der gesammelte Speicher im Rahmen des Speicherbedarfs der Ada- und C++-Implementierungen bewegt (vergleiche Abb. 7.23) und damit unbedenklich ist. Der hier betriebene Aufwand ist so groß, dass er auch in anderen Messungen sichtbar ist (Abb. 7.22). Ursächlich ist hier die Allokation vieler Objekte, die vom GC nicht beseitigt werden können, da sie über die Zustandsverwaltung erreichbar sind. Die enorme GC-Laufzeit entsteht im Wesentlichen dadurch, dass die neu allokierten Objekte nennenswert mehr Speicher benötigen, als alle davor existierenden Objekte zusammen. Wäre dieses Verhalten für ein Programm typisch, so könnte man es durch geschickte Wahl der Startparameter, im Wesentlichen `-Xms`, weitgehend beseitigen. Für eine Messung über eine große Bandbreite an Testdaten wie der hier verwendeten, eignet sich dieses Vorgehen jedoch kaum, da es schwierig wäre, die Vergleichbarkeit der Ergebnisse und die konkrete Wahl des Parameters zu rechtfertigen.[1]

[1] Vergleicht man die Messungen zum Speicherverbrauch mit den Max-Heap-Messungen, so ist erkennbar, dass sich tendenziell ähnliche Werte ergeben. Um diesen Vergleich zu ermöglichen ist in beiden Grafiken die Speicherachse bis 10 GB dargestellt.

Verglichen mit nativem Java, fällt SKilL/Scala durch einen geringeren Gesamtspeicherbedarf und durch eine tendenziell größere Menge an wieder eingesammeltem Speicher auf. Hierbei handelt es sich hauptsächlich um Effekte eines funktionalen Programmierstils. SKilL/Java scheint sowohl weniger Speicher zu benötigen, als auch weniger temporär benötigte Objekte zu allokieren. Bedenkt man die insgesamt geringere Laufzeit von SKilL/Java gegenüber nativem Java (§7.8.3), so ist leicht einzusehen, dass der teilweise höhere Anteil an GC-Laufzeit kein Problem darstellt.

Zudem wurde vor der Durchführung dieser Messung angenommen, dass die Pool-Struktur moderner SKilL Architekturen den GC behindert, da bekannt war, dass die Gesamtzeit der Garbage-Collection, verglichen mit anderen Programmen, relativ hoch war. Betrachtet man die Ergebnisse, so könnte auch das Gegenteil der Fall sein, da die relativen Laufzeiten vergleichsweise ähnlich sind, die absoluten Laufzeiten des Werkzeugs aber gerade bei großen Graphen deutlich geringer ausfallen (vergleiche Abb. 7.22). Hier sei jedoch bemerkt, dass nicht erkennbar ist, wie man zu diesem Thema langfristig gültige Fakten sammeln könnte, auf denen man auch in Zukunft noch relevante Erkenntnisse aufbauen könnte. Aufgrund regelmäßiger Entwicklungen in diesem Bereich empfiehlt es sich, derartige Erkenntnisse Experten dieses Bereichs zu überlassen.

Weil die *lazy*-Implementierungen für viele Testdaten keine GC-Läufe verursachen, werden diese bei der Untersuchung der Heap-Größe in Abhängigkeit der Graphgröße ausgeklammert. Eine entsprechende Regression wurde in Tabelle 7.29 durchgeführt. Es zeigt sich, dass SKilL überraschender Weise tendenziell mit kleineren Heaps zurechtkommt als natives Java. Der $R^2$ Wert für SKilL/Java ist leider zu klein, um einen direkten Vergleich der Steigung mit anderen Implementierungen vorzunehmen. Insgesamt kommt man hier zu dem Ergebnis, dass die native Java-Serialisierung neben mehr Zeit auch einen größeren Stack und einen größeren Heap benötigt.

### 7.8.6. Zusammenfassung

Insgesamt wurde in diesem Abschnitt gezeigt, dass sich das sehr kompakt konstruierte Datenformat von SKilL allenorts positiv bemerkbar macht. So fallen im direkten Vergleich mit anderen vergleichbaren Formaten die Dateigrößen bei SKilL geringer aus. Zudem lassen sich diese Austauschdateien schneller komprimieren und sind auch komprimiert noch kleiner als die Konkurrenz. Außerdem zeigt sich, dass es keine konkurrenzfähige Alternative ist, ein ineffizientes Format zu entwerfen und es durch Kompression effizient zu machen.

Weiterhin wurde demonstriert, dass SKilL im Punkt Ausführungszeit bedeutend besser skaliert als die Konkurrenz, selbst wenn man kein partielles Lesen verwendet. Verwendet man dies zusätzlich bei einem dafür geeigneten Problem, so ergeben sich dramatische Verbesserungen. Es hat sich ferner gezeigt, dass die Sprachunabhängigkeit und die hierbei ebenso entscheidende einfache Implementierbarkeit in verschiedenen Programmiersprachen dazu führt, dass man leicht auf eine Programmiersprache mit effizienterer Implementierung ausweichen kann, sollte dies erforderlich sein.

Zusätzlich konnte gezeigt werden, dass die Hauptspeichernutzung von SKilL basierten Werkzeugen mit denen konkurrierender Zwischendarstellungen vergleichbar ist. Auch hier sieht man, dass erhebliche Mengen an Ressourcen eingespart werden können, falls ein Problem durch partielles Lesen der Austauschdateien bearbeiten werden kann.

Schlussendlich darf nicht vergessen werden, dass man bei teuren Analysen durch partielles Lesen kaum Gewinne erzielen kann, da die Kosten der durchgeführten Analyse die Gesamtkosten dominieren. Dagegen sollte es bei Linearzeitanalysen in der Praxis spürbare Verbesserungen geben. Es wurde jedoch ausführlich demonstriert, dass ein Umstieg auf SKilL nicht an der Ausdrucksmächtigkeit von SKilL oder dessen Ressourcenverbrauch scheitern wird. Dabei wurden rein konstruktive Verbesserungen, wie Änderungstoleranz und Typsicherheit noch nicht berücksichtigt.

Kapitel 8

# Schlusswort

In dieser Arbeit wurde gezeigt, dass es möglich ist, basierend auf einer IR-Spezifikation eine effiziente Zwischendarstellung für Werkzeugketten zu generieren, die sprach- und plattformunabhängig ist. Dabei kann die Zwischendarstellung sowohl mit großen Werkzeugketten, und damit auch großen IR-Spezifikationen, als auch mit großen Datensätzen umgehen. Diese ist zudem fähig, auf Erweiterungen der Zwischendarstellung automatisch zu reagieren, indem kompatible Veränderungen einfach verarbeitet und inkompatible als solche erkannt werden. Ferner lässt sich die Zwischendarstellung so konstruieren, dass sich Austauschdateien partiell lesen lassen und dass neue Daten an existierende Austauschdateien angehängt werden können.

Die Betrachtung existierender Lösungen im Bereich der Zwischendarstellungen und der Serialisierung ergab, dass eine solche Lösung nicht existiert. Folglich wurde eine Lösung aus den Vorgaben konstruiert und die wichtigsten Implementierungsdetails besprochen. Es wurde demonstriert, dass die Konstruktion in allen wichtigen Punkten bessere Ergebnisse liefert als direkt anwendbare existierende Lösungen. Dabei wurden Ergebnisse teilweise auch genutzt, um Anpassungen darzustellen, die durch potentielle zukünftige Entwicklungen von Hardwarearchitekturen notwendig werden könnten.

## Zukünftige Arbeiten

Die SKilL zugrunde liegende Änderungstoleranz wird den Umgang mit zentralen IR-Spezifikationen voraussichtlich beeinflussen. Ebenso hat die Sprachunabhängigkeit in Kombination mit einem generierten API nicht direkt einschätzbare Konsequenzen, da es leicht wird, ein neues Werkzeug in einer passenden Programmiersprache zu entwickeln. Daher muss die langfristige Auswirkung der Verwendung von SKilL als Zwischendarstellung untersucht werden. Hierbei ist insbesondere zu untersuchen, ob eine Anwendung des Lavaflow-Patterns die erwarteten positiven Konsequenzen hat. Ebenso sind die praktischen Auswirkungen von bedarfsorientiertem Lesen auf die Werkzeugentwicklung zu untersuchen, welche hiermit voraussichtlich in Zusammenhang stehen. Es ist zu bedenken, dass Entwicklungsprozesse immer an die technischen Gegebenheiten angepasst werden und es sich hierbei um eine tiefgreifende Veränderung des Möglichen handeln könnte.

Es gibt in SKilL zwar Grundlagen für ein kontrolliertes Werkzeug- und Änderungsmanagement in Form eines Laufzeittypsystems, diese sind aber kaum untersucht. Der Bereich des Werkzeug- und Änderungsmanagements ist, zusammen mit weiteren Untersuchungen der Benutzbarkeit, eine Aufgabe für Softwareingenieure. Dabei sollte auch untersucht werden, inwieweit eine explizite Markierung von verwendeten bzw. berechneten Attributen pro Werkzeug eine falsche Verwendung von Attributen im Rahmen der Änderungstoleranz verhindern kann.

Eine Untersuchung der Auswirkung werkzeugspezifischer Sichten auf Benutzbarkeit, Compilezeit und Wartbarkeit in Bauhaus wäre wünschenswert. Die Grundlage hierfür wurde durch diese Arbeit gelegt. Daneben erfordert diese Untersuchung eine Anpassung der Architektur von Bauhaus. Das sprengt in Verbindung mit der Entwicklung einer von Bauhaus unabhängigen Zwischendarstellungsrepräsentation den Rahmen einer einzelnen Dissertation deutlich, da Bauhaus schlicht zu groß und derzeit nicht automatisiert getestet ist.

Die Erweiterbarkeit durch Restrictions sollte genutzt werden, um eine auf LZ4 basierende Kompression der Felddaten zu untersuchen. Ebenso wäre zu

untersuchen, ob man das bedarfsorientierte Lesen noch weiter verbessern kann, indem man beispielsweise Offsets in 1000er-Blöcken gruppiert und so die Kosten beim Zugriff auf einzelne Daten in großen Austauschdateien stark reduziert. Hierfür ist keine Anpassung des Formats erforderlich.

Die Implementierung von bedarfsorientierter Allokation und der Erkennung unveränderter ungenutzter Daten würden die Effizienz von bedarfsorientiertem Lesen weiter steigern, falls dies in der Praxis einen erkennbaren Nutzen verspricht. Ebenso ist ein Streaming von SKilL-Zuständen ein nahezu unangetastetes Thema, mit dem sich der Speicherverbrauch mancher Anwendung potentiell reduzieren ließe.

Eine Evaluation weiterer Werkzeugketten mit SKilL würde sicher zu einer größeren Verbreitung der Technologie beitragen, es sind jedoch keine grundlegend neuen Ergebnisse zu erwarten.

Im praktischen Einsatz von Append sollte man evaluieren, ob es einer Regel bedarf, die Anhängeoperationen verbietet, um sicherzustellen, dass alle relevanten Operationen, die derzeit linear in der Zahl der Blöcke sind, in der Folge konstant werden. Dabei wird man vermutlich auf eine bessere Regel als ein Maximum von etwa 10 Blöcken pro `.sf`-Datei zurückgreifen wollen. Dem zugrunde liegt die Beobachtung, dass z. B. unabhängige Typhierarchien auf verschiedene Blöcke verteilt werden können, ohne die Laufzeit beobachtbar zu beeinflussen.

Kapitel 9

# Appendix

## A. SKilL-Diff

Das in diesem Abschnitt dargestellte Werkzeug wurde zum Vergleich zweier `.sf`-Dateien eingesetzt. Die grundlegende Strategie besteht in struktureller Induktion[1] über Instanzen von Typen. Dabei wird angenommen, dass alle Knoten mit gleicher ID gleich sein sollen. D.h. für alle Knoten werden über Reflection die Felddaten verglichen. Handelt es sich dabei um Referenzen, so wird die referenzierte ID und der Typ verglichen. Handelt es sich um Hashcontainer, so wird der Inhalt in ein Array umgewandelt, sortiert und verglichen. Die Sortierung ist notwendig, da es keine definierte Reihenfolge für den Inhalt von `set` und `map` gibt und diese daher umsortiert werden können, was einen Vergleich des Bytestroms ausschließt.

Der Induktionsanfang besteht also im Abgleich der Knoten und der Induktionsschritt im Abgleich der Kanten. Dadurch wird der Abgleich der Gesamtstruktur, also insbesondere auch aller im Graph vorhandener Pfade, gewährleistet. Das Werkzeug basiert auf der IML-Spezifikation und SKilL/Scala. Falls die Ausgabe leer ist sind beide Graphen strukturell gleich. Dieses Vorgehen

[1] Siehe `https://de.wikipedia.org/wiki/Strukturelle_Induktion`.

ist in diesem Rahmen ausreichend, da es in IML keine unreferenzierten Strings gibt.

## B. etime

Das in diesem Abschnitt dargestellte Werkzeug wurde zur Laufzeitmessung verwendet. Hierbei wird mithilfe von `vfork` und `std::chrono::high_resolution_clock` versucht eine möglichst präzise Laufzeitmessung mit möglichst geringem Overhead zu produzieren. Da die Präzision in allen hier durchgeführten Messungen ausreicht um ein strikt positives Ergebnis zu produzieren, kann 0 als Fehlerwert verwendet werden, mithilfe dessen fehlerhafte Laufzeitmessungen leicht identifizierbar sind.

## C. Einfluss des unvollständigen C++-Codes auf die Messung

In diesem Abschnitt wird kurz demonstriert, dass die in §7.1.3 erwähnte fehlende Behandlung nicht zusammenhängender verteilter Felder in SKilL/C++ keinen signifikanten Einfluss auf eine Messung hat. Nach Abschluss aller übrigen Messungen wurde die Implementierung mithilfe der in §6.3 beschrieben Algorithmen vollendet.[1] Beim Test dieser ist zudem eine fehlerhafte Behandlung von unbekannten Feldern mit null-Zeigern entdeckt und behoben worden. Die Vervollständigung erforderte auf dem ausgeführten Pfad eine Verschiebung der Aufrufe der Updatefunktion für Feldrepräsentanten während der Neuorganisation der Poolstruktur im Rahmen einer Schreiboperation vor den Punkt der Vergabe der neuen IDs. Hiervon sind keine Konsequenzen für irgendeinen Ressourcenverbrauch zu erwarten. Da diese Methode dispatchend ist und es jetzt anstelle von einer drei Implementierungen gibt, ist hierdurch eventuell ein kleiner Einfluss auf die Laufzeit zu erwarten. Es wurde die Gesamtlaufzeit der `recode`-Messung aus §7.3 für alte und neue SKilL/C++-Implementierung sowie als Kontrolle für die

[1] Die Revision der reparierten Version lautet 9dd63f2a8570b46d57e9e2facef99ced8c5c18a9

| | Ada | C++ used | C++ fixed |
|---|---|---|---|
| $10^6$ Knoten | 0.795*** | 0.173*** | 0.173*** |
| | (0.014) | (0.006) | (0.006) |
| Constant | 0.426*** | 0.163*** | 0.160*** |
| | (0.060) | (0.026) | (0.026) |
| Observations | 2820 | 2820 | 2820 |
| $R^2$ | 0.528 | 0.218 | 0.218 |
| F Statistic (df = 1; 2818) | 3153.368*** | 784.946*** | 783.734*** |
| *Note:* | | *p<0.1; **p<0.05; ***p<0.01 | |

Tabelle 9.1.: Regression über Ausführungszeiten in Abhängigkeit der Graphgröße. Konstanter Anteil in Sekunden und Steigung in Sekunden pro $10^6$ Knoten.

SKilL/Ada-Implementierung auf *Workstation* durchgeführt.[1]

Die Regression der Gesamtlaufzeit gegen die Graphgröße ist in Tabelle 9.1 dargestellt. Die an sich signifikanten Regressionen lassen keinen wesentlichen Unterschied zwischen verwendeter und verbesserter SKilL/C++-Implementierung erkennen. Um ein signifikantes Ausreißen in speziellen Nutzungsszenarien zu beurteilen, sind in Abbildung 9.1 die Laufzeiten für die neu vermessenen Anbindungen in Abhängigkeit der Knotenzahl dargestellt. Eine nennenswert erhöhte Ausführungszeit der reparierten Version ist in keinem Punkt erkennbar. Tatsächlich scheint diese im Bereich der kleinen Graphen, in dem die Werte stark gestreut sind, etwas besser zu sein.

Um den optischen Eindruck mit Zahlen zu untermauern, wird für jeden vermessenen Graphen der Mittelwert der Messungen berechnet. Diese Mittelwerte können direkt miteinander verglichen werden. Um die Abhängigkeit an konkrete Ausführungszeiten zu eliminieren, wird anstelle der Differenz der Faktor verwendet. Um zu einer Darstellung zu gelangen, die Faktoren in beide Richtungen fair in einer Abbildung verteilt, wird zudem der Logarithmus genutzt, sodass für den Vergleich zweier Messungen $x$ und $y$ die

[1] Die Validierung erfolgte analog zum ursprünglichen Experiment vom 2.6. bis 3.6.17.

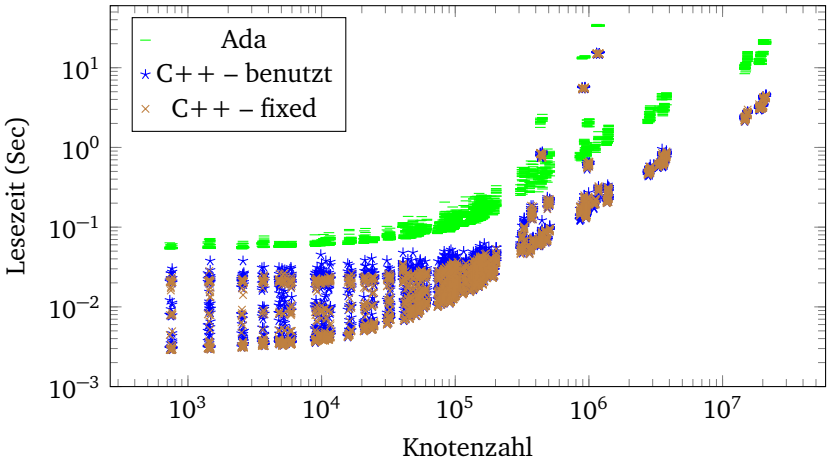



Abbildung 9.1.: Gesamtausführungszeit für `recode` (neu Messung).

Formel $log_{10}(\frac{\bar{x}}{\bar{y}})$ lautet. Hier bedeuten positive Werte eine geringere Ausführungszeit von $y$ und damit eine Verbesserung. Die Werte 0,05/0,1/0,2/0,4 entsprechen den Faktoren 1,12/1,26/1,58/2,51.

In Abbildung 9.2 sind die Verhältnisse als Boxplot dargestellt. *Fixed* bezeichnet dabei das Verhältnis zwischen original und verbesserter Implementierung. Die beiden Plots für *C++* und *Ada* dienen als Kontrolle und entstehen durch Vergleich der neuen mit der in §7.3 durchgeführten Messung. Auch hier entsteht der Eindruck, dass die Vervollständigung der Implementierung die Ausführungszeit eher leicht verbessert hat. Die in *fixed* und *C++* erkennbaren Ausreißer entstehen durch die Messung auf kleinen Graphen bei denen die Gesamtausführungszeit von externen Effekten dominiert sein kann. An den beiden Kontrollen ist erkennbar, dass eine Wiederholung die einzelnen Werte in beide Richtungen verschieben kann, was der Erwartung entspricht. Um die Frage, ob die Vervollständigung der SKilL/C++-Implementierung einen signifikanten Einfluss auf die Schlussfolgerungen im Evaluationsteil der Arbeit hätte, abschließend zu bearbeiten wurde ein gerichteter t-Test durchgeführt. Da die Laufzeit der SKilL/C++-Implementierung stets die kleinste war, reicht es zu zeigen, dass diese durch die Veränderung nicht grö-

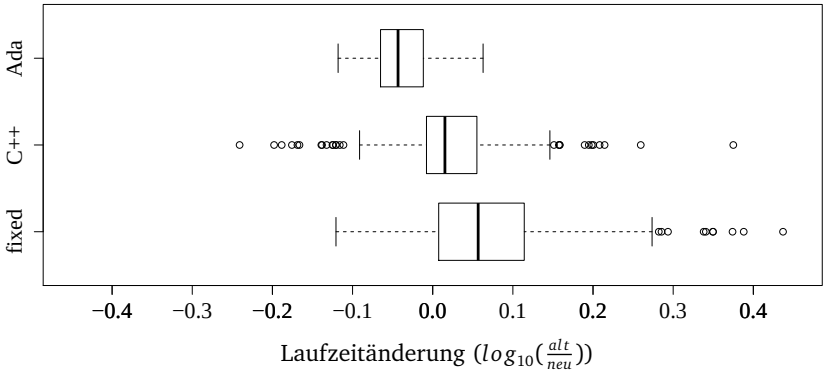



Abbildung 9.2.: Veränderung der Messung durch Ergänzung des fehlenden Codes und neu Messung. *Fixed* – original/fixed C++ neue Messung; *C++* – original C++ alte/neue Messung; *Ada* – original Ada alte/neue Messung.

ßer geworden ist. Der entsprechende Test hat laut `GNU R` einen p-Wert von 1, weswegen die Untersuchung an dieser Stelle mit der Erkenntnis beendet wird, dass keine der Schlussfolgerungen der Arbeit durch die Vervollständigung der SKilL/C++-Implementierung invalidiert wird.

## D. Betrachtungen zu EMF

In diesem Abschnitt wird der fehlgeschlagene Versuch, EMF mit SKilL zu vergleichen, erläutert. Um SKilL mit EMF [MDG+04] zu vergleichen, wurde der SKilL-Codegenerator um einen Generator für *ecore*-Spezifikationen erweitert. Die resultierende Datei kann dann mit dem EMF-Quellcode-Generator analog zu SKilL in eine Implementierung übersetzt werden. An dieser Stelle sei angemerkt, dass es bei dem in Eclipse 4.6.0 beiliegenden Generator erforderlich ist, primitive Typen in Containern durch ihre geboxten Varianten zu ersetzen. Diese Java-Eigenart wird von SKilL korrekt abgebildet und hat keinen wesentlichen Einfluss auf die nachfolgenden Betrachtungen.

Zunächst wurde ein kurzer Konverter implementiert, welcher anhand

des Laufzeittypsystems einen Graph von SKilL nach EMF konvertiert, so dass er von EMF serialisiert werden kann. Dieses Vorgehen ist analog zur Konvertierung von IML nach SKilL und hat dort zu akzeptablen Ergebnissen geführt – für die Konvertierung aller Datensätze wird hier weniger als eine Stunde benötigt. Für kleine Testdateien lieferte der `sf2emf`-Konverter in unter einer Sekunde plausible XML-Austauschdateien. Aus Interesse an der Verteilung der Laufzeit wurde ein Timer eingebaut, der die Schritte SKilL lesen, Typsystem mappen, Instanzen erzeugen, Felder setzen und EMF schreiben misst. Da zahlreiche Austauschdateien zur Verfügung standen, wurde damit begonnen, alle zu konvertieren.

Der Vorgang wurde nach zwei vollen Tagen abgebrochen, weil aufgrund der geringen Zahl konvertierter Austauschdateien nicht damit zu rechnen war, dass es weitere auf EMF basierende Experimente geben würde.[1] Am Ende des ersten Tages wurde die Laufzeit der laufenden Konvertierung von *php-cgi.iml.sf*, basierend auf den *bash43* Laufzeiten, auf etwa 8h geschätzt. Das tatsächliche Ergebnis lautet:

```
php-cgi.iml.sf;1.10;7.61E-4;12.06;2973.54;79874.28
```

Eine Laufzeit von guten 22 Stunden kann nicht mit der inhärenten Unterlegenheit einer der EMF zugrunde liegenden Technologien erklärt werden. Ein echter Vergleich bietet sich daher erst dann an, wenn der Aufruf der `save`-Methode wenigstens ein nahezu lineares Laufzeitverhalten zeigt. Es muss also in der Zustandsverwaltung von EMF einen Fehler geben, der dazu führt, dass der Serialisierungscode mindestens quadratische Laufzeit in der Größe des Graphen hat.

Bei Austauschdateien, die so klein sind, dass SKilL diese in unter 100ms lesen kann, ist das Laufzeitverhalten von EMF zunächst vergleichbar. Unter der Annahme, dass EMF eine Komplexitätsklasse schlechter als SKilL ist, verbietet sich ein Vergleich in Form von Faktoren, da dieser durch Wahl der Messdaten frei wählbar wäre.

[1] Es gab tatsächlich einen zweiten Versuch die Daten nachts und an Wochenenden zu konvertieren, dieser wurde aber ebenso abgebrochen.

Basierend auf der Auseinandersetzung mit einer Vielzahl an SKilL-Architekturen und der vorgefundenen EMF-Architektur ist selbst nach Behebung des angesprochenen Fehlers bei großen Graphen mit einer Laufzeit zu rechnen, die gegenüber SKilL einen Faktor 10 bis 100 höher ist. Die Gründe für diese Einschätzung werden im Folgenden erläutert.

In EMF werden Interfaces verwendet, um Datentypen gegenüber dem Werkzeugbauer zu beschreiben. Etwas ähnliches wurde in der Vergangenheit auch in SKilL/Java evaluiert. Bei der Entwicklung von SKilL/Java hat sich aber herausgestellt, dass es performanter ist, Klassen zu verwenden, die, falls angebracht, abstrakt sind.

Ein erheblicher Teil des in SKilL generierten Codes wird verwendet, um sonst nicht inlinebare häufige Funktionsaufrufe inlinebar zu machen. Derartigen Code findet man im EMF-Generat nicht.

In SKilL wird ein Teil der Performance durch eine Parametrisierung der generischen Zustandsverwaltung z.B. durch Allokation von Arrays korrekten Typs erzielt. Auch solchen Code sucht man vergeblich.

Die von EMF erzeugten Austauschdateien sind etwa vier- bis fünfmal so groß wie ihr SKilL-Äquivalent. Die hierdurch verursachte Verlangsamung ist sehr von der tatsächlich genutzten Hardware abhängig. Für die oben bereits erwähnte *php-cgi.iml* ergibt sich eine knapp 2 GB große XML-Austauschdatei.[1] Betrachtet man die Transferraten in Abb. 7.11, so ist im Vergleich mit Laptop/C++ alleine wegen der Dateigröße schon fast ein Faktor 10 zu erwarten – wohlgemerkt auf einem System mit einer heute eher untypisch langsamen Festplatte.

[1] Es scheint mir auch wenig hilfreich, eine 14507446 Zeilen lange Textdatei als menschenlesbar zu verkaufen, insbesondere, da es sich im Wesentlichen um eine Ansammlung von Zahlen handelt, die aus den zahlreichen Referenzen hervorgehen.

# Glossar

**`.sf`-Datei** Eine Austauschdatei in SKilL-Binärformat. 39, 83, 85, 88, 89, 93–96, 99, 100, 102–112, 129, 131, 134, 135, 140, 147–150, 152, 154, 160, 163, 164, 179, 180, 186, 188, 189, 193, 194, 196, 199, 205–207, 226, 228, 229, 247, 249, 273, 276

**Anbindung** Anbindung bezeichnet die Abstraktionsschicht zwischen Analyseimplementierung und dem Binärformat. Im Rahmen dieser Arbeit haben Anbindungen die Parameter *Programmiersprache* und *Serialisierungssystem*. Die in die Anbindung einfließende Werkzeugspezifikation ist in dieser Arbeit entweder unerheblich oder eindeutig identifizierbar. 17, 26–30, 38, 39, 42, 44, 46, 52, 59, 61, 62, 64, 65, 68, 77, 82, 83, 86, 93, 94, 101, 105–108, 112, 113, 116, 118, 120, 123, 125–129, 143, 144, 146, 147, 157, 159–162, 179, 180, 182, 183, 186, 192, 197, 210, 211, 215, 219, 234, 235, 251, 271, 272, 276

**Anwendungscode** Die Implementierung der eigentlichen Aufgabe eines Werkzeugs. Anwendungscode und Anbindung bilden zusammen das Werkzeug. 27, 38, 123, 126

**Austauschdatei** Die serialisierte Form der Zwischendarstellung. Die Austauschdatei wird von einem Werkzeug an das nächste weitergegeben. 17–20, 27, 28, 30, 32, 33, 39, 46–51, 53, 57, 62, 63, 67, 68, 72, 82, 106, 163, 170, 172, 192, 198, 203, 222, 225, 229–231, 234, 235, 242, 244, 245, 247, 254, 255, 277

**Bereinigung** Eine Abbildung eines Bezeichners in einen Namensraum, die die Zulässigkeit des Bezeichners im Zielnamensraum sicherstellt. Eine solche Abbildung sollte injektiv sein. Außerdem sollte sie die Identität auf möglichst vielen, im Zielnamensraum zulässigen, Bezeichnern sein. Unzulässige Zeichen in Bezeichnern können durch das Ersetzen aus

Sequenzen zulässiger Zeichen bereinigt werden. Der Erhalt der Länge des Bezeichners ist im Allgemeinen aufgrund der Injektivität nicht möglich. Das Konzept wird oft mit den englischen Begriffen escaping bzw. name mangling bezeichnet. 44, 53, 81, 82, 118, 125

**Quellcode-Generator** In dieser Arbeit bezeichnet Quellcode-Generator den SKilL-Compiler. Dieser liest eine SKilL-Spezifikation und generiert daraus für eine Programmiersprache Quellcode, welcher die Interaktion des Werkzeugs mit einer SKilL-basierten Zwischendarstellung kapselt. 26, 27, 29, 36, 38, 43, 61, 62, 65, 68, 76, 79, 81, 82, 107, 108, 113, 114, 116, 117, 120, 124–129, 153, 156, 208, 210, 218, 219, 237, 253

**SKilL-Kern** Der für die Anbindung eines Werkzeugs an SKilL zwingend erforderliche Teil von SKilL. 112, 114, 116, 125

**Werkzeugbauer** Der Entwickler eines Werkzeugs. Dieser legt die IR-Spezifikation fest, sorgt für die Ausführung des Quellcode-Generators und implementiert die Analyse gegen die Anbindung. 27–29, 41, 43–46, 50–52, 64, 67, 69, 71, 72, 74, 81, 82, 94, 104, 107, 109–111, 131, 132, 135, 136, 145, 146, 149, 150, 152, 153, 196, 255, 271

**Werkzeugnutzer** Der Anwender eines Werkzeugs. Dieser führt das Werkzeug aus und hat mit dessen Implementierung nichts zu tun. Ein Mensch kann jedoch gleichzeitig die Rollen des Werkzeugbauers und -nutzers einnehmen. 16, 43, 50, 170, 176, 204

## Akronyme

**API** Application Programming Interface. 17, 26, 29, 38, 44, 46, 48, 67, 71, 74–76, 80, 81, 94, 107, 108, 115, 123, 126, 127, 136, 138, 139, 146, 153, 157

**AST** Abstrakter Syntax Baum.

**BPO** Base Pool Offset. 84, 86, 87, 141, 143, 270

**EMF** Eclipse Modeling Framework. 61, 62, 221, 253–255

**GC** Garbage Collector. 193, 212, 213, 215, 240–243, 272, 273

**IML** Bauhaus Intermediate Language. 59, 60, 63, 156, 157, 162, 163, 191, 200, 202, 205, 221, 222, 225, 229, 231, 232, 236, 239, 240, 250, 254, 276

**IR** Intermediate Representation (Zwischendarstellung). 17–21, 25–28, 31, 36, 38, 39, 42–44, 48–51, 58, 63, 68, 72, 81, 84, 92, 93, 103, 104, 107, 111, 115, 116, 118, 120, 123, 125–129, 132, 156, 161, 177, 178, 186, 192–194, 198–200, 202, 207–210, 215, 216, 219, 234, 245, 257, 271, 272

**JSON** JavaScript Object Notation. 55, 61

**JVM** Java Virtual Machine. 232, 236, 240

**SKilL** Serialization Killer Language. 20, 36, 38, 45, 48, 50, 52, 54, 56–58, 60–65, 67–69, 71, 73, 74, 76, 80, 82–85, 89, 92, 95, 99, 101, 102, 104–109, 111–113, 117–120, 123, 125–128, 131, 132, 136, 144, 148, 153, 155–157, 159, 160, 162, 163, 170, 177–179, 182, 185, 190–192, 194, 198–200, 202, 203, 209, 210, 218, 221, 222, 225, 226, 229–232, 234–237, 239, 243, 244, 246, 247, 249–257, 269, 271, 272, 276

**v64** Variable length 64-bit signed integer. 99, 100, 106, 149

**VBR** Variable bitrate. 70

**XML** Extensible Markup Language. 55, 56, 61, 62, 221, 254, 255

**XSD** XML Schema Definition Language. 56, 62, 200

# Literaturverzeichnis


[And94] L. O. Andersen. „Program Analysis and Specialization for the C Programming Language“. Diss. DIKU, University of Copenhagen, Mai 1994 (Zitiert auf S. 16, 163).

[Apa13] Apache Software Foundation. *Thrift Types*. http://thrift.apache.org/docs/types, 2013 (Zitiert auf S. 60).

[Apa15] Apache Software Foundation. *xmlbeanscxx*. 4. Aug. 2015. URL: http://incubator.apache.org/projects/xmlbeanscxx.html (Zitiert auf S. 56).

[App07] Appel. *Modern Compiler Implementation in Java*. Mineola, NY, USA: Foundation Press, Inc., 2007 (Zitiert auf S. 15).

[BDF+16] J. Berberich, D. R. Degutis, N. Fateev, T. Heck, A. Hüneburg, M. Link, M. Platzer, N. Rusam, K. Singer und L. Tso. „SKilLEd“. Studienprojekt, Results will eventually be published by skill-lang on github. März 2016 (Zitiert auf S. 159, 199, 200).

[BEdN09] O. Ben-Kiki, C. Evans und I. döt Net. *YAML Ain't Markup Language*. Version 1.2. 1. Okt. 2009. URL: http://www.yaml.org/spec/1.2/spec.html (Zitiert auf S. 55).

[Ber98] H. Berghel. „The Year-2000 Problem and the New Riddle of Induction“. In: *Commun. ACM* 41.3 (März 1998), S. 13–17. URL: http://doi.acm.org/10.1145/272287.272289 (Zitiert auf S. 50).

[Blo08] J. Bloch. *Effective Java (2Nd Edition) (The Java Series)*. 2. Aufl. Upper Saddle River, NJ, USA: Prentice Hall, 2008 (Zitiert auf S. 57).

[BPS+06] *Extensible Markup Language (XML) 1.1 (Second Edition)*. W3C Recommendation. http://www.w3.org/TR/2006/REC-xml11-20060816/: W3C - World Wide Web Consortium, Sep. 2006 (Zitiert auf S. 47, 55).

[Dan13] A. Danial. *cloc*. Version 1.60. 16. Aug. 2013. URL: http://cloc.sourceforge.net (Zitiert auf S. 163).

[ECM12] ECMA International. *Common Language Infrastructure (CLI)*. ECMA-335. 6th edition. Rue du Rhône Genève, Switzerland: ECMA International, 2012 (Zitiert auf S. 58).

[ECM13] ECMA International. *The JSON Data Interchange Format*. ECMA-404 (RFC 4627). First Edition. Rue du Rhône Genève, Switzerland: ECMA International, 2013 (Zitiert auf S. 47, 55).

[Efr79] B. Efron. „Bootstrap Methods: Another Look at the Jackknife“. In: *Ann. Statist.* 7.1 (Jan. 1979), S. 1–26. URL: http://dx.doi.org/10.1214/aos/1176344552 (Zitiert auf S. 182).

[ERW08] J. Ebert, V. Riediger und A. Winter. „Graph Technology in Reverse Engineering: The TGraph Approach“. In: *10th Workshop Software Reengineering, 5-7 May 2008, Bad Honnef*. Hrsg. von R. Gimnich, U. Kaiser, J. Quante und A. Winter. Bd. 126. LNI. GI, 2008, S. 67–81. URL: http://subs.emis.de/LNI/Proceedings/Proceedings126/article2088.html (Zitiert auf S. 62, 63).

[Fel13] T. Felden. *The SKilL Language*. Techn. Ber. 2013/06. University of Stuttgart, Sep. 2013 (Zitiert auf S. 83).

[Fel14] T. Felden. „Efficient and Change-Tolerant Serialization for Program Analysis“. In: *Softwaretechnik-Trends*. Bd. 34:2. 2014 (Zitiert auf S. 178).

[Fel15] T. Felden. „Improving lava flow based software development“. In: *2nd Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP), 2015 IEEE*. März 2015, S. 9–10 (Zitiert auf S. 201).

[Fel17] T. Felden. *The SKilL Language V1.0*. Deutsch. Technischer Bericht Informatik 2017/01. Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany,

Jan. 2017, S. 64 (Zitiert auf S. 18, 19, 46, 50, 54, 68, 70, 71, 79, 80, 86, 89, 98, 101, 103–106).

[FP16] T. Felden und D. Przytarski. *SKilL on Github*. https://github.com/skill-lang/skill. 2016 (Zitiert auf S. 19, 65).

[Fro06] S. Frohn. „Konzeption und Implementierung einer Zeigeranalyse für C und C++“. Deutsch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Jan. 2006, S. 126 (Zitiert auf S. 163).

[FW16] T. Felden und M. Wittiger. „Migrating Bauhaus from IML to SKilL“. In: *Softwaretechnik-Trends*. Bd. 36:2. 2016 (Zitiert auf S. 161).

[GHJV95] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (Zitiert auf S. 108).

[GJLB00] D. Grune, C. Jacobs, K. Langendoen und H. Bal. *Modern Compiler Design*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 2000 (Zitiert auf S. 15).

[GJS+14] J. Gosling, B. Joy, G. L. Steele, G. Bracha und A. Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014 (Zitiert auf S. 52, 57, 104, 117).

[Goo13] Google. *Protocol Buffers Language Guide*. https://developers.google.com/protocol-buffers/docs/proto, 2013 (Zitiert auf S. 59, 65, 67).

[GST+08] S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, N. Mendelsohn, D. Beech und M. Maloney. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. World Wide Web Consortium, Working Draft WD-xmlschema11-1-20080620. http://www.w3.org/TR/2008/WD-xmlschema11-1-20080620, Juni 2008 (Zitiert auf S. 56).

[Har14] F. Harth. „Plattform- und sprachunabhängige Serialisierung mit SKilL“. Deutsch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Nov. 2014, S. 55. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3665&engl= (Zitiert auf S. 124).

[Har16] M. Harrer. *Prototypenentwicklung mit Bauhaus und SKilL*. Deutsch. Masterarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Masterarbeit. Okt. 2016 (Zitiert auf S. 191, 199, 207).

[Hil16] G. Hillairet. *emfjson*. Version 0.15.0. 21. März 2016. URL: http://emfjson.org (Zitiert auf S. 61).

[HJR+03] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec und A. Zivkovic. „Object Serialization Analysis and Comparison in Java and .NET“. In: *SIGPLAN Not.* 38.8 (Aug. 2003), S. 44–54. URL: http://doi.acm.org/10.1145/944579.944589 (Zitiert auf S. 221).

[HSSW00] R. Holt, A. Schürr, S. E. Sim und A. Winter. *Graph eXchange Language*. http://www.gupro.de/GXL/Introduction/background, 2000 (Zitiert auf S. 62).

[IdFC15a] R. Ierusalimschy, L. H. de Figueiredo und W. Celes. *Lua 5.2 Reference Manual*. United Kingdom: Samurai Media Limited, 2015 (Zitiert auf S. 101).

[IdFC15b] R. Ierusalimschy, L. H. de Figueiredo und W. Celes. *Lua 5.3 Reference Manual*. lua.org, 2015 (Zitiert auf S. 101).

[IEE08] IEEE. *Standard for Floating-Point Arithmetic*. Techn. Ber. 3 Park Avenue, New York, NY 10016-5997, USA: Microprocessor Standards Committee of the IEEE Computer Society, Aug. 2008, S. 1–58. URL: http://dx.doi.org/10.1109/ieeestd.2008.4610935 (Zitiert auf S. 71).

[Inf08] Info-ZIP. *Zip*. Version 3.0. 5. Juli 2008. URL: http://www.info-zip.org (Zitiert auf S. 50, 225).

[ISO11] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, Dez. 2011, 683 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853 (Zitiert auf S. 72, 131).

[ISO12] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372 (Zitiert auf S. 18, 52).

[Kai15] M. Kaistra. *SKilL vs. XML: Performance of Serialization-Concepts*. Studienarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Studienarbeit. 2015 (Zitiert auf S. 62, 199).

[LA04] C. Lattner und V. Adve. „LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation“. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society, 2004 (Zitiert auf S. 15).

[Lam87] D. A. Lamb. „IDL: Sharing Intermediate Representations“. In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), S. 297–318 (Zitiert auf S. 15, 60).

[LLV15] LLVM Project. *LLVM Bitcode File Format*. http://llvm.org/docs/BitCodeFormat.html, 2015 (Zitiert auf S. 58, 70).

[LY99] T. Lindholm und F. Yellin. *Java Virtual Machine Specification*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999 (Zitiert auf S. 58, 103, 106).

[McC76] T. J. McCabe. „A Complexity Measure“. In: *IEEE Trans. Softw. Eng.* 2.4 (Juli 1976), S. 308–320. URL: http://dx.doi.org/10.1109/TSE.1976.233837 (Zitiert auf S. 210).

[MDG+04] B. Moore, D. Dean, A. Gerber, G. Wagenknecht und P. Vanderheyden. *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. Riverton, NJ, USA: IBM Corp., 2004 (Zitiert auf S. 61, 253).

[NSL02] M. L. Noga, S. Schott und W. Löwe. „Lazy XML Processing“. In: *ACM DocEng'02*. McLean, VA: ACM Press, Nov. 2002 (Zitiert auf S. 191).

[NWF06] N. Nethercote, R. Walsh und J. Fitzhardinge. „Building Workload Characterization Tools with Valgrind“. In: *Proceedings of the 2006 IEEE International Symposium on Workload Characterization, IISWC 2006, October 25-27, 2006, San Jose, California, USA*. 2006, S. 2. URL: http://dx.doi.org/10.1109/IISWC.2006.302723 (Zitiert auf S. 236).

[NWL81] J. Nestor, W. A. Wulf und D. A. Lamb. *IDL, Interface Description Language*. Techn. Ber. Computer Science Department, Carnegie Mellon University, 1981 (Zitiert auf S. 60).

[Ora10] Oracle. *Java Object Serialization Specification*. 2010. URL: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html (Zitiert auf S. 47).

[Ora16] Oracle. *Java™ Platform, Standard Edition 8 API Specification*. 2016. URL: https://docs.oracle.com/javase/8/docs/api/index.html (Zitiert auf S. 18).

[Pav10] I. Pavlov. *7zip*. Version 9.20 (via Ubuntu xenial). 18. Nov. 2010. URL: http://www.7-zip.org (Zitiert auf S. 50, 226).

[POS16] POSIX. „Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7“. In: *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)* (Sep. 2016), S. 1–3957 (Zitiert auf S. 96, 190).

[Prz14] D. Przytarski. *Performance-Evaluation einer sprach- und plattformunabhängigen Serialisierungssprache*. Deutsch. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Bachelorarbeit. Juni 2014 (Zitiert auf S. 57, 124, 159, 178).

[Prz16] D. Przytarski. *SKilLed Bauhaus*. Deutsch. Masterarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Masterarbeit. Dez. 2016 (Zitiert auf S. 119, 127, 128, 156, 199).

[Rat17] M. Rathgeber. *SKilL Graph Visualiziation and Manipulation*. Englisch. Diplomarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Diplomarbeit. März 2017 (Zitiert auf S. 199).

[Ros15] A. Roshal. *RAR*. Version RAR 5.30 beta 2. 4. Aug. 2015. URL: http://www.rarlabs.com (Zitiert auf S. 50).

[Rot15] J. Roth. *Reduktion des Speicherverbrauchs generierter SKilL-Zustände*. Deutsch. Masterarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Masterarbeit. Mai 2015. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=MSTR-0019&engl=0 (Zitiert auf S. 63, 124, 159, 178, 191).

[RP94] J. Reynolds und J. Postel. *Assigned Numbers*. RFC 1700 (Historic). Obsoleted by RFC 3232. http://www.ietf.org/rfc/rfc1700.txt: Internet Engineering Task Force, Okt. 1994 (Zitiert auf S. 100).

[RVP06] A. Raza, G. Vogel und E. Plödereder. „Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering“. In: *Reliable Software Technologies – Ada-Europe 2006*. Bd. 4006. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 71–82 (Zitiert auf S. 15).

[Sak09] S. Sakr. „XML compression techniques: A survey and comparison“. In: *Journal of Computer and System Sciences* 75.5 (2009), S. 303–322. URL: http://www.sciencedirect.com/science/article/pii/S0022000009000142 (Zitiert auf S. 56).

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro und E. Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009 (Zitiert auf S. 67).

[SG17] R. M. Stallman und GCC Developer Community. *GNU Compiler Collection Internals*. 7.0.1. Free Software Foundation, Inc. 2017. URL: https://gcc.gnu.org/onlinedocs/gccint.pdf (Zitiert auf S. 58).

[SPT83] L. K. Schubert, M. A. Papalaskaris und J. Taugher. „Determining Type, Part, Color, and Time Relationships“. In: *Computer* 16.10 (Okt. 1983), S. 53–60. URL: http://dx.doi.org/10.1109/MC.1983.1654198 (Zitiert auf S. 100).

[Ste96] B. Steensgaard. „Points-to Analysis in Almost Linear Time“. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, S. 32–41. URL: http://doi.acm.org/10.1145/237721.237727 (Zitiert auf S. 164).

[Sun06] Sun Microsystems Inc. *JSR 222: JavaTM Architecture for XML Binding*. https://jcp.org/en/jsr/detail?id=222. 11. Mai 2006 (Zitiert auf S. 56).

[TDB+13] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, P. Leroy und E. Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*. Bd. 8339. Lecture Notes in Computer Science. Springer, 2013, S. 349. URL: http://dx.doi.org/10.1007/978-3-642-45419-6 (Zitiert auf S. 57).

[Tho15] M. Thompson. *Simple Binary Encoding*. https://github.com/real-logic/simple-binary-encoding, 2015 (Zitiert auf S. 60).

[Ung14] W. Ungur. „Nutzbarkeitsevaluation einer sprach- und plattformunabhängigen Serialisierungssprache“. Deutsch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Juli 2014, S. 66 (Zitiert auf S. 124).

[Var15] K. Varda. *Cap'n Proto*. https://capnproto.org/, 2015 (Zitiert auf S. 60).

[vHee13] D. van Heesch. *Doxygen User Manual*. http://www.stack.nl/~dimitri/doxygen/manual/, 2013 (Zitiert auf S. 199).

[Völ15] C. Völker. *Zustandsverwaltung mit SKilL*. Deutsch. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Bachelorarbeit. 2015 (Zitiert auf S. 124).

[Was90] A. I. Wasserman. „Tool Integration in Software Engineering Environments“. In: *Proceedings of the International Workshop on Environments on Software Engineering Environments*. Chinon, France: Springer-Verlag New York, Inc., 1990, S. 137–149. URL: http://dl.acm.org/citation.cfm?id=111335.111346 (Zitiert auf S. 15).

[Wei16] C. M. Weißer. *Serialization of Foreign Types with SKilL*. English. Masterarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Masterarbeit. Okt. 2016 (Zitiert auf S. 119, 127, 128).

[WM92] R. Wilhelm und D. Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer Verlag, 1992 (Zitiert auf S. 15).

[Yer03] F. Yergeau. *RFC 3629: UTF-8, a transformation format of ISO 10646*. Techn. Ber. Alis Technologies, 2003. URL: http://www.rfc-archive.org/getrfc.php?rfc=3629 (Zitiert auf S. 96).

Alle URLs wurden zuletzt am 22.06.2017 geprüft.

# Abbildungsverzeichnis

# Tabellenverzeichnis