Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Skilled LLVM

Daniel Pfister

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dr. rer. nat. Timm Felden
Beginn am:	3. November 2017
Beendet am:	3. Mai 2018

Kurzfassung

SKilL ist ein quelloffenes System für die plattform- und sprachunabhängige Serialisierung von Zwischendarstellungen. Es besteht aus einer Spezifikationssprache, einem binären Dateiformat und einem Werkzeug zur Generierung von Programmierschnittstellen. Das LLVM-Projekt ist eine quelloffene Infrastruktur für die Entwicklung von Compilern. Einen Teil dieses Projekts bilden die LLVM-Core-Bibliotheken, welche auf Basis einer Zwischendarstellung (LLVM-IR) Algorithmen zur Analyse, Optimierung und Codegenerierung bereitstellen. Da diese Bibliotheken in C++ geschrieben sind, kann es schwer sein, LLVM-IR in Programmiersprachen zu bearbeiten, für die es keine offizielle Sprachanbindung gibt, wie beispielsweise Java oder Scala. In dieser Masterarbeit wird untersucht, wie gut sich eine SKilL-basierte Darstellung für die Verwendung von LLVM-IR eignet.

Abstract

SKilL is an open source system for the platform- and language-independent serialization of intermediate representations. It consists of a specification language, a binary data format and a tool for generating language bindings. The LLVM project is an open source infrastructure for the development of compilers. The project contains the LLVM Core Libraries that provide algorithms for analysis, optimization and code generation based on an intermediate representation (LLVM IR). As these libraries are written in C++, it might be hard to use LLVM IR in programming languages, for which no official language binding exists, for instance Java or Scala. This master thesis investigates how suitable a SKilL-based representation is for using LLVM IR.

Inhaltsverzeichnis

1 Einleitung

Beim Analysieren und Erzeugen von Programmen müssen üblicherweise Daten zwischen verschiedenen Software-Werkzeugen (englisch *software tools*) ausgetauscht werden. Für diesen Datenaustausch werden sogenannte Zwischendarstellungen (IR, englisch *intermediate representation*) verwendet [Fel17a]. Das SKilL-Projekt [Fel17a] (Serialization Killer Language) ist ein quelloffenes System für die plattform- und sprachunabhängige Serialisierung von solchen Zwischendarstellungen.

Das LLVM-Projekt [LA04] ist eine quelloffene Infrastruktur für die Entwicklung von Compilern. Einen Teil dieses Projekts bilden die LLVM-Core-Bibliotheken, welche auf Basis einer Zwischendarstellung (LLVM-IR) Algorithmen zur Analyse, Optimierung und Codegenerierung bereitstellen. Da diese Bibliotheken in C++ geschrieben sind, kann es schwer sein, LLVM-IR in Programmiersprachen zu bearbeiten, für die es keine offizielle Sprachanbindung gibt, wie beispielsweise Java oder Scala. Momentan gibt es für LLVM-IR zwei verschiedene Serialisierungsformate. Zum einen gibt es das Bitcode-Format, das sich beispielsweise zum schnellen Laden durch einen JIT-Compiler eignet [LLVd]. Zum anderen gibt es das Assembly-Format, welches eine textbasierte, menschenlesbare Repräsentation ist [LLVd].

1.1 Ziele des Projekts

In dieser Arbeit wird LLVM-IR um ein zusätzliches Serialisierungsformat erweitert. Hierfür wird das SKilL-Serialisierungssystem verwendet. Der Vorteil von diesem neuen Format ist, dass es sich einfach mit verschiedenen Programmiersprachen lesen und bearbeiten lässt. Konkret sollen die folgenden Ziele erreicht werden.

- LLVM-IR muss durch die SKilL-basierte Darstellung vollständig abgedeckt werden.
- Es muss ein Konverter für die SKilL-basierte Darstellung geschrieben werden. Mit diesem sollen sich die Serialisierungsformate in beide Richtungen konvertieren lassen.
- Einige bestehende LLVM-Werkzeuge sollen an die SKilL-basierte Darstellung angebunden werden, um die Funktionsfähigkeit zu demonstrieren.
- Ferner sollen Werkzeuge geschrieben werden, die direkt auf der SKilL-basierten Darstellung arbeiten. Damit wird demonstriert, dass das Format auch mit anderen Programmiersprachen nutzbar ist.
- Die Dateigröße der verschiedenen Serialisierungsformate und die Laufzeit der Werkzeuge soll für einen repräsentativen Datensatz untersucht werden.

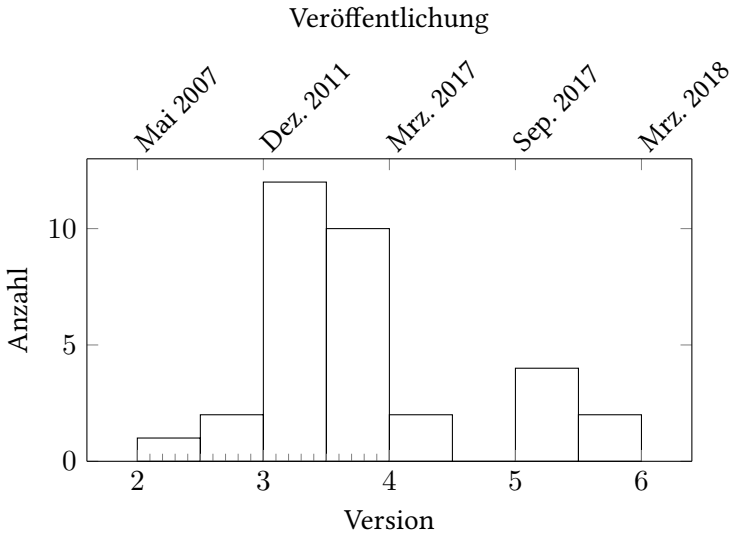


Abbildung 1.1: Histogramm der unterstützen LLVM-Versionen von 33 Sprachanbindungen auf GitHub

Das Ziel dieser Arbeit ist es nicht, einen Ersatz für die bereits bestehenden Serialisierungsformate zu entwickeln, sondern lediglich zu untersuchen, ob das SKilL-Format als sinnvolle Ergänzung genutzt werden kann.

1.2 Verwandte Arbeiten

Felden und Wittiger [FW16] haben das SKilL-Serialisierungssystem bereits verwendet, um die Zwischendarstellung IML des Projekts Bauhaus [EKP+99] zu modellieren.

Um LLVM-IR in weiteren Programmiersprachen bearbeiten zu können, gibt es auch andere Ansätze. Die wohl häufigste Vorgehensweise ist das Schreiben einer Sprachanbindung für das C-API von LLVM. Im Rahmen dieser Arbeit wurden 33 Projekte für Sprachanbindungen der LLVM-Bibliotheken auf der GitHub-Plattform betrachtet.[1] Durch diese Projekte werden insgesamt 19 verschiedene Programmiersprachen abgedeckt. Dies zeigt, dass ein starkes Interesse daran besteht, LLVM-IR mit anderen Programmiersprachen nutzen zu können.

Abbildung 1.1 zeigt das Ergebnis dieser Untersuchung. Auf der unteren horizontalen Achse ist die LLVM-Version angegeben, mit der die jeweilige Sprachanbindung zuletzt getestet wurde.[2,3] Da einige der Sprachanbindungen automatisch mit Werkzeugen wie zum Beispiel SWIG (Simplified Wrapper and Interface Generator) erzeugt werden, handelt es sich dabei aber nicht unbedingt

[1] Siehe https://github.com

[2] Es werden nur Projekte betrachtet, bei denen sich diese Information aus der Projektbeschreibung oder den Konfigurationsdateien entnehmen ließ.

[3] Ab Version 4.0 hat sich das Versionsschema geändert (siehe http://blog.llvm.org/2016/12/llvms-new-versioning-scheme.html).

um die letzte kompatible Version.[4] Auf der oberen horizontalen Achse ist das Jahr der Veröffentlichung für die jeweiligen Versionen angegeben. Man kann erkennen, dass ein großer Teil der Sprachanbindungen nicht aktiv gepflegt wird und bereits veraltet ist. Es kann davon ausgegangen werden, dass einige der Sprachanbindungen mit der aktuellen LLVM-Version nicht mehr nutzbar sind, weil das C-API über Versionssprünge hinweg keine Stabilität garantiert [LLVa].

1.3 Gliederung

Die Arbeit ist in folgender Weise gegliedert.

Kapitel 2 – Grundlagen stellt die Grundlagen dieser Arbeit vor. Es werden kurz einige Prinzipien aus dem Compilerbau zusammengefasst. Weiterhin werden die beiden Projekte SKilL und LLVM genauer beschrieben.

Kapitel 3 – LLVM-IR-Modellierung beschreibt die Modellierung von LLVM-IR mit der SKilL-Spezifikationssprache.

Kapitel 4 – Implementierung bietet eine Übersicht über den Implementierungsteil der Arbeit. Dieser beinhaltet die Werkzeuge zur Konvertierung des SKilL-Formats sowie weitere angebundene und neu geschriebene Werkzeuge.

Kapitel 5 – Evaluation untersucht das Laufzeitverhalten und den Speicherverbrauch der an das SKilL-Format angebundenen Werkzeuge. Zum Vergleich werden die beiden Serialisierungsformate des LLVM-Projekts herangezogen.

Kapitel 6 – Zusammenfassung fasst die Ergebnisse der Arbeit zusammen.

[4] Siehe http://www.swig.org

2 Grundlagen

Der Compilierprozess besteht klassischerweise mehrere Phasen (siehe Abb. 2.1). Zunächst wird im Front-End das Quellprogramm durch die lexikalische, syntaktische und semantische Analyse in einen attributierten abstrakten Syntaxbaum (AST, englisch *abstract syntax tree*) überführt. Aus dem AST wird eine Zwischendarstellung erzeugt, die sich für maschinen-unabhängige Optimierungen und Analysen eignet. Im Back-End wird aus der Zwischendarstellung der Zielcode generiert. [ALSU06]

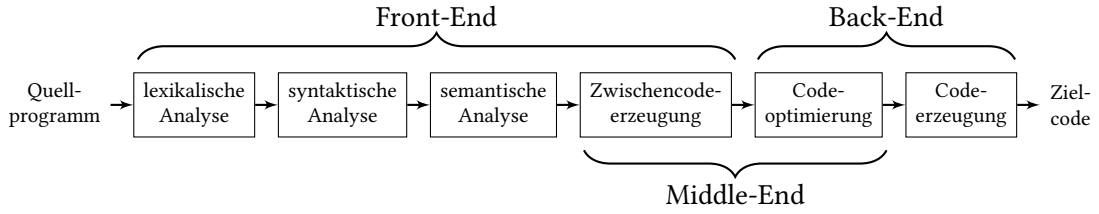


Abbildung 2.1: Modell eines Compilers [ALSU06; Plö16]

Die LLVM-Core-Bibliotheken enthalten unter anderem die Funktionalität für das Middle- und Back-End eines Compilers.[1] Die Bibliotheken arbeiten auf einer gemeinsamen Zwischendarstellung namens LLVM-IR, die sich als Bitcode-Format (.bc-Datei) oder Assembly-Format (.ll-Datei) serialisieren lässt. Weiterhin enthält das LLVM-Projekt den Compiler `clang`, ein Front-End für C, C++ und Objective-C.[2]

2.1 LLVM-IR

Im folgenden Abschnitt wird ein Überblick über die Struktur der LLVM-IR-Sprache gegeben. Zur Veranschaulichung anhand von Beispielen wird das Assembly-Format von LLVM-IR verwendet. Für weitere Details wird auf das LLVM Language Reference Manual [LLVd] verwiesen.

2.1.1 Modul

Programme bestehen in LLVM-IR aus Modulen, welche jeweils einer Übersetzungseinheit aus dem Eingabeprogramm entsprechen [LLVd]. Anhand des folgenden Programms für die iterative

[1] Siehe https://llvm.org
[2] Siehe https://clang.llvm.org

Berechnung der Fibonacci-Folge (siehe Listing 2.1 und 2.2) soll der Aufbau eines LLVM-IR-Moduls veranschaulicht werden.

Auf oberster Ebene besteht ein Modul aus einer Auflistung von globalen Werten, wie zum Beispiel globalen Variablen und Funktionen. Durch das Zeichen „`;`“ werden Zeilenkommentare begonnen, welche nur im Assembly-Format von LLVM-IR existieren. [LLVd]

Listing 2.1 Fibonacci-Folge in C

```
1  int c = 42;
2
3  int fib(int n) {
4    int a = 1, b = 1, c;
5    for (int i = 2; i < n; ++i) {
6      c = a + b;
7      a = b;
8      b = c;
9    }
10   return b;
11 }
12
13 int main() {
14   return fib(c) != 267914296;
15 }
```

Listing 2.2 Fibonacci-Folge in LLVM-IR

```
1  @c = global i32 42, align 4
2
3  define i32 @fib(i32) {
4    %2 = icmp sgt i32 %0, 2
5    br i1 %2, label %3, label %4
6
7  ; <label>:3
8    br label %6
9
10 ; <label>:4
11   %5 = phi i32 [ 1, %1 ], [ %10, %6 ]
12   ret i32 %5
13
14 ; <label>:6
15   %7 = phi i32 [ %11, %6 ], [ 2, %3 ]
16   %8 = phi i32 [ %9, %6 ], [ 1, %3 ]
17   %9 = phi i32 [ %10, %6 ], [ 1, %3 ]
18   %10 = add nsw i32 %8, %9
19   %11 = add nuw nsw i32 %7, 1
20   %12 = icmp eq i32 %11, %0
21   br i1 %12, label %4, label %6
22 }
23
24 define i32 @main() {
25   %1 = load i32, i32* @c, align 4
26   %2 = call i32 @fib(i32 %1)
27   %3 = icmp ne i32 %2, 267914296
28   %4 = zext i1 %3 to i32
29   ret i32 %4
30 }
```

2.1.2 Bezeichner

Im Assembly-Format haben Bezeichner von globalen Werten das Präfix „`@`“. Bezeichner von lokalen Werten (Register und Grundblöcke) und Typen haben das Präfix „`%`“. Bezeichner von Metadaten haben das Präfix „`!`“. Entitäten ohne Namen werden durch einen Bezeichner identifiziert, der aus dem entsprechenden Präfix und einer eindeutigen Nummer besteht. [LLVd]

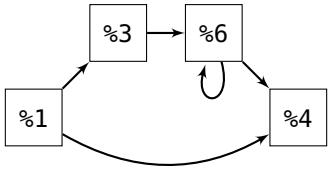


Abbildung 2.2: Der Kontrollflussgraph der Funktion `fib`. Die Grundblöcke werden durch ihren temporären Namen identifiziert, die sich aus den Kommentaren entnehmen lassen. Der erste Grundblock hat hier den Bezeichner `%1`.

2.1.3 Typsystem

LLVM-IR ist eine streng typisierte Sprache, im Sinne, dass es keine impliziten Typumwandlungen gibt [LA04]. Dies lässt sich in Zeile 28 von Listing 2.2 sehen, wo der boolesche Wert `%3` vor der Rückgabe explizit in einen 32-Bit-Integer konvertiert werden muss.

Es gibt Grundtypen für Integer mit beliebiger Bitweite und Gleitkommazahlen in verschiedenen Formaten.[3] Ferner lassen sich abgeleitete Typen für Funktionen, Zeiger, Arrays, Vektoren und Records definieren. Für spezielle Werte existieren noch weitere Typen wie zum Beispiel `void` oder `label`. [LLVd]

Typen sind äquivalent, falls sie die gleiche Struktur haben. Die einzige Ausnahme hiervon sind Record-Typen mit einem Namen. [LLVd]

2.1.4 Instruktionssatz

LLVM-IR verwendet einen RISC-artigen (englisch *reduced instruction set computer*) Instruktionssatz mit einer unbegrenzten Anzahl von typisierten Registern. Es gibt Instruktionen mit verschiedener Anzahl von Operanden. Die meisten Instruktionen sind Drei-Adress-Anweisungen, haben also zwei Operanden und ein Resultat. Dazu gehören unter anderem arithmetische und logische Operationen. [LA04]

Bestimmte Instruktionen haben Überladungen für verschiedene Typen. Beispielsweise kann die Instruktion `add` für beliebige Integer oder Vektoren von Integern verwendet werden, solange beide Operanden den gleichen Typ haben. [LLVd]

2.1.5 Kontrollflussgraph

Ein Grundblock (englisch *basic block*) ist eine Folge von Instruktionen mit sequentiellem Kontrollfluss und einem Eingang [ALSU06]. Jede Instruktion in LLVM-IR befindet sich in einem Grundblock und jeder Grundblock endet mit einer Terminierungsinstruktion (englisch *terminator instruction*). Ein Kontrollflussgraph (CFG, englisch *control flow graph*) ist ein gerichteter Graph, dessen Knoten Grundblöcke sind und der einen designierten Knoten als Einstiegspunkt hat. Die

[3] Die obere Schranke für die Integer-Bitweite ist $2^{23} - 1$, was für alle sinnvollen Anwendungen ausreichen sollte.

Kanten innerhalb des Graphen folgen aus den Sprungzielen der letzten Instruktion eines jeden Grundblocks.

Eine Funktionsdefinition in LLVM-IR enthält eine Liste von Grundblöcken, die wiederum einen CFG bilden. Insbesondere gibt es in LLVM-IR also keine Kontrollstrukturen, wie sie in höheren Programmiersprachen vorkommen. Abbildung 2.2 zeigt den CFG für die Funktion `fib` aus Listing 2.2.

Im CFG dominiert ein Knoten X einen Knoten Y, sofern X auf jedem Pfad vom Einstiegspunkt zu Y liegt [CFR+89]. Im Abbildung 2.2 wird beispielsweise Knoten `%6` durch Knoten `%3` dominiert.

2.1.6 SSA-Form

Ein Programm befindet sich in SSA-Form [CFR+89] (englisch *static single assignment form*), sofern jede Variable genau eine statische Definition hat und jede Verwendung (englisch *use*) einer Variable von deren Definition dominiert wird [LA04].[4,5] Beim Transformieren eines Programms in SSA-Form kann es zu Konflikten kommen, falls die Verwendung einer Variable von mehreren Definitionen dieser Variable erreicht werden kann. Daher werden sogenannte ϕ-Knoten zum Auflösen dieser Konflikte verwendet. Ein ϕ-Knoten hat dabei die Semantik einer bedingten Zuweisung in Abhängigkeit vom vorangegangenen Grundblock [CFR+89].

In LLVM-IR befinden sich die Register in SSA-Form, nicht jedoch Werte, die im Speicher liegen [LA04]. Die ϕ-Knoten werden explizit durch `phi`-Instruktionen dargestellt, die immer am Anfang eines Grundblocks stehen [LLVd]. In Listing 2.2 kann man diese Instruktionen in den Zeilen 11, 15, 16 und 17 sehen.

Konstanten, Grundblöcke und weitere Konstrukte werden in LLVM-IR ebenfalls als SSA-Werte betrachtet. Zu den Konstanten zählen beispielsweise globale Variablen, deren Wert ein konstanter Zeiger auf einen Speicherbereich ist. [LLVd]

2.1.7 Wohlgeformtheit

Es gibt eine Reihe von Kriterien, die ein Modul erfüllen muss, damit es wohlgeformt ist. Ein syntaktisch korrektes Programm ist nicht unbedingt wohlgeformt [LLVd]. Die LLVM-Bibliotheken bieten die Funktionalität, um diese Kriterien zu überprüfen.

Die korrekte Dominanzrelation der SSA-Form ist eines dieser Kriterien. Sofern Code in einem unerreichbaren Grundblock vorkommt, werden die Dominanzeigenschaften ignoriert.[6] Alle Verwendungen von Definitionen aus anderen Grundblöcken würden sonst dazu führen, dass das Modul nicht mehr wohlgeformt wäre.

[4] Der Begriff „Definition“ bezieht sich bei Datenflussanalysen auf Zuweisungsstellen einer Variable.

[5] Die Dominanzrelation auf Definitionen und Verwendungen lässt sich aus der Dominanzrelation auf CFG-Knoten herleiten. Dabei ist zu beachten, dass eine Definition ihre eigenen Verwendungen nicht dominiert.

[6] Siehe https://bugs.llvm.org/show_bug.cgi?id=361

Das Beispiel aus Listing 2.3 zeigt einen solchen Fall. Der Grundblock `unreachable_bb` wird von keinem anderen Grundblock im CFG dominiert, weil dieser eine eigene Zusammenhangskomponente bildet. Somit dominiert die Definition von Register `%1` nicht seine Verwendung in Zeile 7. Eine Folge dieser Behandlung ist, dass auch die Dominanzrelation innerhalb eines Grundblocks nicht geprüft wird. Offensichtlich dominiert die Definition von Register `%3` nicht seine Verwendung in Zeile 6.

Listing 2.4 zeigt ein weiteres Beispiel mit unerreichbaren Grundblöcken. Hier gibt es auch ohne ϕ-Knoten zyklische Abhängigkeiten zwischen Instruktionen. Die Definitionen der Register `%1` und `%2` dominieren beide nicht ihre jeweiligen Verwendungen.

Listing 2.3 Verwendung dominiert Definition

```
1 define i32 @foo() {
2   %1 = alloca i32
3   store i32 42, i32* %1
4   ret i32 0
5 unreachable_bb:
6   %2 = add i32 %3, %3
7   %3 = load i32, i32* %1
8   ret i32 %2
9 }
```

Listing 2.4 Zyklische Abhängigkeit

```
1 define i32 @foo() {
2   ret i32 0
3 unreachable_bb:
4   %1 = add i32 %2, %2
5   br label %unreachable_bb2
6 unreachable_bb2:
7   %2 = add i32 %1, %1
8   br label %unreachable_bb
9 }
```

2.1.8 Metadaten

Metadaten können in LLVM-IR verwendet werden, um zusätzliche Informationen über das Programm zu vermitteln. Eine Anwendung hiervon sind Debug-Informationen [LLVf]. Metadaten werden anhand ihrer Struktur unifiziert und bilden einen gerichteten Graphen.

Listing 2.5 zeigt ein Beispiel, das in LLVM-IR nicht möglich ist. Die beiden Metadatenknoten `!0` und `!1` haben die gleiche Struktur. Daher müssen sie zu einem einzigen Knoten vereinigt werden. Der Metadatenknoten `!2` ist als `distinct` gekennzeichnet. Das führt dazu, dass dieser von der Unifizierung ausgeschlossen wird.

In Listing 2.6 ist ein Beispiel für einen zyklischen Metadatengraphen gegeben. Metadatenknoten, die sich direkt selbst referenzieren, müssen `distinct` sein.

Listing 2.5 Unmögliche Metadaten

```
1 !foo = !{!0, !1, !2}
2 !0 = !{i32 42}
3 !1 = !{i32 42}
4 !2 = distinct !{i32 42}
```

Listing 2.6 Zyklische Metadaten

```
1 !foo = !{!0, !1, !2}
2 !0 = !{!1, !2}
3 !1 = distinct !{!1}
4 !2 = !{!0}
```

2.1.9 Attribute

Attribute können verwendet werden, um zusätzliche Eigenschaften von Funktionen, Parametern, globalen Variablen sowie Funktionsaufrufen festzulegen. Ein Beispiel für ein Attribut wäre

Typ	Beschreibung
`bool`	Boolescher Wert
`i8`	8-Bit-Integer
`i16`	16-Bit-Integer
`i32`	32-Bit-Integer
`i64`	64-Bit-Integer
`v64`	64-Bit-VBR-Integer (Variable Bitrate)
`f32`	IEEE-754-Gleitkommazahl mit einfacher Genauigkeit
`f64`	IEEE-754-Gleitkommazahl mit doppelter Genauigkeit
`string`	UTF-8-kodierter String
`annotation`	Zeiger auf beliebigen benutzerdefinierten Typ
`T[]`	Array von Typ `T`
`list<T>`	Liste von Typ `T`
`set<T>`	Set von Typ `T`
`map<T>`	Map von Typ `T`

Tabelle 2.1: Typen in der SKilL-Spezifikationssprache (Grundtypen oben, abgeleitete Typen unten)

`noreturn` für Funktionen, durch das festgelegt wird, dass die Funktion niemals aus ihrem Aufruf zurückkehrt [LLVf]. In Listing 2.2 gibt es keine Attribute.

2.2 SKilL

Das SKilL-Serialisierungssystem besteht aus einer Spezifikationssprache, einem binären Dateiformat (.sf-Datei) und einem Werkzeug zur Generierung von Programmierschnittstellen. Die Funktionsweise des binären Dateiformats ist für diese Arbeit nicht relevant. Die Spezifikationssprache wird im Folgenden kurz beschrieben. Für weitere Details wird auf den technischen Bericht über SKilL [Fel17b] und die Dissertation von Felden [Fel17a] verwiesen.

2.2.1 Spezifikationssprache

Auf oberster Ebene besteht eine SKilL-Spezifikation aus einer Sammlung von Typdefinitionen und Interfaces. Benutzerdefinierte Typen (siehe [Fel17b], Abschnitt 4.3) enthalten eine Anzahl von benannten Feldern mit beliebigem Typ.

Die Grundtypen (siehe [Fel17b], Abschnitt 4.1) und abgeleiteten Typen (siehe [Fel17b], Abschnitt 4.2) sind in Tabelle 2.1 dargestellt. Dabei sind alle Integertypen vorzeichenbehaftet.

Benutzerdefinierte Typen können durch Einfachvererbung eine Hierarchie bilden. Wie in objektorientierten Sprachen üblich, werden dabei die definierten Felder vererbt. [Fel17b]

Ein benutzerdefinierter Typ kann beliebig viele Interfaces (siehe [Fel17b], Abschnitt 4.4.1) implementieren. Ein Interface gibt dabei die Garantie, dass ein Typ ein bestimmtes Feld enthält.

Auf Felder und Typen lassen sich Restriktionen (siehe [Fel17b], Abschnitt 5.1) anwenden, die beim Serialisieren und Deserialisieren geprüft werden. Ein Beispiel hierfür wäre die Einschränkung des Wertebereichs eines Integers mit der Restriktion `range`. Ferner gibt es die Möglichkeit dem API-Generator bestimmte Hinweise (siehe [Fel17b], Abschnitt 5.2) zu geben. Restriktionen haben das Präfix „`@`“ und Hinweise das Präfix „`!`“.

Felder lassen sich außerdem mit dem Schlüsselwort `auto` (siehe [Fel17b], Abschnitt 3.4.2) markieren, damit sie nicht serialisiert werden. Solche Felder können von Algorithmen durch das SKilL-API genutzt werden, um in den Instanzen weitere Daten zu hinterlegen. Auf gleiche Weise können `custom`-Felder (siehe [Fel17b], Abschnitt 4.4.5) benutzt werden. Diese werden nur für das API einer bestimmten Programmiersprache generiert. Damit kann das Feld dann auch einen beliebigen Typ aus der Programmiersprache haben.

Die Spezifikationssprache ähnelt syntaktisch der C++-Programmiersprache. Wie bei XML-Schema wird hier aber nur die Struktur von Daten beschrieben. Wie in Java haben die Felder mit benutzerdefinierten Typen eine Referenzsemantik. Daher lassen sich die mit SKilL spezifizierten Datenstrukturen als gerichtete Graphen interpretieren. Als Knoten werden alle Instanzen benutzerdefinierter Typen und Strings betrachtet.[7] Zu Kanten werden alle Felder gezählt, die keine Nullzeiger enthalten. Falls das Feld ein Array oder eine Liste ist, wird stattdessen jedes Element davon als Kante gezählt.

In Listing 2.7 ist eine einfache SKilL-Spezifikation dargestellt. Der Typ `A` erbt dabei vom Typ `B` und implementiert das Interface `C`. Typ `A` enthält also die drei Felder `x`, `y` und `z`.

Listing 2.7 Beispiel für eine SKilL-Spezifikation

```
1  A : B : C {
2    f32 z;
3  }
4
5  B {
6    @range(0,42)
7    i32 x;
8  }
9
10 interface C {
11   string y;
12 }
```

2.2.2 API-Generator

Sobald die Zwischendarstellung spezifiziert ist, lassen sich mit dem `skill`-Werkzeug Programmierschnittstellen generieren. Zum aktuellen Zeitpunkt gibt es Generatoren für sechs verschiedene Programmiersprachen. Diese Programmiersprachen sind Ada 2012 [Prz14], C99 [Har14], C++11, Haskell [Har16], Java 8 [Ung14] und Scala 2.12.[8] Weiterhin wird in einer noch unveröffentlichten Masterarbeit ein Generator für die Programmiersprache Rust entwickelt.

[7] Hierzu werden auch die Namen von Typen und Feldern gezählt.
[8] Siehe https://github.com/skill-lang/skill

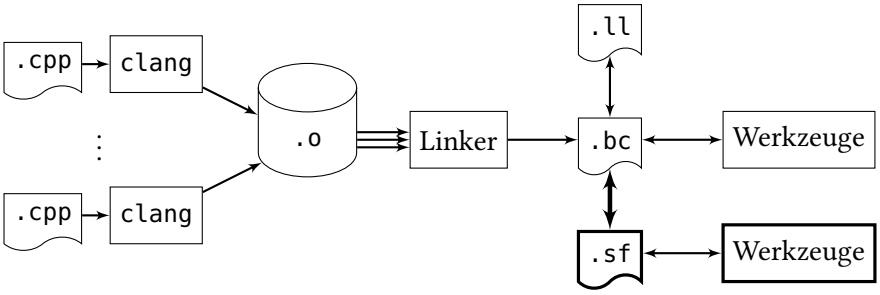


Abbildung 2.3: Prozess um ein komplettes C++-Programm zu analysieren

2.3 Werkzeugketten

Eine Werkzeugkette (englisch *toolchain*) ist eine Sammlung von Werkzeugen, in der die Ergebnisse mancher Werkzeuge von anderen Werkzeugen weiterverarbeitet werden. Dabei kommunizieren die Werkzeuge über eine gemeinsame Zwischendarstellung. Als Vorteil dieser Vorgehensweise lassen sich zum Beispiel Werkzeuge in unterschiedlichen Programmiersprachen schreiben oder Zwischenergebnisse von zeitaufwändigen Analysen speichern. [Fel17a]

Ein interessanter Anwendungsfall für die SKilL-basierte Darstellung von LLVM-IR ist eine Programmanalyse auf einem kompletten Programm. Der Prozess hierfür ist in Abbildung 2.3 skizziert. Die fehlenden Elemente, die in dieser Arbeit entwickelt werden, sind dabei fett markiert.

2.3.1 Zwischencodeerzeugung

Bevor Werkzeuge auf der Zwischendarstellung arbeiten können, muss diese zunächst einmal aus dem Quellprogramm generiert werden. In dieser Arbeit werden nur Quellprogramme betrachtet, die in C oder C++ geschrieben sind, sodass sich diese mit `clang` übersetzen lassen. Falls das Quellprogramm aus nur einer Übersetzungseinheit besteht, lässt sich LLVM-Bitcode erzeugen, indem `clang` mit den Optionen `-c -emit-llvm` aufgerufen wird. Für komplexere Programme gibt es jedoch zahlreiche Übersetzungseinheiten. Der Erstellungsprozess wird dabei meistens durch ein Buildsystem wie CMake oder das GNU Build System gesteuert.

Eine Methode um auch große Projekte nach LLVM-IR zu übersetzen ist die Linkzeitoptimierung. Durch die Option `-flto` generiert `clang` bei der Übersetzung weiterhin Dateien mit der Endung `.o`, allerdings enthalten diese nun LLVM-Bitcode anstatt einer ELF-Datei. Damit das Linken trotzdem funktioniert, muss der `lld`-Linker aus dem LLVM-Projekt verwendet werden.[9] Mit den Optionen `-fuse-ld=lld -Wl,-save-temps` wird `clang` dazu angewiesen den `lld`-Linker zu verwenden. Außerdem werden alle Zwischenschritte serialisiert, sodass auch der Bitcode vor der Linkzeitoptimierung vorhanden ist. Bei diesem Vorgehen entspricht der Linker aus Abbildung 2.3 dem `lld`-Linker.

Eine weitere Methode bietet das WLLVM-Projekt [LLM+18] (Whole Program LLVM), welches ein Wrapper für `clang` bereitstellt. Hierbei werden für jede Übersetzungseinheit eine Objektdatei im

[9] Siehe https://lld.llvm.org

ELF-Format und eine Bitcode-Datei erzeugt. In einer speziellen Sektion der ELF-Datei wird der Pfad der Bitcode-Datei hinterlegt. Beim Linken der Objektdateien werden diese Sektionen dann konkateniert. Aus der finalen ELF-Datei werden diese Pfade ausgelesen und mit dem Werkzeug `llvm-link` zu einer einzelnen Bitcode-Datei gelinkt [LLM+18]. Bei diesem Vorgehen entspricht der Linker aus Abbildung 2.3 dem Standard-Linker des Systems mit einem anschließenden Aufruf des Werkzeugs `extract-bc`. Die zusätzlichen Bitcode-Dateien sind dabei nicht dargestellt.

In dieser Arbeit wird WLLVM benutzt, da es sich ohne zusätzlichen Aufwand mit beliebigen Buildsystemen verwenden lässt. Für die hier verwendeten Projekte lassen sich aber prinzipiell beide Methoden anwenden.

2.3.2 Werkzeuge

Im LLVM-Projekt werden Analysen und Transformationen üblicherweise nicht in Werkzeuge gekapselt, sondern als Pässe implementiert. Es gibt verschiedene Arten von Pässen, die jeweils auf unterschiedlichen Teilen des Programms arbeiten. Als Beispiel seien Pässe auf Funktionen und Pässe auf ganzen Modulen genannt. In den Pässen lassen sich Abhängigkeiten definieren, die der `PassManager` nutzt, um die korrekte Ausführungsreihenfolge zu bestimmen. [LLVh]

Bei einer SKilL-basierten Darstellung lässt sich die Zwischendarstellung aufgrund der Änderungstoleranz (siehe [Fel17a], Abschnitt 2.2) anpassen. So kann ein Werkzeug zum Beispiel neue Typen definieren oder vorhandene Typen um neue Felder erweitern. Werkzeuge können also die Spezifikation anpassen, um ihre Ergebnisse weiterzugeben. Die einzelnen Werkzeuge müssen dabei nicht die komplette Spezifikation kennen, sondern nur den Teil, den sie tatsächlich benötigen.

3 LLVM-IR-Modellierung

Einer der zentralen Punkte dieser Arbeit ist die Modellierung von LLVM-IR mit einer SKilL-Spezifikation. Ausgangspunkt für diese Modellierung sind die C++-Klassen der LLVM-Bibliotheken, die im Folgenden als In-Memory-Format bezeichnet werden. Für die restliche Arbeit wird die auf dieser SKilL-Spezifikation basierende LLVM-IR-Darstellung als SKilL-Format bezeichnet. Die SKilL-Spezifikation hat 1759 Zeilen und enthält dabei 212 Klassen mit insgesamt 230 Feldern. In diesem Kapitel wird auf die Unterschiede zwischen dem In-Memory-Format und dem SKilL-Format eingegangen.

3.1 In-Memory-Format

Das In-Memory-Format stellt LLVM-IR durch C++-Klassen dar. Ein großer Teil der Klassen befindet sich in den drei Klassenhierarchien für Typen, SSA-Werte und Metadaten. Im weiteren Verlauf dieses Kapitels werden einige dieser Klassen näher beschrieben. In Listing 3.1 ist der C++-Code aufgeführt, der das Beispiel aus Listing 3.2 im In-Memory-Format erzeugt.

Listing 3.1 C++-Codeausschnitt zum Erzeugen von Listing 3.2

```
1  LLVMContext ctx;
2  Module mod("", ctx);
3  IRBuilder<> builder(ctx);
4  std::vector<Type*> argTypes(2, Type::getFloatTy(ctx));
5  FunctionType* fnType = FunctionType::get(Type::getFloatTy(ctx), argTypes, false);
6  Function* fn = Function::Create(fnType, Function::ExternalLinkage, "foo", &mod);
7  BasicBlock* bb = BasicBlock::Create(ctx, "", fn);
8  builder.SetInsertPoint(bb);
9  Argument& arg1 = *fn->arg_begin();
10 Argument& arg2 = *(fn->arg_begin() + 1);
11 Instruction* val = builder.CreateFAdd(&arg1, &arg2);
12 builder.CreateRet(val);
```

Listing 3.2 Funktion zum Addieren zweier Gleitkommazahlen in LLVM-IR

```
1  define float @foo(float, float) {
2    %3 = fadd float %0, %1
3    ret float %3
4  }
```

Module werden im In-Memory-Format durch die `Module`-Klasse dargestellt (siehe Zeile 2). Weiterhin werden Module hier immer innerhalb eines Kontexts betrachtet, wobei ein Kontext mehrere

Module umfassen kann. Dieser Kontext wird beispielsweise verwendet, um Konstanten und Typen zu unifizieren. Im In-Memory-Format wird er durch die `LLVMContext`-Klasse dargestellt (siehe Zeile 1).

SSA-Werte sind im In-Memory-Format Instanzen einer Klasse aus der `Value`-Klassenhierarchie (siehe Abb. 3.1). Register befinden sich immer in SSA-Form (siehe Abschnitt 2.1.6), somit können sie eindeutig durch eine Instruktion oder ein Argument identifiziert werden. Im In-Memory-Format wird also keine explizite Darstellung der Register aus dem Assembly-Format benötigt [LLVc]. In Zeile 12 kann man sehen, dass die Instanz der `fadd`-Instruktion von der Instanz der `ret`-Instruktion als Operand benutzt wird.

3.2 Module-Klasse

Da es Fälle gibt, in denen man ein Modul innerhalb verschiedener Kontexte verwenden will, ergibt es keinen Sinn den `LLVMContext` zu serialisieren.[1] Deshalb wird beim SKilL-Format genauso wie beim Assembly- und Bitcode-Format der `LLVMContext` nicht serialisiert.

Im In-Memory-Format enthält die Klasse `Module` das Feld `DL` vom Typ `DataLayout`, welches beschreibt, wie die Daten im Speicher der Zielplattform angeordnet werden [LLVd]. Zu jeder `DataLayout`-Instanz gibt es eine entsprechende Stringrepräsentation. Da sich ein `DataLayout`-Objekt nur über eine solche Stringrepräsentation konstruieren lässt, wird im SKilL-Format ebenfalls diese Repräsentation verwendet. Dadurch wird das Konvertieren in das In-Memory-Format trivial. Allerdings müssen Werkzeuge, die das SKilL-Format nutzen, diesen String parsen, wenn sie Informationen daraus benötigen.

Weiterhin enthält die Klasse `Module` die Felder `ValSymTab`, `ComdatSymTab` und `NamedMDSymTab`. Diese Symboltabellen können verwendet werden, um auf `Value`-, `Comdat`- und `NamendMDNode`-Objekte zuzugreifen. Da die SKilL-Spezifikationssprache `map`-Typen unterstützt, könnte man diese Symboltabellen leicht hinzufügen. Der zusätzliche Speicherbedarf würde sich in Grenzen halten, da es im SKilL-Graphen zwar viele Objekte mit Typ `Value` gibt, diese jedoch meist keinen Namen haben. Ein Problem ist jedoch, dass sich Werkzeuge selbst darum kümmern müssten, diese Symboltabellen konsistent zu halten. Dies ist der Fall, wenn entsprechende Objekte entfernt, eingefügt oder ersetzt werden. Die Symboltabellen sind also kein Teil der SKilL-Spezifikation, um Werkzeugen einen zusätzlichen Implementierungsaufwand zu ersparen. Falls sie dennoch benötigt werden, können sie auch vom Werkzeug selbst erstellt werden.

Das Feld `OwnedMemoryBuffer` wird im Zusammenhang mit der Bitcode-Deserialisierung verwendet. Das Feld `Materializer` ist ebenfalls nur für die Implementierung der Deserialisierung relevant. Beide Felder werden daher nicht in der SKilL-Spezifikation benötigt.

3.3 Value-Klassenhierarchie

Listing 3.3 zeigt die SKilL-Spezifikation für die Klasse `Value`.

[1] Ein Modul kann zum Beispiel Teil einer Bibliothek sein.

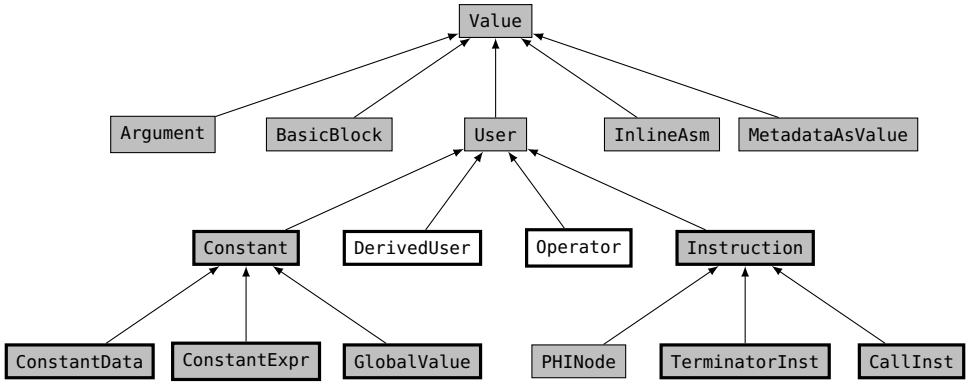


Abbildung 3.1: Die `Value`-Klassenhierarchie im In-Memory-Format. Fett umrahmte Knoten enthalten weitere nicht dargestellte Unterklassen. Knoten mit weißem Hintergrund sind im SKilL-Format nicht vorhanden.

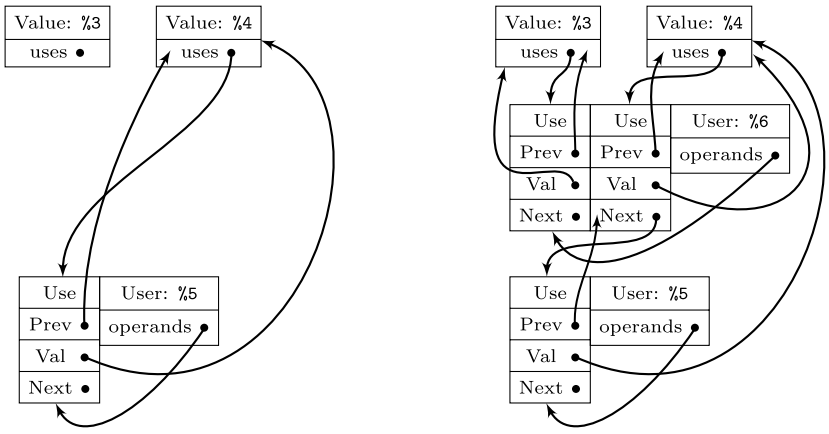
Listing 3.3 SKilL-Spezifikation von `Value`

```
1  @abstract
2  Value {
3    custom cpp
4    !include "llvm/IR/Value.h"
5    "llvm::Value*" llvmPtr;
6
7    @nonnull
8    list<User> users;
9    list<uint> useOperandNo;
10
11   @nonnull
12   Type type;
13   string name;
14   bool isUsedByMetadata;
15 }
```

Listing 3.4 SKilL-Spezifikation von `User`

```
1  @abstract
2  User : Value {
3    @nonnull
4    Value[] operands;
5  }
```

Eine Änderung bei der Klasse `Value` ist, dass das Feld `SubclassID` nicht Teil der SKilL-Spezifikation ist. Da die LLVM-Bibliotheken keine Laufzeittypinformationen (RTTI, englisch *runtime type information*) verwenden, dient dieses Feld dazu den konkreten Typ eines Objektes zu bestimmen. Dafür gibt es die Funktions-Templates `isa<T>`, `cast<T>`, `dyn_cast<T>`, `cast_or_null<T>` und `dyn_cast_or_null<T>`, die sich, im Gegensatz zu `dynamic_cast<T>`, auch für Klassen ohne V-Table nutzen lassen [LLVf]. Dieses Feld ergibt im SKilL-Format wenig Sinn, da der Programmierer des Werkzeugs, ohne die obigen Templates, die Werte in den Unterklassen kennen muss. Das Visitor-Pattern in Java oder Pattern-Matching in Scala ist bedeutend einfacher zu verstehen. Ferner müssten Werkzeuge das Feld auch beim Generieren von `Value`-Objekten immer mit dem korrekten Wert besetzen.



(a) Nach Zeile 3 **(b)** Nach Zeile 4

Abbildung 3.2: Eine vereinfachte Darstellung von Listing 3.5 im Speicher. Sich berührende Objekte haben benachbarte Speicheradressen. Zeiger ohne Ziel sind Nullzeiger.

Listing 3.5 Use Beispiel

```
1  %3 = load i32, i32* %1
2  %4 = load i32, i32* %2
3  %5 = trunc i32 %4 to i8
4  %6 = add i32 %3, %4
```

3.3.1 User

Die `User`-Klasse (siehe Listing 3.4) ist die Basisklasse für alle `Value`-Objekte, die andere `Value`-Objekte als Operanden verwenden.

Eine zentrale Datenstruktur im In-Memory-Format ist die Use-List. Hierbei handelt es sich um eine verkettete Liste, welche die Def-Use- und Use-Def-Ketten repräsentiert. Listing 3.5 und Abbildung 3.2 zeigen beispielhaft, wie die Use-List aufgebaut ist.[2] Um den Speicherverbrauch zu reduzieren, wird der Zeiger auf das `User`-Objekt in den 2 LSB (englisch *least significant bit*) der `Use::Prev` Felder kodiert [LLVf].[3] Mit dem Waymarking Algorithmus (siehe [LLVf]) kann der Zeiger wieder rekonstruiert werden.

Solch eine Datenstruktur lässt sich nicht durch das SKilL-Format darstellen, weil es hier keine Kontrolle darüber gibt, an welcher Speicheradresse Objekte abgelegt werden. Selbst wenn dies möglich wäre, so müsste bei einer vergleichbaren Modellierung jedes Werkzeug den Waymarking Algorithmus implementieren, um die Def-Use-Ketten zu benutzen. Daher wird die Use-List im SKilL-Format durch die Felder `users`, `useOperandNo` und `operands` dargestellt (siehe Listing 3.3 und 3.4). Die `User`- und `Value`-Objekte werden also direkt referenziert und die Position des Operanden wird im Feld `useOperandNo` kodiert.

[2] Das Beispiel zeigt nur den Fall für `User`, bei denen es eine feste Anzahl von Operanden gibt.
[3] Diese Bits werden aufgrund der Speicherausrichtung nicht benötigt.

Diese Modellierung der Operanden hat den Vorteil, dass die Use-Def-Ketten einfach iterierbar sind, selbst wenn man den konkreten Typ des `User`-Objekts nicht kennt. Ein Nachteil ist allerdings, dass das generierte API weniger aussagekräftig ist, denn dem Programmierer des Werkzeugs müssen nun die Positionen spezieller Operanden in der Liste bekannt sein. Als Beispiel wird die `store`-Instruktion, für das Schreiben von Daten an eine Speicheradresse, betrachtet. Der erste Operand enthält hier den zu schreibenden Wert, der zweite Operand die Speicheradresse.

Gäbe es in SKilL die Möglichkeit, eine Sicht auf Arrayelemente zu definieren, so ließe sich das generierte API einfacher nutzen. Für diesen Zweck könnte sich das bereits vorhandene `view`-Konstrukt (siehe [Fel17b], Abschnitt 4.4.4) erweitern lassen. Ein Vorschlag für die mögliche Syntax hierfür ist in Listing 3.6 gegeben. Damit es zu keinen Zugriffen außerhalb der Grenzen kommt, müsste beim Serialisieren und Deserialisieren geprüft werden, ob das Array größer ist als der maximale Index in solch einem `view`. Da dieser Vorschlag jedoch nicht implementiert wurde, wird die Semantik der Operanden momentan durch Kommentare vermittelt.

Listing 3.6 Vorschlag für Erweiterung der SKilL-Spezifikationssprache

```
1  StoreInst : Instruction {
2    view operands[0] as Value valueOperand;
3    view operands[1] as Value pointerOperand;
4
5    AtomicOrdering ordering;
6    string syncScope;
7    uint alignment;
8    bool isVolatile;
9  }
```

3.3.2 DerivedUser

In Abbildung 3.1 sieht man, dass die `DerivedUser`-Klasse nicht in der SKilL-Spezifikation vorkommt. Diese Klasse dient als Erweiterungspunkt der `Value`-Klassenhierarchie für Analysen. Ein Nutzer hiervon ist die MemorySSA-Analyse [LLVe]. Der Grund für diesen Erweiterungspunkt ist, dass dadurch die Use-List-Funktionalität genutzt werden kann [LLVb]. Ein Beispiel für diese Funktionalität ist das Ersetzen eines `Value`-Objekts bei allen `User`-Objekten, die dieses verwenden (RAUW, englisch *replace all uses with*). Unterklassen von `DerivedUser` sind jedoch kein Teil von LLVM-IR und werden somit auch nicht vom Assembly- und Bitcode-Format serialisiert.

Im SKilL-Format gibt es nicht die Funktionalität, die das In-Memory-Format für die Use-Listen bietet. Durch die Änderungstoleranz von SKilL lässt sich die `Value`-Klassenhierarchie trotzdem von Werkzeugen erweitern, sollte dies erforderlich sein. Für die MemorySSA-Analyse würde es ausreichen, ein Feld für `MemoryPhi` in `BasicBlock` und ein Feld für `MemoryUse` oder `MemoryDef` in Unterklassen von `Instruction` hinzuzufügen.

3.3.3 Operator

Die `Operator`-Klasse ist ebenfalls kein Teil der SKilL-Spezifikation. Diese Klasse ist eine Abstraktion der gemeinsame Funktionalität von Instruktionen und konstanten Ausdrücken [LLVb]. Es

gibt also keine Instanzen von Klassen aus diesem Teil der Klassenhierarchie und somit wird sie auch nicht im SKilL-Format benötigt.

3.3.4 BasicBlock

Die Klasse `BasicBlock` repräsentiert im In-Memory-Format einen Grundblock. Ihre Spezifikation (siehe Listing 3.7) dient hier als Beispiel für Klassen aus der `Value`-Klassenhierarchie, die bezüglich der Serialisierung redundante Felder enthalten. Das Feld `parent` ist nicht notwendig, weil bereits eine Referenz in die andere Richtung besteht. Die Redundanz ist jedoch erwünscht, da Werkzeuge so auch direkt auf dem SKilL-Graphen arbeiten können, ohne vorher entsprechende `auto`-Felder zuzuweisen.

Listing 3.7 SKilL-Spezifikation von `BasicBlock`

```
1  BasicBlock : Value {
2    @nonnull
3    list<Instruction> instList;
4    @nonnull
5    Function parent;
6  }
```

Ein weiteres Beispiel für redundante Daten sind die Def-Use-Ketten (siehe Abschnitt 3.3.1). Hier könnte man ebenfalls auf eine explizite Serialisierung verzichten, weil sie sich aus den Use-Def-Ketten herleiten lassen. In diesem Fall entspricht die SKilL-Spezifikation also dem In-Memory-Format, aber nicht dem Bitcode-Format.

3.3.5 CmpInst

Die `CmpInst`-Klasse ist Basisklasse für die `FCmpInst`- und `ICmpInst`-Klassen. Diese repräsentieren die `fcmp`- und `icmp`-Instruktionen zum Vergleichen von Gleitkommazahlen und Integern. Die `CmpInst`-Klasse enthält den Aufzählungstyp `Predicate`, welcher die verschiedenen Arten von Vergleichen aufzählt. Anders als im In-Memory-Format gibt es im SKilL-Format für Prädikate auf Gleitkommazahlen und Integern jeweils einen eigenen Aufzählungstyp. Somit kann nicht der Zustand eintreten, dass beispielsweise eine `ICmpInst` ein Prädikat für Gleitkommazahlen benutzt.

3.3.6 BinaryOperator

Im In-Memory-Format wird die `SubclassID` aus der `Value`-Klasse auch verwendet, um den Opcode von binären Operatoren zu kodieren. Da diese Information im SKilL-Format nicht vorhanden ist, gibt es hier für jeden möglichen Opcode eine eigene Klasse. Dadurch können auch optionale Flags dieser Instruktionen präziser modelliert werden. Zum Beispiel ergeben Fast-Math-Flags nur Sinn für Operatoren, die auf Gleitkommazahlen arbeiten. Solche zusätzlichen Eigenschaften werden in SKilL durch Interfaces spezifiziert.

Bei dieser Modellierung gibt es mehr Typen, deren Beschreibung serialisiert werden muss. Da es sich aber nur um 18 Klassen handelt, ist der Overhead hierfür vernachlässigbar.

3.3.7 CallInst

Die `CallInst`-Klasse repräsentiert die `call`-Instruktionen im In-Memory-Format. Außerdem ist sie dort auch die Basisklasse von Klassen für den Aufruf bestimmter intrinsischer Funktionen. Hierbei wird nur eine kleine Auswahl der intrinsischen Funktionen abgedeckt. Im In-Memory-Format definieren diese Klassen Zugriffsmethoden für bestimmte Operanden. Da sich mit dem SKilL-API solche Zugriffsmethoden auf Arrayelemente nicht realisieren lassen (siehe Abschnitt 3.3.1), werden diese Klassen nicht in die SKilL-Spezifikation übernommen. Werkzeuge können spezielle intrinsische Funktionen auch anhand ihres Namens identifizieren.

In LLVM-IR ist es möglich, `call`- und `invoke`-Instruktionen mit sogenannten Operand-Bundles zu annotieren. Hierbei handelt es sich jeweils um ein Paar von einem Bezeichner und potenziell mehreren SSA-Werten [LLVd]. Die `Use`-Instanzen für diese zusätzlichen Operanden werden zusammen mit den restlichen `Use`-Instanzen gespeichert. Die Operand-Bundle-Operanden werden also wie die normalen Operanden der `User`-Klasse gesehen. Da es im SKilL-Format keine `Use`-Klasse gibt, werden die Operanden aus den Operand-Bundles sowohl vom Feld `operands` der Klasse `User`, als auch vom Feld `inputs` der Klasse `OperandBundleUse` referenziert. Falls ein Operand ersetzt wird, muss die Referenz also an beiden Stellen geändert werden.

3.3.8 ConstantDataSequential

Im In-Memory-Format gibt es die Basisklasse `ConstantDataSequential` für konstante Arrays oder Vektoren, die nur einfache Integer oder Gleitkommazahlen enthalten.[4] Zuvor wurden solche Daten, wie bei anderen Typen auch, als Operanden eines `ConstantAggregate`-Objekts gespeichert. Der Zweck von `ConstantDataSequential` ist eine Reduktion des Speicherbedarfs, da nicht mehr ein `Constant`-Objekt mit einer Use-List für jedes Element benötigt wird.[5]

Ein Problem bei dieser Modellierung ist, dass hier direkt auf den Binärdaten gearbeitet wird. Die LLVM-Bibliotheken bieten Methoden zum Konvertieren der Daten. Werkzeuge, die das SKilL-Format nutzen, müssten diese Konvertierung selbst implementieren. Daher wird im SKilL-Format so vorgegangen, wie im In-Memory-Format vor der Einführung der `ConstantDataSequential`-Klasse, mit der Ausnahme, dass die Elemente keine Einträge in ihrer Use-List haben. Dies geschieht, um unerwartete Konflikte in der Ordnung der Use-List zu vermeiden.

3.3.9 ConstantExpr

Die `ConstantExpr`-Klasse repräsentiert konstante Ausdrücke aus LLVM-IR. Konstante Ausdrücke entsprechen Instruktionen ohne Seiteneffekten, bei denen alle Operanden Konstanten sind [LLVd]. Im In-Memory-Format werden hier durch Verwendung eines Opcodes verschiedene Ausdrücke

[4] Genauer 1/2/4/8-Byte-Integer und IEEE-754-Gleitkommazahlen mit einfacher oder doppelter Genauigkeit [LLVb].
[5] Siehe https://bugs.llvm.org/show_bug.cgi?id=1324

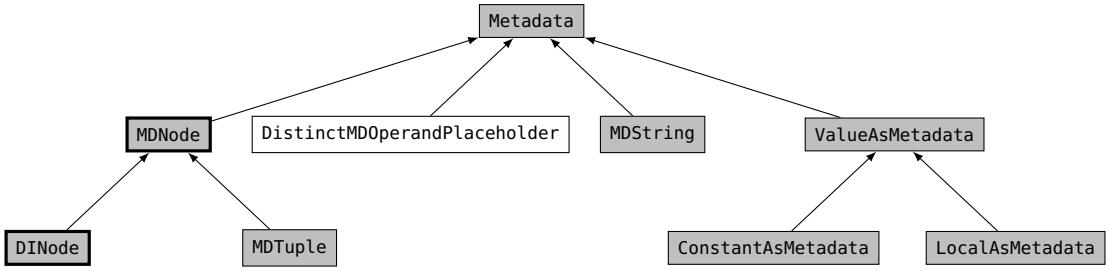


Abbildung 3.3: Die `Metadata`-Klassenhierarchie im In-Memory-Format. Fett umrahmte Knoten enthalten weitere nicht dargestellte Unterklassen. Knoten mit weißem Hintergrund sind im SKilL-Format nicht vorhanden.

durch gemeinsame Klassen repräsentiert. Beispielsweise gibt es für konstante Ausdrücke mit unären Operatoren nur die `UnaryConstantExpr`-Klasse. Damit sich Programmierer von Werkzeugen beim Benutzen des SKilL-Formats keine Opcodes merken müssen, gibt es hier für jede Art von Ausdruck eine eigene Klasse.

3.4 Metadata-Klassenhierarchie

LLVM-IR-Metadaten sind im In-Memory-Format durch Klassen aus der `Metadata`-Klassenhierarchie repräsentiert (siehe Abb. 3.3). Bis auf wenige Unterschiede ist das SKilL-Format identisch zum In-Memory-Format.

Ein Unterschied ist, dass die Operanden in der `MDNode`-Klasse (siehe Listing 3.8) direkt die Metadaten sind. Im In-Memory-Format gibt es dazwischen noch die `MDOperand`-Klasse. Die Klasse wird dort benötigt, um die Referenz auf den tatsächlichen Operanden zu aktualisieren, falls dieser ersetzt oder gelöscht wird. Da es diese Funktionalität beim SKilL-Format nicht gibt, wird die `MDOperand`-Klasse nicht benötigt. Ein weiterer Unterschied ist, dass es im SKilL-Format die `DistinctMDOperandPlaceholder`-Klasse nicht gibt. Instanzen dieser Klasse werden bei `distinct` Knoten als Platzhalter für Vorwärtsreferenzen verwendet.

Listing 3.8 SKilL-Spezifikation von `MDNode`

```
1  @abstract
2  MDNode : Metadata {
3    list<Metadata> operands;
4  }
```

Beim Erzeugen der Metadaten im SKilL-Graphen werden die obigen Mechanismen nicht benötigt, weil hier keine strukturelle Unifikation stattfindet. Im Gegenzug wird es dadurch aber auch schwieriger im SKilL-Format korrekte Metadaten zu erzeugen.

Wie man in Listing 3.8 sieht, werden die Operanden bei `MDNode`-Unterklassen in einem Array gespeichert. Wie bei den Operanden aus den `User`-Unterklassen erfolgt der Zugriff also auch über einen Index. Hier ergibt es allerdings keinen Sinn auf die Operanden zuzugreifen, ohne den konkreten Typ des `MDNode`-Objekts zu kennen. Ferner können bestimmte Operanden Nullzeiger

enthalten. Daher werden die Operanden in zukünftigen Versionen vermutlich durch Felder in den Unterklassen dargestellt, für die sich einzelne Restriktionen definieren lassen.

3.5 Weitere Klassen

In diesem Abschnitt werden einige Klassen außerhalb der `Value`- und `Metadata`-Klassenhierarchien beschrieben, bei denen die Modellierung vom In-Memory-Format abweichen muss.

3.5.1 Listen

Im In-Memory-Format kommen an vielen Stellen verkettete Listen vor. Damit sind Operationen wie das Löschen und Einfügen von Elementen sowie das Zusammenfügen von Listen in konstanter Zeit durchführbar. Diese Operationen sollten bei Transformationen der IR möglichst schnell sein, da sich beispielsweise Instruktionen oder Grundblöcke häufig ändern können. Mit Ausnahme der Use-List (siehe Abschnitt 3.3.1) werden doppelt verkettete Listen als Spezialisierung von `ilist<T>` implementiert. Dabei handelt es sich um sogenannte Intrusive-Listen, bei denen die Zeiger für die benachbarten Elemente in den Elementen selbst abgelegt sind [LLVf]. Dafür muss die Klasse des Elements von `ilist_node<T>` erben, wodurch es an vielen Stellen zu Mehrfachvererbung kommt.

Eine gleichartige Modellierung wäre auch in SKilL mit Interfaces möglich, die `auto`-Felder für benachbarte Elemente bereitstellen. Da es in SKilL keine generischen Interfaces gibt, müsste für jeden Typ, der in solch einer Liste vorkommt, ein Interface definiert werden. Bei dieser Modellierung bleibt dann aber die Eigenschaft erhalten, dass jedes Element in genau einer Liste enthalten sein kann. Der Nachteil dieser Modellierung ist, dass man die Algorithmen zur Bearbeitung der Liste in den Werkzeugen implementieren müsste. Dies beinhaltet auch den Aufbau der Liste selbst durch das Zuweisen der `auto`-Felder.

Aus diesem Grund werden Listen im SKilL-Format anders modelliert. In SKilL gibt es bereits einen Typen `list<T>`, der in dem generierten API auf eine passende Datenstruktur abgebildet wird. In Scala ist dies ein `scala.collection.mutable.ListBuffer[T]`, in Java eine `java.util.LinkedList<T>` und in C++ ein `std::vector<T>`. Ein Nachteil dabei ist, dass der SKilL-Typ `list<T>` keinerlei Garantien über das Laufzeitverhalten der Operationen gibt, da ein `std::vector<T>` nicht durch eine verkettete Liste implementiert ist. Daher sollte geprüft werden, wie effizient sich die verschiedenen Datenstrukturen in der Praxis verhalten.[6] Ein weiterer Nachteil ist, dass nun beispielsweise eine Instruktion in zwei Grundblöcken enthalten sein kann.

3.5.2 Aufzählungstypen

Die SKilL-Spezifikationssprache unterstützt zwar `enum`-Typen, allerdings hat sich herausgestellt, dass sich diese schlecht in verschiedenen Programmiersprachen integrieren lassen [Fel17a]. Die Aufzählungstypen aus dem In-Memory-Format werden stattdessen im SKilL-Format als Integer

[6] Falls es wenige Elemente gibt, kann ein `std::vector<T>` durchaus schneller beim Einfügen sein als eine `std::list<T>`.

mit eingeschränktem Wertebereich modelliert, was bei C++ `enum`-Typen problemlos möglich ist. Die Namen der Aufzählung werden über einen Kommentar in der Spezifikation vermittelt. Ein Beispiel für diese Art der Spezifikation ist in Listing 3.9 gegeben.

Listing 3.9 Aufzählungstyp in SKilL-Spezifikation

```
1  /**
2   * @see "llvm/IR/InlineAsm.h:33"
3   *
4   * @note 0 = ATT
5   * @note 1 = Intel
6   */
7  typedef AsmDialect
8    !pragma enum
9    @range(0,1)
10   i8;
```

3.5.3 APInt und APFloat

Da LLVM-IR Integer-Typen mit beliebiger Bitweite erlaubt, müssen diese auch im In-Memory-Format dargestellt werden. Dafür gibt es die Klasse `APInt`, die es auch erlaubt mit solchen Integern zu rechnen.[7] Zum Speichern der Daten wird dabei eine Union aus `uint64_t` und `uint64_t*` verwendet. Für Integer mit weniger als 64 Bit werden die Daten direkt gespeichert. Für Integer mit mehr als 64 Bit wird in ein Zeiger auf die Daten gespeichert. Zusätzlich gibt es ein Feld, welches die Bitweite enthält.

Im SKilL-Format wird Rücksicht darauf genommen, dass sich große Integer auch in anderen Programmiersprachen leicht nutzen lassen. Daher werden alle Integer mit mehr als 64 Bit in ihrer dezimalen Stringrepräsentation gespeichert. Die Stringrepräsentation lässt sich beispielsweise in Java mit `java.math.BigInteger` und in Scala mit `scala.math.BigInt` nutzen. Integer mit weniger als 64 Bit werden ebenfalls direkt gespeichert (siehe Listing 3.10).

Die LLVM-Bibliotheken enthalten weiterhin die Klasse `APFloat` für Gleitkommazahlen mit beliebiger Präzision. Hiermit werden im In-Memory-Format die sechs Typen für Gleitkommazahlen aus LLVM-IR dargestellt. Die SKilL-Spezifikationssprache unterstützt IEEE-754-Gleitkommazahlen mit einfacher und doppelter Genauigkeit [Fel17b]. Diese können also direkt gespeichert werden. Bei den restlichen Typen werden die Daten in einer hexadezimalen Stringrepräsentation gespeichert (siehe Listing 3.11).

[7] Das ist zum Beispiel beim Falten von Konstanten nötig.

Listing 3.10 SKilL-Spezifikation von `APInt`

```
@abstract
APInt {
  i32 bitWidth;
}

SmallInt : APInt {
  v64 value;
}

BigInt : APInt {
  @nonnull
  string value;
}
```

Listing 3.11 SKilL-Spezifikation von `APFloat`

```
@abstract
APFloat {}

IEEEhalfFloat : APFloat {
  @nonnull
  string value;
}

IEEEsingleFloat : APFloat {
  f32 value;
}

IEEEdoubleFloat : APFloat {
  f64 value;
}

IEEEquadFloat : APFloat {
  @nonnull
  string value;
}

X87DoubleExtendedFloat : APFloat {
  @nonnull
  string value;
}

PPCDoubleDoubleFloat : APFloat {
  @nonnull
  string value;
}
```

4 Implementierung

Ein Ziel dieser Arbeit ist es, einen Konverter zu schreiben, der zwischen den vorhandenen Serialisierungsformaten und dem SKilL-Format übersetzen kann. Dafür wurden die beiden Klassen `ModuleSkillWriter` und `ModuleSkillReader` geschrieben. Erstere erstellt für ein gegebenes Modul im In-Memory-Format ein Modul im SKilL-Format, letztere macht das Gleiche in die andere Richtung. Mithilfe dieser beiden Klassen wurden dann die Werkzeuge `ll2sf` und `sf2ll` zum Konvertieren der Formate geschrieben. Dabei werden die Funktionen zum Serialisieren und Deserialisieren des Assembly- und Bitcode-Formats aus den LLVM-Bibliotheken genutzt. Des Weiteren wurden verschiedene Werkzeuge an das SKilL-Format angebunden oder speziell dafür geschrieben. In diesem Kapitel werden einige Details bezüglich der Implementierung betrachtet.

4.1 Serialisierung

In diesem Abschnitt wird die Funktionsweise der Klasse `ModuleSkillWriter` beschrieben. Prinzipiell geht es hier darum, einen Graphen zu kopieren. Die Konvertierung ist in zwei Phasen aufgeteilt.

In der ersten Phase wird für jedes `Value`-, `Metadata`- oder `Type`-Objekt ein entsprechender Knoten im SKilL-Graphen erzeugt. Dabei wird auch dem `custom`-Feld `llvmPtr` des Knotens das entsprechende In-Memory-Objekt zugewiesen. Durch Hashtabellen lassen sich die Knoten des SKilL-Graphen den In-Memory-Objekten zuordnen. Das Modul wird zunächst entsprechend der syntaktischen Struktur der Sprache traversiert, da diese einen Baum aufspannt. Im Teilgraphen der Metadaten kann es jedoch Zyklen geben und der Teilgraph der Konstanten ist ein gerichteter azyklischer Graph. Deshalb werden diese beiden Teilgraphen durch einen rekursiven Funktionsaufruf ausgewertet. Die Hashtabellen werden dabei genutzt, um bereits besuchte Objekte zu erkennen.

Um den konkreten Typ der In-Memory-Objekte zu bestimmen, wird der RTTI-Mechanismus aus den LLVM-Bibliotheken verwendet. In `llvm/IR/{Value,Instruction,Metadata}.def` sind Makros definiert, mit denen sich auf kompakte Weise zu jeder Klasse im In-Memory-Format die entsprechende Klasse im SKilL-Format bestimmen lässt.

Beim Konvertieren der `ConstantDataSequential`-Objekte müssen die Elemente zuerst zu `Constant`-Objekten umgewandelt werden. Dafür stellt die Klasse `ConstantDataSequential` bereits die Methode `getElementAsConstant` bereit.

In der zweiten Phase wird über die Knoten des Graphen iteriert. Mit dem Wert des Feldes `llvmPtr` und den Hashtabellen können alle Felder, ohne eine spezielle Behandlung von Vorwärtsreferenzen oder Zyklen, berechnet werden.

Manche Strings in LLVM-IR können Nullzeichen (englisch *null character*) enthalten. Diese stellen für SKilL kein Problem dar. Es wird lediglich davon abgeraten sie zu benutzen, weil es zu Problemen mit dem C-API führen kann [Fel17b]. Da es für die LLVM-Bibliotheken bereits eine offizielle C-Sprachanbindung gibt, ist das von SKilL generierte C-API von geringer Relevanz. Der API-Generator für C++ musste modifiziert werden, damit sich Strings mit Nullzeichen erzeugen lassen.

Die Implementierung der Klasse `ModuleSkillWriter` benötigt 1223 SLOC (englisch *source lines of code*).[1] Der Serialisierungscode für das Assembly-Format hat 2884 SLOC.[2] Der Serialisierungscode für das Bitcode-Format hat 4873 SLOC.[3]

4.2 Deserialisierung

In diesem Abschnitt wird die Funktionsweise der Klasse `ModuleSkillReader` beschrieben. Wie bei der Serialisierung muss auch hier wieder ein Graph kopiert werden. Allerdings gibt es durch das LLVM-API bestimmte Vorgaben beim Erzeugen von Objekten.

Wie zuvor werden die Knoten des Graphen erst entsprechend der syntaktischen Struktur der Sprache besucht, allerdings zuerst nur bis auf die Ebene der Grundblöcke. Beim Erzeugen von `Instruction`-Objekten müssen die Operanden bereits bekannt sein. Globale Werte und Grundblöcke wurden bereits vor allen Instruktionen besucht. Falls es keine unerreichbaren Grundblöcke gibt, müssen zyklische Abhängigkeiten zwischen Instruktionen einen ϕ-Knoten enthalten. Die ϕ-Knoten werden erst vervollständigt, nachdem alle Instruktionen einer Funktion besucht wurden. Somit können die Operanden durch einen rekursiven Funktionsaufruf erzeugt werden. Hier werden allerdings keine Hashtabellen benötigt, weil besuchte Knoten mit dem `custom`-Feld `llvmPtr` identifizierbar sind.

Um die konkreten Typen der Objekte im SKilL-Graphen zu unterscheiden, wird das Visitor-Pattern verwendet. Der API-Generator für C++ kann den dafür benötigten Code erzeugen.

Das LLVM-API erlaubt eine Deserialisierung einzelner Funktionen nach Bedarf. Hierfür muss dem Modul eine Implementierung der Klasse `GVMaterializer` übergeben werden. Diese ist momentan noch nicht für das SKilL-Format vorhanden. Es werden also immer alle Funktionen eines Moduls deserialisiert.

Die Implementierung der Klasse `ModuleSkillReader` benötigt 2262 SLOC. Der Parser für das Assembly-Format hat 6675 SLOC.[4] Der Parser für das Bitcode-Format hat 7542 SLOC.[5]

[1] Das generierte C++-API hat 55276 SLOC und wird hierbei nicht mitgezählt.
[2] Code aus `include/llvm/IR/ModuleSlotTracker.h` und `lib/IR/AsmWriter.cpp` im LLVM-Projektverzeichnis
[3] Entsprechender Code aus `include/llvm/IR/Bitcode` und `lib/IR/Bitcode/Writer` im LLVM-Projektverzeichnis
[4] Code aus `include/llvm/AsmParser` und `lib/AsmParser` im LLVM-Projektverzeichnis
[5] Entsprechender Code aus `include/llvm/IR/Bitcode` und `lib/IR/Bitcode/Reader` im LLVM-Projektverzeichnis

Algorithmus 4.1 Metadaten deserialisieren

```
1:  function VISIT(n : SKilL-MDNode)
2:      if n.llvmPtr ≠ null then          // Ziel einer Querkante oder mehrerer Rückwärtskanten
3:          return n.llvmPtr
4:      end if
5:      if n.distinct then                // distinct Knoten können direkt erzeugt werden
6:          n.llvmPtr ← DistinctMDNode()
7:      else if n.visited = false then    // Ziel einer Rückwärtskante
8:          n.llvmPtr ← TemporaryMDNode()
9:          return n.llvmPtr
10:     end if
11:     n.visited ← true
12:     operands ← [ ]                    // Leeres Array
13:     for all i ∈ n.operands do         // Besuche alle Operanden des SKilL-Knoten
14:         operands.push(VISIT(i))
15:     end for
16:     if n.distinct then                // Setze Operanden bei distinct Knoten
17:         n.llvmPtr.setOperands(operands)
18:     else if n.llvmPtr ≠ null then     // Ersetze temporären Knoten durch unifizierten Knoten
19:         n.llvmPtr.RAUW(UniqueMDNode(operands))
20:     else                              // Erzeuge unifizierten Knoten
21:         n.llvmPtr ← UniqueMDNode(operands)
22:     end if
23:     return n.llvmPtr
24: end function
```

4.2.1 Metadaten

Metadaten werden in LLVM-IR strukturell unifiziert, sofern sie nicht explizit als `distinct` markiert sind (siehe Abschnitt 2.1.8). Zyklische Abhängigkeiten von Metadaten stellen somit eine Schwierigkeit dar. Da die Darstellung zu jedem Zeitpunkt unifiziert ist, müssen Zyklen durch temporäre Knoten aufgelöst werden. Temporäre Knoten werden nicht unifiziert und lassen sich zu einem späteren Zeitpunkt ersetzen. Ein naiver Ansatz wäre, für jede Vorwärtsreferenz einen temporären Knoten zu erzeugen. Das Ersetzen von temporären Knoten ist jedoch eine teure Operation. Daher versucht der für diese Arbeit entworfene Algorithmus die Anzahl der temporären Knoten zu reduzieren.

Algorithmus 4.1 zeigt das Vorgehen als Pseudocode. Die Eingabe der Funktion VISIT ist ein Metadatenknoten aus dem SKilL-Graphen. Die Ausgabe ist ein Metadatenknoten aus dem In-Memory-Format. Die `custom`-Felder `llvmPtr` und `visited` werden mit dem Wert `null` bzw. `false` initialisiert. Der Graph wird in DFS-Ordnung (englisch *depth-first search order*) traversiert. Beim Besuch jedes Knotens wird vorher geprüft, ob dieser über eine Rückwärtskante (englisch *retreating edge*) oder eine Querkante (englisch *cross edge*) erreicht wurde. Die Art der Kante lässt sich durch die beiden Felder `llvmPtr` und `visited` bestimmen. Somit wird für jeden Zyklus

mindestens ein temporärer Knoten eingefügt.[6] Nachdem in Zeile 15 alle Operanden eines Knotens besucht wurden, sind die folgenden Fälle möglich. Für `distinct` Knoten werden die erzeugten Operanden zugewiesen. Falls dem eigenen `llvmPtr`-Feld ein temporärer Knoten zugewiesen wurde, kann dieser nun mit den Operanden durch einen neuen unifizierten Knoten ersetzt werden. Andernfalls wird einfach ein neuer unifizierter Knoten mit den Operanden erzeugt.

Nachdem alle temporäre Knoten im Metadatengraphen ersetzt wurden, muss die Methode `resolveCycles` für mindestens einen Knoten aus jedem Zyklus aufgerufen werden [LLVb].

4.2.2 Unerreichbare Grundblöcke

In Abschnitt 2.1.7 wurde gezeigt, dass durch unerreichbare Grundblöcke zyklische Abhängigkeiten ohne ϕ-Knoten entstehen können. Wie zuvor erwähnt, müssen die Operanden bekannt sein, um ein `Instruction`-Objekt erzeugen zu können. Die rekursive Auswertung der Operanden kann also in einem Stapelüberlauf resultieren.

Damit diese zyklischen Abhängigkeiten die Deserialisierung nicht stören, muss die Implementierung noch angepasst werden. Wie auch schon bei den Metadaten könnten an den entsprechenden Stellen temporäre Werte erzeugt werden.

4.2.3 Ordnung der Use-List

Beim Erstellen eines Objekts im In-Memory-Format wird die Use-List automatisch angepasst. Durch die rekursive Auswertung stimmt die Ordnung der Use-List möglicherweise nicht mehr mit der Ordnung vor der Serialisierung überein. Dies macht es erforderlich die Use-List zu sortieren, falls eine Erhaltung der Ordnung erwünscht ist. Bei einer Änderung der Use-List-Ordnung können manche Analysen und Transformationen zu unterschiedlichen Ergebnissen kommen.[7]

Zum Sortieren der Use-List gibt es in der `Value`-Klasse bereits die Methode `sortUseList`, bei der eine Merge-Sort-Implementierung verwendet wird. Der Methode muss eine Funktion übergeben werden, die für jeweils zwei `Use`-Objekte die Ordnung angibt.

Da die Use-List-Ordnung im SKilL-Format bereits korrekt ist, muss diese Ordnung nur noch auf die `Use`-Objekte des In-Memory-Formats übertragen werden. Einen Spezialfall bei dieser Abbildung gibt es, falls ein `User` für mehrere Operanden den gleichen `Value` verwendet. Hier muss dann zusätzlich die Information aus dem Feld `useOperandNo` im SKilL-Format genutzt werden, um die korrekte Ordnung zu bestimmen.

4.3 Werkzeuge

Um die Funktionsfähigkeit des SKilL-Formats zu demonstrieren, werden verschiedene Werkzeuge daran angebunden. Einerseits werden bereits existierende Werkzeuge aus dem LLVM-Projekt erweitert. Andererseits werden neue Werkzeuge speziell für das SKilL-Format geschrieben.

[6] Falls mehrere Zyklen verschachtelt sind, können auch mehrere temporäre Knoten in einem der Zyklen vorkommen.
[7] Siehe https://bugs.llvm.org/show_bug.cgi?id=5680

4.3.1 Anbindung der LLVM-Werkzeuge

Im Folgenden werden die LLVM-Werkzeuge aufgezählt, die bereits an das SKilL-Format angebunden sind.

Manche LLVM-Werkzeuge können die serialisierte Darstellung über einen Standard-Datenstrom lesen oder schreiben. Das SKilL-API sieht diesen Fall jedoch nicht vor und erlaubt es nur, durch Angabe eines Pfads, über das Dateisystem zu lesen und zu schreiben. Vorerst wird eine nicht portable Lösung für einige Unix-artige Systeme verwendet, bei der aus `/dev/stdin` gelesen und nach `/dev/stdout` geschrieben wird.

opt

Das Werkzeug `opt` kann auf modulare Weise Analysen und Optimierungen auf LLVM-IR-Programmen durchführen. Der Benutzer kann vorgeben, welche Pässe ausgeführt werden und in welcher Reihenfolge dies stattfindet. Es besteht auch die Möglichkeit, externe Pässe aus dynamischen Bibliotheken zu laden.[8]

Für die Anbindung des SKilL-Formats muss der Code zum Deserialisieren des LLVM-IR-Programms angepasst werden. Um das SKilL-Format auch für die Ausgabe nutzen zu können, ist ein `ModulePass` für die Serialisierung notwendig. Eine zusätzliche Kommandozeilenoption `-sf` wird definiert, um das SKilL-Format als Ausgabeformat wählbar zu machen.

verify-uselistorder

Das Werkzeug `verify-uselistorder` dient dazu, die Erhaltung der Use-List-Ordnung beim Serialisieren zu testen. Im ersten Schritt wird ein LLVM-IR-Programm aus einer serialisierten Datei geladen. Es wird eine bijektive Abbildung erzeugt, bei der jedes `Value`-Objekt einer Nummer zuordnet wird.[9] Das Modul wird nun unter Erhaltung der Use-List-Ordnung in eine temporäre Datei serialisiert und anschließend wieder deserialisiert. Nachdem die Abbildung erneut berechnet wurde, kann für jedes `Value`-Objekt geprüft werden, ob sich die Use-List-Ordnung geändert hat. Der Test ist bestanden, falls es hierbei keinen Unterschied gibt und die Wohlgeformtheit des Moduls verifiziert ist. Der Test kann beliebig oft für zufällige Permutationen der Use-List-Ordnung wiederholt werden. Standardmäßig wird zu jeder Permutation auch immer die umgekehrte Ordnung getestet.

Für die Anbindung des SKilL-Formats muss der Code für das Lesen und Schreiben der temporären Dateien angepasst werden. Dafür wird die Klasse `TempFile` um die Methoden `readSkillfile` und `writeSkillfile` erweitert. Die Funktion `verifyUseListOrder` muss angepasst werden, sodass das SKilL-Format auch verwendet wird. Als letzte Anpassung werden noch die drei Kommandozeilenoptionen `-ll`, `-bc` und `-sf` definiert, damit sich die Formate auch einzeln testen lassen.

[8] Siehe https://releases.llvm.org/5.0.1/docs/CommandGuide/opt.html

[9] Eine Ausnahme sind Werte, die nicht serialisiert werden, wie zum Beispiel Konstanten ohne Verwendungen.

lli und llc

Mit dem Werkzeug `lli` lässt sich LLVM-IR direkt ausführen, wobei entweder ein Interpreter oder ein JIT-Compiler verwendet wird.[10] Das Werkzeug `llc` kann LLVM-IR in Assembler-Sprache für bestimmte Architekturen übersetzen.[11]

Für die Anbindung des SKilL-Formats muss lediglich der Code für das Deserialisieren des Moduls angepasst werden.

4.3.2 SKilL-basierte Programmanalyse

In dieser Arbeit soll demonstriert werden, dass es möglich ist, direkt auf der SKilL-basierten Darstellung zu arbeiten. Dies ist notwendig, damit die Darstellung auch in anderen Programmiersprachen sinnvoll nutzbar ist.

Die implementierte Programmanalyse berechnet die McCabe-Metrik [McC76] für jede Funktion eines gegebenen LLVM-IR-Programms. Das ursprüngliche Ziel dieser Software-Metrik war es, zu quantifizieren wie hoch der Aufwand zum Warten und Testen eines Programms ist [McC76].

Es gibt viele Werkzeuge, die diese Software-Metrik für verschiedene Programmiersprachen berechnen können wie beispielsweise das Eclipse Metrics Plugin.[12] Für gewöhnlich arbeiten diese Werkzeuge auf einer AST-Darstellung, sind also jeweils immer nur für eine Sprache nutzbar. In dieser Hinsicht bietet es sich an, ein Werkzeug zu schreiben, das auf einer Darstellung arbeitet, die unabhängig von der Quellsprache ist. An dieser Stelle sei jedoch angemerkt, dass sich die Resultate durch Anwendung von Optimierungen verändern können. Falls es darum geht, die Anzahl von benötigten Testfällen zu bestimmen, kann das von Vorteil sein, da der CFG möglicherweise vereinfacht wird. Wenn es darum geht, schwer verständliche Funktionen für ein Refactoring zu identifizieren, sollte jedoch auf Optimierungen verzichtet werden.

McCabe-Metrik

Für einen Kontrollflussgraphen G mit n Knoten und e Kanten sowie einem Ausgang ist die McCabe-Metrik durch

$$V(G) = e - n + 2$$

definiert [McC76]. Kontrollflussgraphen mit mehreren Ausgängen lassen sich transformieren, sodass sie nur noch einen Ausgang enthalten. Dafür wird ein neuer Ausgangsknoten mit eingehenden Kanten von allen alten Ausgangsknoten hinzugefügt. Gibt es in einem Kontrollflussgraphen s Ausgänge, so lässt sich die McCabe-Metrik nun als

$$V(G) = e - n + s + 1$$

definieren, weil s Kanten und ein Knoten hinzugefügt werden müssen, um den Kontrollflussgraphen zu transformieren [Har84]. Der CFG aus Abbildung 2.2 hat demnach den Wert 3 in der McCabe-Metrik.

[10] Siehe https://releases.llvm.org/5.0.1/docs/CommandGuide/lli.html
[11] Siehe https://releases.llvm.org/5.0.1/docs/CommandGuide/llc.html
[12] Siehe http://eclipse-metrics.sourceforge.net

Implementierung

Um die Analyse für LLVM-IR zu implementieren, muss also bestimmt werden, wie viele Knoten, Kanten und Ausgänge es im CFG gibt. Beim Iterieren über alle Grundblöcke einer Funktion kann die Knotenzahl einfach hochgezählt werden. Die letzte Instruktion in einem Grundblock entscheidet über den weiteren Kontrollfluss. Daher entspricht jeder Operand einer terminierenden Instruktion mit Typ `label` einer Kante im CFG. Grundblöcke, die mit einer `return`-Instruktion terminieren, sind offensichtlich Ausgänge im CFG.

Zusätzliche Ausgangsknoten können durch den expliziten Kontrollfluss der Ausnahmebehandlung (englisch *exception handling*) entstehen. Dies betrifft die Instruktionen `catchswitch` und `cleanupret`, welche nur optional einen Nachfolger im CFG haben und ansonsten die Funktion verlassen. Ferner ist hier auch die Instruktion `resume` zu beachten, die in jedem Fall die Funktion verlässt.

Eine weitere Möglichkeit für einen Ausgangsknoten besteht durch die Instruktion `unreachable`. Für den Fall, dass es noch weitere Instruktionen im Grundblock gibt, muss die vorletzte Instruktion ein `call` mit Attribut `noreturn` sein. Dies entspricht also auch einem Ausgang im CFG. Falls es nur die Instruktion `unreachable` im Grundblock gibt, kann dieser wirklich nicht erreicht werden. In diesem Fall muss die Kantenzahl um die Anzahl der eingehenden Kanten reduziert werden.[13]

Des Weiteren gibt es in LLVM-IR noch die `select`-Instruktion, welche dem ternären Operator in C entspricht. Hier wird die Knoten- und Kantenzahl so inkrementiert, dass sich dieser Fall wie eine gewöhnliche Verzweigung verhält. Da eine `select`-Instruktion an einer beliebigen Stelle im Grundblock vorkommen kann, reicht es nicht, nur die terminierende Instruktion zu betrachten.

Eine Annahme, die bei der Implementierung getroffen wird, ist, dass der CFG aus nur einer Zusammenhangskomponente besteht. Beispiele, wie in Abschnitt 2.1.7 beschrieben, werden also nicht berücksichtigt. Diese Annahme ist realistisch, weil der CFG in der Regel nur bei der Anwendung von Transformationen in mehrere Zusammenhangskomponenten zerfallen kann. Bei nicht optimierten Programmen werden keine Transformationen angewendet. Bei optimierten Programmen werden solche Grundblöcke in der Regel entfernt.

Um den Funktionsnamen ohne Name-Mangling und die Position im Quelltext zu bekommen, werden Debug-Informationen benutzt. Jede Funktion zeigt auf einen `DISubprogram`-Metadatenknoten. Dieser enthält unter anderem den Funktionsnamen und ein `DIFile`-Metadatenknoten als Operanden. Aus letzterem lässt sich der Dateiname und die Zeilennummer der Funktion entnehmen.

4.3.3 SKilL-basierter Compiler

Neben der Analyse ist auch das Generieren neuer Programme von Interesse. Es soll demonstriert werden, dass sich die SKilL-basierte Darstellung ebenfalls für diesen Zweck eignet. Hierfür wurde ein Front-End für die Sprache Kaleidoscope in Scala implementiert.

[13] Die Gleichung für die McCabe-Metrik setzt voraus, dass der CFG stark zusammenhängend wird, wenn eine Kante vom Ausgangsknoten zum Eingangsknoten hinzugefügt wird (vgl. [McC76], Satz 1).

Kaleidoscope

Die Sprache Kaleidoscope dient als einfaches Beispiel für die Implementierung eines LLVM-Front-Ends. Für die Sprachen C++ und OCaml gibt es eine Anleitung, die in mehrere Schritte gegliedert ist [LLVg]. Der SKilL-basierte Compiler ist für die vollständige Sprachvariante aus Schritt 8 geschrieben. Eine kleine Änderung ist, dass die Zeichen „`(`“, „`)`“ und „`;`“ nicht als Operatoren erlaubt sind, da die Sprache sonst nicht mehr in LL(1) ist.[14,15]

Listing 4.1 zeigt ein Beispielprogramm in Kaleidoscope. In der ersten Zeile wird der binäre „`:`“ Operator mit Präzedenz 1 definiert. Der Operator reiht mehrere Ausdrücke aneinander und gibt die rechte Seite als Ergebnis zurück.[16] Von Zeile 3 bis 9 wird die Funktion für die iterative Berechnung der Fibonacci-Folge definiert.[17] In Zeile 11 wird diese Funktion dann aufgerufen.

Listing 4.1 Fibonacci-Folge in Kaleidoscope [LLVg]

```
1  def binary : 1 (x y) y;
2
3  def fibi(x)
4    var a = 1, b = 1, c in
5    (for i = 3, i < x in
6      c = a + b :
7      a = b :
8      b = c) :
9    b;
10
11 fibi(10)
```

Implementierung

Lexer und Parser des Front-Ends sind mit Parserkombinatoren geschrieben.[18] Für den AST werden algebraische Datentypen verwendet. Die AST-Knoten werden dann in späteren Pässen via Pattern-Matching unterschieden.

Der in Scala geschriebene Zwischencodegenerator funktioniert analog zum Zwischencodegenerator im Referenzcompiler. Hier nutzen LLVM-Front-Ends für gewöhnlich die Klasse `IRBuilder` (siehe Listing 3.1). Eine Instanz dieser Klasse hält eine Referenz auf den aktuellen Grundblock und bietet Fabrikmethoden, um Instruktionen darin einzufügen. Die Klasse `IRBuilder` ist so weit in Scala nachgebaut, wie es für das Front-End erforderlich ist. Listing 4.2 zeigt als Beispiel

[14] Der Referenzcompiler verwendet rekursiven Abstieg im Parser und stürzt bei entsprechender Eingabe ab.

[15] Kaleidoscope erlaubt es beliebige Symbole als binäre oder unäre Operatoren zu definieren, solange diese nicht bereits definiert sind. Die Präzedenz kann dabei als ganze Zahl zwischen 1 (schwächste) und 100 (stärkste) frei gewählt werden.

[16] Dieser Operator ist notwendig, weil es in Kaleidoscope keine Anweisungen (englisch *statements*) gibt, sondern nur Ausdrücke (englisch *expressions*).

[17] Im Referenzcompiler wird die Bedingung der Schleife erst nach der Auswertung des enthaltenen Ausdrucks geprüft. Daher stimmt die Funktion `fibi` nur für Argumente, die größer als 2 sind. In der Scala Implementierung wurde dieses Verhalten nachgebildet.

[18] Siehe https://github.com/scala/scala-parser-combinators

die Fabrikmethode für die Instruktion `fadd`. Weiterhin müssen Konstanten und Typen unifiziert werden, was bei dieser einfachen Sprache jedoch nur wenige Zeilen Code benötigt.

Listing 4.2 Eine der Fabrikmethoden aus dem Scala `IRBuilder`

```
1  def createFAdd(lhs: Value, rhs: Value, name: String = "") = {
2    val inst = sf.FAddOperator.make(‘type‘ = lhs.‘type‘, parent = bb,
3                            fastMathFlags = fmf, name = name,
4                            operands = ArrayBuffer(lhs,rhs))
5    bb.instList += inst; inst
6  }
```

Insgesamt braucht der auf Scala und SKilL basierende Compiler 692 SLOC, wohingegen der auf C++ und den LLVM-Bibliotheken basierende Compiler 812 SLOC benötigt.[19] Trotz des zusätzlichen Aufwands aufseiten der Scala/SKilL Implementierung lässt sich also auf kompakte Weise Code generieren.

4.3.4 Zusammenfassung

Die vorangegangenen Abschnitte haben gezeigt, dass Programmanalysen und Zwischencodeerzeugung für LLVM-IR mit dem SKilL-Format möglich sind. In Scala lässt sich durch das SKilL-Format auch vergleichsweise kompakterer Code schreiben.

Sowohl die Analyse, als auch das Front-End sind jedoch nur sehr einfache Beispiele. Es wurde noch kein Versuch unternommen, komplexe Metadatengraphen im SKilL-Format zu erzeugen. Außerdem könnten komplexere Analysen von der Funktionalität der LLVM-Bibliotheken profitieren. Werkzeuge, die ein LLVM-IR-Programm transformieren, wurden noch nicht geschrieben. Es ist also noch nicht klar, wie sehr die Wahl der Datenstrukturen in den Programmiersprachen die Laufzeit beeinflusst.

[19] Einschließlich der teilweise implementierten Scala `IRBuilder`-Klasse

5 Evaluation

In diesem Kapitel wird untersucht, inwiefern die Ziele dieser Arbeit (siehe Abschnitt 1.1) erfüllt wurden. Zunächst wird der Testaufbau für die durchgeführten Experimente betrachtet. Daraufhin werden die einzelnen Zielsetzungen untersucht. Im Anschluss an jede Untersuchung werden die jeweiligen Ergebnisse zusammengefasst.

Auf die Nutzbarkeit des generierten SKilL-API wurde bereits in den Abschnitten 4.3.2 und 4.3.3 eingegangen. In diesem Kapitel werden noch das Laufzeitverhalten und der Speicherbedarf der Implementierung untersucht.

5.1 Testaufbau

Um die Experimente nachvollziehbar zu machen, wird im folgenden Abschnitt der allgemeine Testaufbau beschrieben.

5.1.1 Testumgebung

Zur Ausführung aller Experimente wird ein System mit 4 AMD Opteron 6174 Prozessoren verwendet. Damit stehen insgesamt 48 CPU-Kerne zur Verfügung. Ferner hat das System 256 GB DDR3-Hauptspeicher. Das Betriebssystem ist Debian 8 mit Linux-Kernel 3.16.

Die Implementierung und Modellierung der SKilL-Darstellung basiert auf LLVM 5.0.1. Zum Erzeugen von LLVM-IR wird somit auch Clang 5.0.1 benutzt. Die SKilL-Anbindung ist mit dem Werkzeug `skill` in Commit `a966849bbbc2710a53eac99a54078915983d0b5e` generiert und mit der Bibliothek `cppCommon` in Commit `53cf9cf72f5cc6468593c768d0db19c4d6e99ea3` gelinkt. Der verwendete Commit des WLLVM-Projekts ist `6450800f2c0e1581ad0fcf72ee99a65d8c3b269f`. Das LLVM-Projekt, die SKilL-Anbindung und die Bibliothek `cppCommon` werden mit `gcc` 4.9.2 übersetzt. Für das LLVM-Projekt wird die Option `-DCMAKE_BUILD_TYPE=Release` verwendet. Für die SKilL-Anbindung, `cppCommon` und die SKilL-Werkzeuge werden die Optionen `-O2 -DNDEBUG` verwendet.

5.1.2 Testdaten

Zum Testen der Implementierung wird eine Auswahl von Open-Source-Projekten verwendet. Alle verwendeten Projekte sind entweder in C oder C++ geschrieben, können also von `clang`

Projekt	Version	Projekt	Version
Clang	5.0.1	GNU Screen	4.6.2
Git	2.16.2	GNU sed	4.4
GNU Bash	4.4	Gnuplot	5.2.2
GNU Bison	3.0.4	Jikes	1.22
GNU Core Utilities	8.29	LLVM	5.0.1
GNU Chess	6.2.5	Open MPI	3.0.0
GNU Go	3.8	Python	3.6.4
GNU Gzip	1.9	STREAM Benchmark	5.10
GNU Make	4.2	Vim	8.0.1542
GNU nano	2.9.4	XZ Utils	5.2.3

Tabelle 5.1: Projektversionen der Testdaten

übersetzt werden.[1] Als Tests für kleine Programme werden der STREAM Benchmark[2] und die GNU Core Utilities verwendet. Für größere Testdaten werden bekannte Programme wie `vim`, `git` und `clang` genutzt. Eine Auflistung aller Projekte und der verwendeten Version ist in Tabelle 5.1 zu sehen. Um auch Testdaten für sehr kleine Programme zu haben, gibt es noch ein leeres C Programm (empty), ein Hello-World-Programm in C (hello) und das Fibonacci-Programm aus Listing 2.1 (fib).

Um zu überprüfen, wie gut sich das SKilL-Format für verschiedene Anwendungen eignet, werden alle Projekte sowohl als Debug-Version (mit Optionen `-O0 -g`) als auch als Release-Version (mit Optionen `-O3 -g0`) übersetzt. Die Debug-Testdaten repräsentieren Fälle, bei denen der Bezug zum ursprünglichen Programm erhalten bleiben soll, wie beispielsweise bei Programmanalysen. Die Release-Testdaten repräsentieren Fälle, in denen dieser Bezug nicht notwendig ist, zum Beispiel beim Generieren oder Optimieren von Code. Ein Unterschied zwischen den beiden Versionen ist, dass die Debug-Testdaten mehr Metadaten enthalten. Die Metadaten werden dabei zur Darstellung der Debug-Informationen genutzt.

Die Laufzeit und der Speicherbedarf der Implementierung sind abhängig von der Größe des verwendeten SKilL-Graphen, also der Summe aus der Anzahl von Knoten und Kanten. Eine Regressionsanalyse der Testdaten hat ergeben, dass sich die Kantenzahl um einen konstanten Faktor von der Knotenzahl unterscheidet. Für die Debug-Testdaten ist dieser Faktor ca. vier, für die Release-Testdaten ca. sieben. Der SKilL-Graph ist also für praktische Daten dünn besetzt, weshalb in den folgenden Betrachtungen nur die Knotenzahl als unabhängige Variable verwendet wird.[3] Eine vollständige Auflistung der Knotenzahl für die Testdaten ist in Tabelle 5.2 und 5.3 gegeben.

[1] Es wäre möglich, mit anderen Front-Ends auch Projekte in weiteren Programmiersprachen zu verwenden. Darauf wird an dieser Stelle jedoch verzichtet, weil es für den zusätzlichen Aufwand keinen erkennbaren Mehrwert gibt.
[2] Siehe https://www.cs.virginia.edu/stream
[3] Für dünn besetzte Graphen (V, E) gilt $|E| \in \mathcal{O}(|V|)$.

Name	Knotenzahl	
	Debug	Release
empty	379	359
hello	527	452
fib	618	557
libstdbuf.so	1.301	768
make-prime-list	2.462	1.114
stream	3.870	2.700
stream_mpi	5.467	4.533
true	16.656	5.254
false	16.659	5.254
tty	16.984	5.371
hostid	17.017	5.352
logname	17.043	5.372
unlink	17.096	5.399
link	17.171	5.430
whoami	17.181	5.413
printenv	17.316	5.443
env	17.608	5.661
dirname	17.616	5.539
uname	17.631	5.961
echo	17.906	5.788
runcon	17.910	5.500
yes	17.954	5.620
sync	18.069	5.641
basename	18.080	5.653
users	18.305	5.929
rmdir	18.513	5.752
nice	18.762	6.283
pathchk	18.858	5.681
sleep	18.992	5.797
cksum	19.014	6.258
pwd	19.367	6.156
nohup	19.487	5.936
paste	19.532	6.376
getlimits	19.678	6.973
kill	19.815	6.736
groups	19.819	6.177
nproc	20.009	6.129
fold	20.030	6.417

Name	Knotenzahl	
	Debug	Release
expand	20.175	6.595
mkfifo	20.268	6.042
tee	20.331	6.253
unexpand	20.605	6.670
truncate	20.616	6.811
cat	21.154	6.948
nl	21.495	7.514
pinky	21.565	7.121
timeout	21.929	7.263
stdbuf	22.013	6.895
mknod	22.111	6.690
tsort	22.212	6.893
who	22.321	7.476
comm	22.415	6.897
tac	22.425	6.867
seq	22.501	7.109
printf	23.083	7.372
chroot	23.408	7.284
base64	23.562	7.363
fmt	23.840	8.578
dircolors	24.039	7.639
base32	24.055	7.560
sum	24.184	7.630
cut	24.595	8.053
id	24.722	7.441
uniq	25.370	8.140
test	25.546	7.630
head	26.252	7.918
mkdir	26.326	7.504
mktemp	26.650	7.593
expr	26.733	9.285
[	27.031	8.303
join	28.439	9.287
md5sum	28.711	9.283
wc	28.783	8.366
csplit	29.278	9.934
readlink	29.561	9.069
uptime	30.014	8.725

Tabelle 5.2: Liste der Testdaten und deren Knotenzahl im SKilL-Graphen

Name	Knotenzahl	
	Debug	Release
tr	30.142	10.118
stty	31.173	11.764
realpath	31.440	9.536
sha1sum	32.239	10.263
numfmt	33.310	11.200
split	33.481	10.497
od	34.777	11.213
xz	36.210	12.316
shred	38.454	10.972
dd	39.443	13.275
sha224sum	40.207	11.788
sha256sum	40.209	11.788
shuf	41.189	11.863
ptx	43.397	15.445
sha512sum	45.432	12.451
sha384sum	45.434	12.453
b2sum	45.566	13.023
chmod	45.861	13.774
chgrp	46.634	14.207
stat	46.912	23.550
ln	47.576	13.302
chcon	47.723	14.058
rm	47.936	14.588
chown	47.963	14.580
tail	48.667	14.656
factor	50.746	19.331
gzip	54.222	29.419
df	54.473	17.662
pr	73.017	17.868
cp	77.827	21.117
du	78.348	23.319
ginstall	85.521	22.719
sort	88.818	28.243
dir	91.878	31.348
ls	91.878	31.348
vdir	91.878	31.348
mv	93.141	25.325
sed	105.916	35.143
nano	145.403	67.409
make	158.555	49.649
touch	216.206	18.395
date	224.995	20.072
gnuchess	229.045	94.001
bison	324.266	116.366
screen	381.461	139.779
clang-tblgen	618.472	193.884
bash	768.818	256.383
jikes	783.373	452.764
gnuplot	911.912	345.310
libmpi.so	1.187.996	202.740
gnugo	1.229.808	617.350
git	1.869.078	601.669
python	1.990.134	659.005
vim	2.075.053	781.822
clang-format	2.086.688	528.080
llvm-dis	3.558.622	954.416
llvm-as	4.871.431	1.307.418
llvm-link	5.136.507	1.369.325
llvm-split	7.504.614	2.043.963
lli	18.564.725	5.357.733
clang-rename	20.309.030	6.176.210
clang-check	24.865.914	7.250.373
llc	32.082.915	10.324.510
bugpoint	33.045.658	10.765.684
opt	34.054.065	10.856.418
clang	35.482.681	9.044.428

Tabelle 5.3: Liste der Testdaten und deren Knotenzahl im SKilL-Graphen (Fortsetzung)

5.2 Abdeckung

Das erste Ziel dieser Arbeit ist die vollständige Abdeckung von LLVM-IR durch das SKilL-Format. Um die Abdeckung zu überprüfen, werden die Testdaten aus Abschnitt 5.1.2 verwendet. Dabei werden die SKilL-Werkzeuge `ll2sf` und `sf2ll` sowie die LLVM-Werkzeuge `llvm-as` und `llvm-dis` getestet.[4,5]

5.2.1 Round-Trip

Die erste Methode zum Testen der Abdeckung ist ein Round-Trip zwischen dem Assembly-Format und dem SKilL-Format (siehe Abb. 5.1). Da das Assembly-Format textbasiert ist, kann die Datei nach dem Round-Trip mit dem `diff`-Werkzeug des GNU-Projekts verglichen werden. Das `diff`-Werkzeug wird dabei mit Option `-I '^;'` aufgerufen, um Zeilen mit Kommentaren zu ignorieren. Diese sind nicht Teil von LLVM-IR und können sich somit beim Round-Trip ändern. Weiterhin wird konvertiert, ohne die Ordnung der Use-List zu erhalten, da dies in Abschnitt 5.2.2 separat getestet wird.

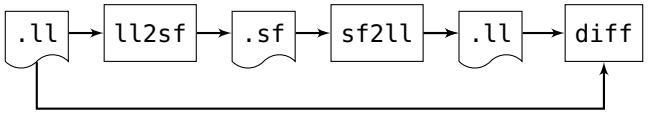


Abbildung 5.1: Round-Trip zwischen Assembly- und SKilL-Format

Hierbei sei angemerkt, dass eventuelle Bugs im Assembly-Format dazu führen könnten, dass Bugs im SKilL-Format verborgen bleiben. Da beim Verifizieren der Use-List-Ordnung (siehe Abschnitt 5.2.2) noch ein Round-Trip zum In-Memory-Format stattfindet, sollte das jedoch kein Problem darstellen.

Es gibt keine Differenz zwischen den Testdaten nach der Durchführung eines Round-Trips mit den SKilL-Werkzeugen. Ein Round-Trip zum Bitcode-Format mit den LLVM-Werkzeugen ist für diese Testdaten genauso in jedem Fall erfolgreich.

5.2.2 Ordnung der Use-List

Um die Erhaltung der Use-List-Ordnung zu testen, wird das Werkzeug `verify-uselistorder` verwendet (siehe Abschnitt 4.3.1). Es wird jeweils eine zufällige Permutation der Use-List benutzt, wodurch insgesamt vier verschiedene Ordnungen getestet werden.

Für die Testdaten aus Abschnitt 5.1.2 sind die Ergebnisse in Tabelle 5.4 zusammengefasst. Das SKilL-Format kann als einzige Darstellung die Use-List-Ordnung für alle Testdaten erhalten.

[4] Siehe https://releases.llvm.org/5.0.1/docs/CommandGuide/llvm-as.html
[5] Siehe https://releases.llvm.org/5.0.1/docs/CommandGuide/llvm-dis.html

Format	Erfolgsquote	
	Debug	Release
SKilL	100%	100%
Bitcode	92,4%	100%
Assembly	90,3%	85,4%

Tabelle 5.4: Erfolgsquote beim Erhalten der Use-List-Ordnung für die verschiedenen Serialisierungsformate mit den Testdaten aus Abschnitt 5.1.2

5.2.3 Tests aus dem LLVM-Projekt

Zusätzlich zu den vorherigen Tests werden noch Tests aus dem LLVM-Projekt verwendet.[6] Diese dienen normalerweise dazu, die Serialisierung und Deserialisierung durch das Assembly- und Bitcode-Format zu testen. Um die Tests zu definieren, wird das Assembly-Format verwendet, wobei die Regeln für die Ausführung der Tests in speziell formatierten Kommentaren stehen. Diese Kommentare werden vom Werkzeug `lit` (LLVM Integrated Tester) interpretiert und ausgeführt.

Da in den Tests auch Werkzeuge verwendet werden, die noch nicht an das SKilL-Format angebunden sind, wird zum Testen der SKilL-Werkzeuge hier `lit` nicht verwendet. Stattdessen sucht ein einfaches Bash-Skript (siehe Listing 5.1) nach Tests für einen Round-Trip und die Verifizierung der Use-List-Ordnung. Bei Round-Trips wird das Ergebnis mit dem des Bitcode-Formats verglichen. Das `verify-uselistorder` Werkzeug ist bereits an das SKilL-Format angebunden, es wird also einfach mit der hinzugefügten Option `-sf` ausgeführt.

Listing 5.1 Bash-Skript zum Ausführen der LLVM-Tests

```
#!/bin/bash
TESTS="Assembler/*.ll Bitcode/*.ll Feature/*.ll Feature/OperandBundles/*.ll"
for f in $TESTS; do
  # check for round-trip
  if grep -q '^; RUN: llvm-as < %s | llvm-dis' "$f"; then
    rm -f "$f.sf" "$f.bc"
    ../build/src/ll2sf "$f" && ../build/src/sf2ll "${f::-2}sf" -o "$f.sf"
    llvm-as "$f" && llvm-dis "${f::-2}bc" -o "$f.bc"
    if diff -u -I '^;' "$f.sf" "$f.bc"
    then echo "$f: PASS"; else echo "$f: FAIL"; fi
  fi
  # check for verify-uselistorder
  if grep -q '^; RUN: verify-uselistorder \(< \)\?%s$' "$f"; then
    if ../build/tools/llvm/verify-uselistorder/verify-uselistorder -sf "$f"
    then echo "$f: PASS"; else echo "$f: FAIL"; fi
  fi
done
```

Für die SKilL-Werkzeuge sind nur die beiden Testfälle `2004-02-27-SelfUseAssertError.ll` und `2004-06-07-VerifierBug.ll` nicht erfolgreich. Der Grund hierfür wurde bereits in Abschnitt 4.2.2 beschrieben.

[6] Dateien aus `test/{Assembler,Bitcode,Feature}` im LLVM-Projektverzeichnis.

Im Gegensatz zum Assembly- und Bitcode-Format kann das SKilL-Format auch für den `compatibility.ll` Testfall die Use-List-Ordnung erhalten. Hier scheitern die LLVM-Werkzeuge zurzeit wegen dem Bug aus PR24755.[7] Dieser Testfall ist außerdem ein guter Indikator für den Grad der Abdeckung, weil viele der Spracheigenschaften von LLVM-IR getestet werden.

5.2.4 Zusammenfassung

Die Implementierung schafft für einen repräsentativen Datensatz erfolgreich einen Round-Trip zum Assembly-Format. Im Gegensatz zu den LLVM-Werkzeugen lässt sich auch in jedem Fall die Use-List-Ordnung erhalten. Von 353 Tests aus dem LLVM-Projekt gibt es nur in zwei Fällen einen Fehler. Der Fehler besteht aufgrund falscher Annahmen bezüglich der Dominanzeigenschaften von totem Code. Das Problem sollte sich mit überschaubarem Aufwand beheben lassen. Allerdings lassen sich entsprechende Grundblöcke auch entfernen ohne die Semantik des Programms zu verändern. Somit kann das Ziel der Abdeckung als erfüllt betrachtet werden.

Durch die Tests wurden einige bisher unbekannte Bugs in der Bitcode- und Assembly-Implementierung entdeckt. Drei dieser Bugs (PR36778, PR36789 und PR37120) wurden auf einfache Testfälle reduziert und an die Entwickler gemeldet.[8,9,10]

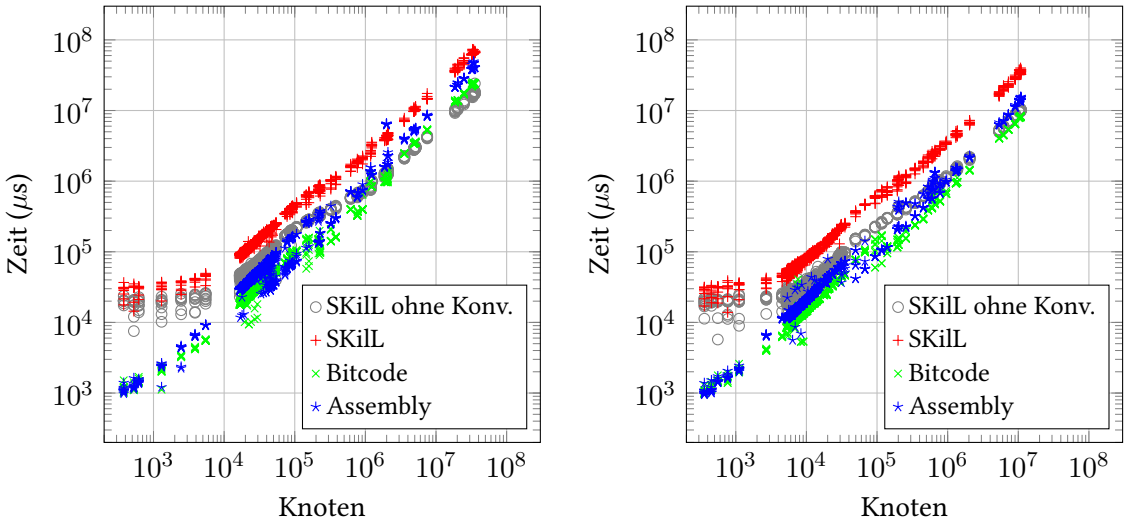
5.3 Laufzeit

LLVM-IR soll sich durch die SKilL-basierte Darstellung effizient bearbeiten lassen. Um dieses Ziel zu erreichen, muss die Laufzeit der Werkzeuge die gleiche Größenordnung haben wie bei den anderen Formaten. Die Ordnung der Use-List ist hauptsächlich beim Debuggen der Werkzeuge von Bedeutung (siehe Abschnitt 4.2.3). Daher wird bei den folgenden Zeitmessungen darauf verzichtet, diese zu erhalten.

5.3.1 Konvertierung

Zunächst wird untersucht, wie lange es dauert zwischen dem In-Memory-Format und dem SKilL-Format zu konvertieren. In die eine Richtung wird hierbei die Zeit zum Aufbauen des SKilL-Graphen und dessen anschließende Serialisierung gemessen. In die andere Richtung wird die Zeit für die Deserialisierung des SKilL-Graphen und den Aufbau des In-Memory-Formats gemessen. Als Vergleich dienen die entsprechenden Zeiten für das Bitcode- und Assembly-Format. Ferner werden die Laufzeiten für das SKilL-Format ohne Konvertierung betrachtet, welche eine untere Schranke für die Laufzeiten des SKilL-Formats mit Konvertierung bilden. Für jedes der Formate wurde ein Werkzeug geschrieben, welches die Laufzeit für das Serialisieren und Deserialisieren misst.

[7] Siehe https://bugs.llvm.org/show_bug.cgi?id=24755
[8] Siehe https://bugs.llvm.org/show_bug.cgi?id=36778 und https://lists.llvm.org/pipermail/llvm-dev/2018-March/121760.html
[9] Siehe https://bugs.llvm.org/show_bug.cgi?id=36789
[10] Siehe https://bugs.llvm.org/show_bug.cgi?id=37120



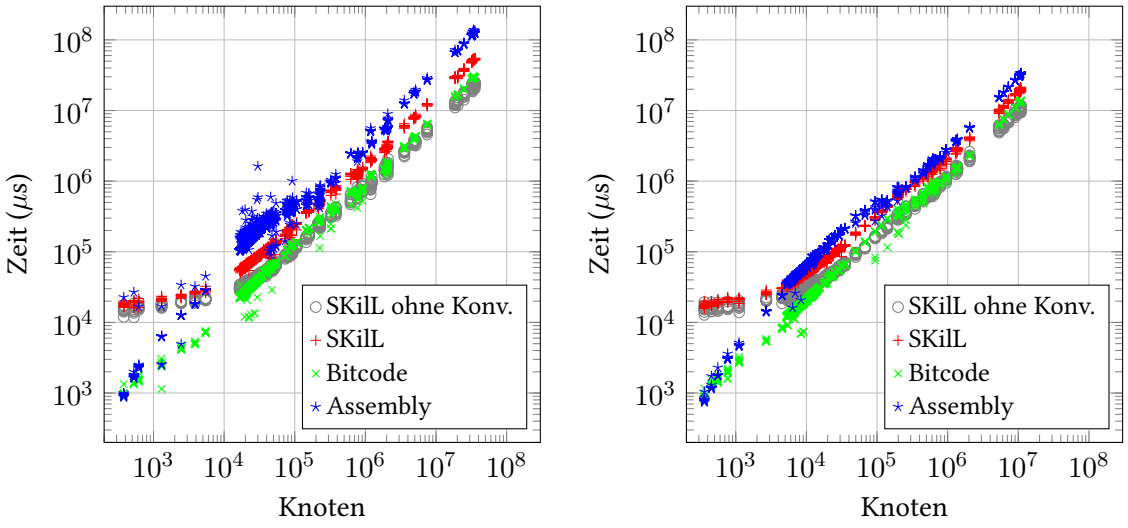
(a) Debug-Testdaten

(b) Release-Testdaten

Abbildung 5.2: Serialisierungszeit in Abhängigkeit von der Knotenzahl des SKilL-Graphen

	SKilL	Bitcode	Assembly
10^6 Knoten	1,951***	0,700***	1,211***
	(0,003)	(0,001)	(0,003)
Konstante	0,139***	−0,004	−0,014
	(0,019)	(0,004)	(0,020)
Beobachtungen	1.420	1.420	1.420
R^2	0,997	0,999	0,991
F-Statistik (df = 1; 1418)	467.692,500***	1.106.823,000***	162.599,100***
10^6 Knoten	3,240***	0,746***	1,291***
	(0,005)	(0,001)	(0,002)
Konstante	0,081***	0,006***	−0,009*
	(0,010)	(0,001)	(0,005)
Beobachtungen	1.420	1.420	1.420
R^2	0,996	0,999	0,995
F-Statistik (df = 1; 1418)	374.019,000***	1.562.437,000***	268.893,900***
Hinweis:			*p<0,1; **p<0,05; ***p<0,01

Tabelle 5.5: Lineare Regression über Serialisierungszeit in Abhängigkeit der Knotenzahl (oben Debug, unten Release). Konstante in Sek. und Steigung in Sek. pro 10^6 Knoten.



(a) Debug-Testdaten

(b) Release-Testdaten

Abbildung 5.3: Deserialisierungszeit in Abhängigkeit von der Knotenzahl des SKilL-Graphen

	SKilL	Bitcode	Assembly
10^6 Knoten	1,532***	0,855***	3,620***
	(0,001)	(0,001)	(0,004)
Konstante	0,062***	0,010*	−0,006
	(0,008)	(0,005)	(0,026)
Beobachtungen	1.420	1.420	1.420
R^2	0,999	0,999	0,998
F-Statistik (df = 1; 1418)	1.556.544,000***	1.158.002,000***	827.805,000***
10^6 Knoten	1,827***	1,235***	2,972***
	(0,002)	(0,001)	(0,002)
Konstante	0,060***	0,008***	0,030***
	(0,005)	(0,002)	(0,004)
Beobachtungen	1.420	1.420	1.420
R^2	0,998	0,999	0,999
F-Statistik (df = 1; 1418)	593.455,000***	1.654.705,000***	1.944.435,000***
Hinweis:			*p<0,1; **p<0,05; ***p<0,01

Tabelle 5.6: Lineare Regression über Deserialisierungszeit in Abhängigkeit der Knotenzahl (oben Debug, unten Release). Konstante in Sek. und Steigung in Sek. pro 10^6 Knoten.

Für alle Testdaten werden jeweils 10 Messungen nacheinander durchgeführt. Es ist zu erwarten, dass die erste Messung den höchsten Wert aufweist, da durch den Page-Cache und Festplattencache die folgenden I/O-Operationen beschleunigt werden. Dieses Vorgehen ist jedoch trotzdem berechtigt, weil das Format in Werkzeugketten genutzt werden soll. In der Praxis ist also auch zu erwarten, dass eine Datei mehrere Male nacheinander gelesen und beschrieben wird. Das Resultat dieser Messungen zeigen Abbildungen 5.2 und 5.3. Weiterhin wurde zu jeder Messreihe noch eine Lineare Regression durchgeführt, welche in Tabellen 5.5 und 5.6 zu sehen sind. Die Steigung und Konstante der Regressionsgerade ist in Sekunden pro 10^6 Knoten bzw. Sekunden angegeben.

Zunächst ist festzustellen, dass die Laufzeit der Implementierung für beide Richtungen mit hoher Signifikanz linear in der Knotenzahl ist. Dies war zu erwarten, da jeder Knoten und jede Kante nur konstant oft besucht wird und alle Operationen dabei eine erwartete Laufzeit in $\mathcal{O}(1)$ haben.[11] Ferner wurde bereits in Abschnitt 5.1.2 festgestellt, dass die Graphen dünn besetzt sind.

Entsprechend der Erwartungen ist auch, dass das SKilL-Format für kleine Dateien deutlich langsamer ist als das Assembly- und Bitcode-Format. Im SKilL-Format sind auch die Typen des Formats kodiert, wodurch es einen konstanten Overhead gibt. Für größere Dateien spielen diese zusätzlichen Kosten jedoch keine Rolle.

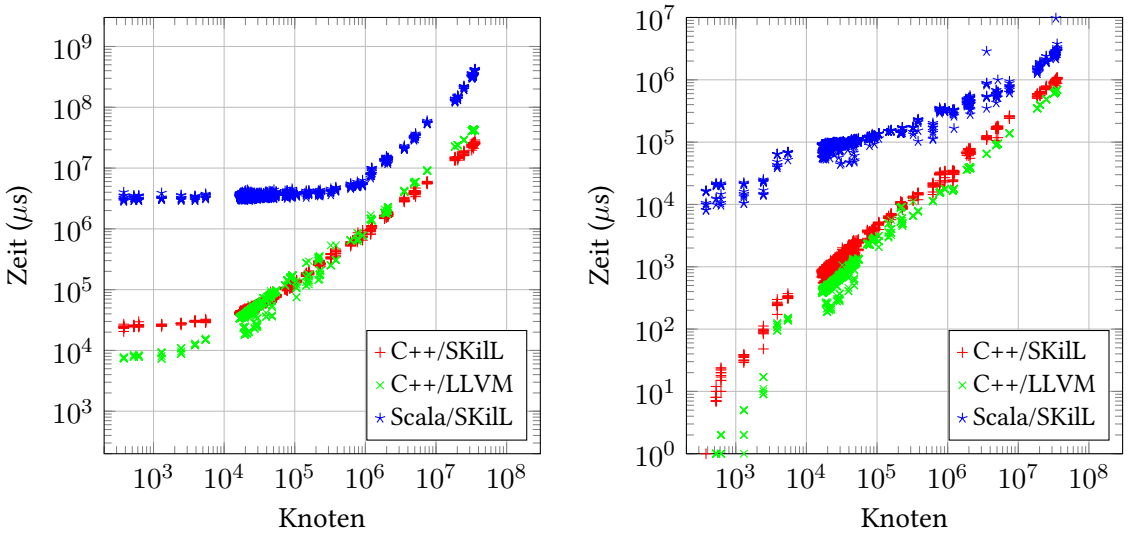
Bei der Serialisierung ist das SKilL-Format langsamer als das Bitcode- und Assembly-Format. Dies gilt sowohl für die Debug-Testdaten, als auch die Release-Testdaten. Für alle Formate ist die Laufzeit pro Knoten mit den Debug-Testdaten kürzer. Dieser Effekt ist für das SKilL-Format am stärksten ausgeprägt. Objekte vom Typ `Value` verursachen hier also einen größeren Overhead pro Knoten als Objekte vom Typ `Metadata`. Das liegt daran, dass viele Klassen der `Value`-Hierarchie Felder mit redundanten Daten enthalten (siehe Abschnitt 3.3.4).

Für manche Messreihen ist die Konstante der Regressionsgeraden negativ. Die Beträge sind jedoch jeweils maximal eine Standardabweichung. Daher liegt es nahe, dass es sich hierbei nur um Messungenauigkeit handelt und nicht um superlineares Laufzeitverhalten.

Bei der Deserialisierung ist das SKilL-Format schneller als das Assembly-Format und langsamer als das Bitcode-Format. Diese Ordnung gilt sowohl für die Debug-Testdaten als auch für die Release-Testdaten. Die Laufzeit ohne Debug-Informationen ist für das SKilL- und Bitcode-Format kürzer. Für das Assembly-Format ist das Lesen von Metadaten hingegen weniger effizient. Man kann erkennen, dass für das Assembly-Format ein erheblich größerer Aufwand zum Parsen nötig ist.

Das etwas schlechtere Ergebnis beim SKilL-Format entspricht den Erwartungen. Hier gibt es mit dem Aufbau des SKilL-Graphen einen zusätzlichen Zwischenschritt. Für die Laufzeit wäre also ein Faktor zwei im Vergleich zum Bitcode-Format eine realistische Erwartung, unter der Annahme, dass die beiden Formate ansonsten ähnlich effizient sind. Bei der Serialisierung ist das SKilL-Format etwas schlechter als diese Erwartung, bei der Deserialsierung etwas besser.

[11] Beim Aufbau des SKilL-Graphen werden Hashtabellen verwendet.



(a) Laufzeit des Werkzeugs

(b) Laufzeit der McCabe-Analyse

Abbildung 5.4: Laufzeit des McCabe-Werkzeugs in Abhängigkeit von der Knotenzahl des SKilL-Graphen

5.3.2 Werkzeuge

Um zu testen, wie das Laufzeitverhalten bei Werkzeugen in der Praxis ist, wird die McCabe-Analyse (siehe Abschnitt 4.3.2) untersucht. Das Werkzeug wurde in drei Varianten implementiert. Die erste Variante ist in C++ geschrieben und arbeitet direkt auf dem SKilL-Graphen (C++/SKilL). Variante zwei ist ebenfalls in C++ geschrieben, arbeitet jedoch auf dem In-Memory-Format und liest Dateien im Bitcode-Format (C++/LLVM). Die dritte Variante ist in Scala geschrieben und nutzt den SKilL-Graphen (Scala/SKilL). Alle Varianten sind möglichst idiomatisch programmiert. Zum Unterscheiden der verschiedenen Instruktionen nutzt Variante C++/SKilL das Visitor-Pattern, Variante C++/LLVM eine Spezialisierung von `InstVisitor` und Variante Scala/SKilL Pattern-Matching.

Hier werden nur die Debug-Testdaten verwendet, da das McCabe-Werkzeug Bezug auf den Quelltext nimmt (siehe Abschnitt 4.3.2). Wie in Abschnitt 5.3.1 wurden für alle Testdaten jeweils 10 Messungen nacheinander durchgeführt. Zum Messen der Laufzeit wurde das Werkzeug `perf stat` benutzt.[12] Die Ausgabe der Werkzeuge wurde nach `/dev/null` weitergeleitet. Die Ergebnisse sind in Abbildung 5.4 dargestellt.

Zunächst ist festzustellen, dass die Laufzeit der Analyse (siehe Abb. 5.4b) zwei bis vier Größenordnungen unter der Laufzeit des Werkzeugs (siehe Abb. 5.4a) liegt. Eine Regressionsanalyse hat für jedes Format ergeben, dass beide Laufzeiten linear in der Knotenzahl des SKilL-Graphen sind.

Die McCabe-Analyse ist bei Variante C++/LLVM etwas schneller als bei Variante C++/SKilL. Hinsichtlich des geringen Einflusses der Analyse spielt dies jedoch keine Rolle für die gesamte Laufzeit des Werkzeugs. Aus Abbildung 5.3a lässt sich entnehmen, dass das Deserialisieren des

[12] Siehe https://perf.wiki.kernel.org/index.php/Main_Page

SKilL-Graphen etwas schneller ist, als das Deserialisieren des Bitcode-Formats. Dementsprechend kann man auch in Abbildung 5.4a erkennen, dass das Werkzeug in Variante C++/SKilL für große Graphen am schnellsten ist.

Die Laufzeit der Variante Scala/SKilL hat für kleine Testdaten einen deutlichen Overhead, der sich durch das Starten der JVM (Java Virtual Machine) und den JIT-Compiler erklären lässt. Für die kleinsten Testdaten ist die Laufzeit dieser Variante ungefähr zwei Größenordnungen höher als die der Variante C++/SKilL. Für Testdaten mit mehr als 10^6 Knoten gibt es nur noch einen Unterschied von einer Größenordnung, was den Ergebnissen von Felden (siehe [Fel17a], Seite 183, Abb. 7.9) entspricht.

5.3.3 Zusammenfassung

Zusammenfassend lässt sich festhalten, dass die Laufzeiten bei der Konvertierung für alle getesteten Serialisierungsformate in der gleichen Größenordnung liegen. Das Serialisieren und Deserialisieren des SKilL-Graphen ist für große Testdaten in vielen Fällen schneller als das Konvertieren beim Bitcode-Format. Hierbei ist jedoch zu beachten, dass das In-Memory-Format auch Datenstrukturen enthält, die nicht im SKilL-Format abgebildet sind.[13]

Für diese einfache Analyse zeigt sich, dass Werkzeuge, die auf dem SKilL-Graphen arbeiten, eine vergleichbare Laufzeit haben, wie Werkzeuge die mit den LLVM-Bibliotheken arbeiten. Daraus lässt sich jedoch nicht schließen, wie das Laufzeitverhalten bei komplizierteren Analysen ist.

5.4 Speicherbedarf

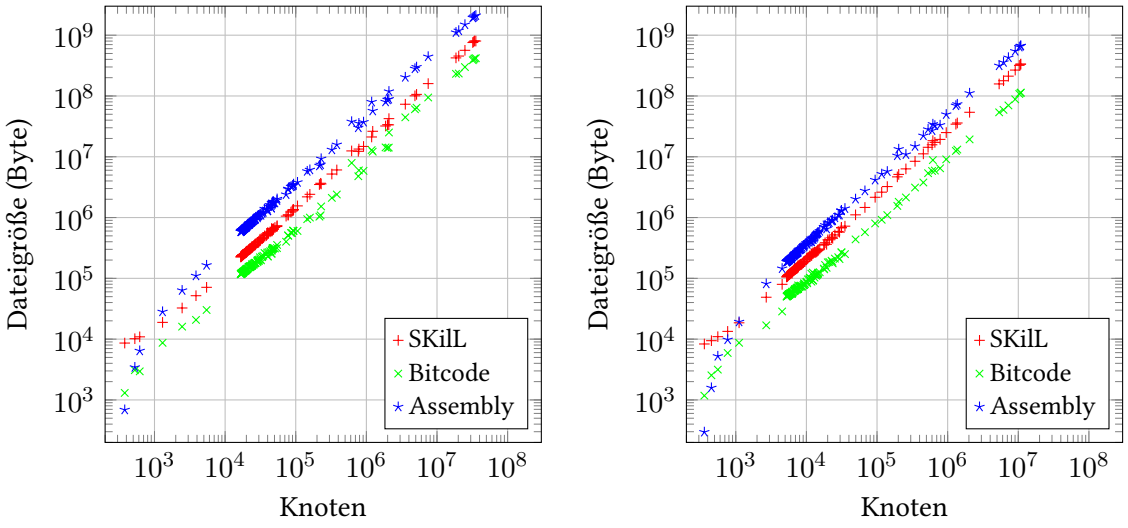
Neben der Laufzeit ist für die effiziente Bearbeitung auch der Speicherbedarf von Bedeutung. Es wird geprüft, wie groß die serialisierten Dateien beim SKilL-Format werden. Weiterhin wird untersucht, wie hoch der Hauptspeicherbedarf (Heap- und Stack-Allokationen) beim Konvertieren der Formate ist. Der Speicherbedarf sollte für beide Fälle in der gleichen Größenordnung liegen wie bei den anderen Formaten.

5.4.1 Dateigröße

Die Dateigröße der Testdaten in Abhängigkeit von der Knotenzahl ist in Abbildung 5.5 zu sehen. Tabelle 5.7 zeigt die lineare Regression der Messreihen. Die Steigung und Konstante der Regressionsgerade ist in MB (Megabyte) pro 10^6 Knoten bzw. MB angegeben.

Die Regressionsanalyse zeigt für alle Formate, dass die Größe mit hoher Signifikanz linear in der Knotenzahl des SKilL-Graphen ist. Auffällig ist hierbei, dass der konstante Anteil für alle Messreihen negativ ist. Die großen Testdaten teilen sich viel Code, da es sich um statisch gelinkte Werkzeuge aus den LLVM- und Clang-Projekten handelt. Es besteht die Möglichkeit, dass andere Programme mit ähnlich vielen Knoten weniger Speicherplatz benötigen. Durch den hohen Einfluss dieser Testdaten könnte der Wert der Konstante verzerrt werden.

[13] Ein Beispiel hierfür sind die Symboltabellen (siehe Abschnitt 3.2).



(a) Debug-Testdaten

(b) Release-Testdaten

Abbildung 5.5: Dateigröße in Abhängigkeit von der Knotenzahl des SKilL-Graphen

	SKilL	Bitcode	Assembly
10^6 Knoten	23,152***	12,032***	59,855***
	(0,058)	(0,029)	(0,083)
Konstante	−1,217***	−0,478**	−2,297***
	(0,379)	(0,187)	(0,541)
Beobachtungen	142	142	142
R^2	0,999	0,999	1,000
F-Statistik (df = 1; 140)	158.808,600***	177.055,700***	521.965,200***
10^6 Knoten	30,258***	10,185***	60,795***
	(0,082)	(0,034)	(0,155)
Konstante	−0,460***	−0,090	−1,046***
	(0,161)	(0,068)	(0,305)
Beobachtungen	142	142	142
R^2	0,999	0,998	0,999
F-Statistik (df = 1; 140)	137.681,800***	87.258,280***	154.794,600***
Hinweis:			*p<0,1; **p<0,05; ***p<0,01

Tabelle 5.7: Lineare Regression über Dateigröße in Abhängigkeit der Knotenzahl (oben Debug, unten Release). Konstante in MB und Steigung in MB pro 10^6 Knoten.

Verständlicherweise braucht das ASCII-basierte Assembly-Format für große Graphen den meisten Speicherplatz pro Knoten. Das SKilL-Format braucht für die Debug-Testdaten doppelt und für die Release-Testdaten dreimal so viel Speicherplatz pro Knoten wie das Bitcode-Format. Dies war zu erwarten, da im SKilL-Graphen manche Kanten redundant sind (siehe Abschnitt 3.3.4). Beim Bitcode-Format werden diese redundanten Informationen hingegen nicht serialisiert. Bei den Metadaten gibt es keine redundanten Felder, was erklärt, dass das Verhältnis für die Debug-Testdaten besser ist. Weiterhin kann man beim SKilL-Format den Overhead des kodierten Typsystems für Testdaten mit weniger als 1000 Knoten erkennen.

5.4.2 Hauptspeicher

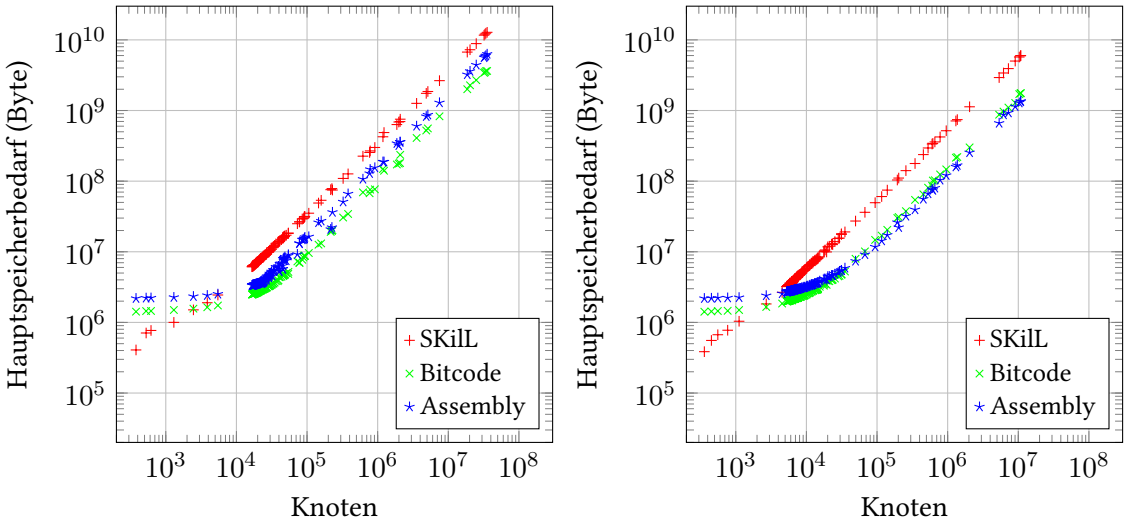
Um den Hauptspeicherbedarf bei der Konvertierung zu messen, wird das `valgrind massif` Werkzeug verwendet.[14] Die Option `-stacks=yes` wird genutzt, um auch Allokationen auf dem Stack zu berücksichtigen.

In Abbildung 5.6 ist das Ergebnis der Messungen zu sehen. Tabelle 5.8 zeigt die dazugehörige lineare Regression. Die Steigung und Konstante der Regressionsgerade ist in MB pro 10^6 Knoten bzw. MB angegeben. Wie erwartet hat das SKilL-Format den höchsten Speicherbedarf, da beide Darstellungen gleichzeitig im Speicher gehalten werden müssen. Für Testdaten mit bis zu ca. 1000 Knoten ist das SKilL-Format jedoch effizienter als das Assembly- und Bitcode-Format. Für das SKilL- und Assembly-Format kommen negative Konstante vor, die jedoch relativ nahe bei Null sind.

5.4.3 Zusammenfassung

Die Dateigröße ist beim SKilL-Format geringer als beim Assembly-Format, aber höher als beim Bitcode-Format. Dies entspricht der Erwartung, da im SKilL-Format manche Daten redundant sind. Der benötigte Hauptspeicher für die Konvertierung ist beim SKilL-Format am höchsten, da der SKilL-Graph zusätzlich im Speicher gehalten werden muss.

[14] Siehe http://valgrind.org/docs/manual/ms-manual.html



(a) Debug-Testdaten **(b)** Release-Testdaten

Abbildung 5.6: Hauptspeicherbedarf in Abhängigkeit von der Knotenzahl des SKilL-Graphen

	SKilL	Bitcode	Assembly
10^6 Knoten	361,971***	105,954***	175,668***
	(0,285)	(0,236)	(0,129)
Konstante	−2,936	0,542	−1,931**
	(1,863)	(1,540)	(0,843)
Beobachtungen	142	142	142
R^2	1,000	0,999	1,000
F-Statistik (df = 1; 140)	1.610.924,000***	201.988,800***	1.850.521,000***
10^6 Knoten	551,771***	158,345***	125,710***
	(0,359)	(0,710)	(0,370)
Konstante	−0,569	0,295	1,825**
	(0,708)	(1,400)	(0,731)
Beobachtungen	142	142	142
R^2	1,000	0,997	0,999
F-Statistik (df = 1; 140)	2.363.016,000***	49.791,440***	115.252,200***
Hinweis:			*p<0,1; **p<0,05; ***p<0,01

Tabelle 5.8: Lineare Regression über Hauptspeicherbedarf in Abhängigkeit der Knotenzahl (oben Debug, unten Release). Konstante in MB und Steigung in MB pro 10^6 Knoten.

6 Zusammenfassung

In dieser Arbeit wurde eine SKilL-basierte Darstellung für LLVM-IR entwickelt. Die zu Beginn gesetzten Ziele wurden alle erreicht.

Es wurden Werkzeuge zur Konvertierung der SKilL-basierten Darstellung in beide Richtungen implementiert. Existierende LLVM-Werkzeuge wurden erfolgreich an die SKilL-basierte Darstellung angebunden. Zusätzlich wurden noch Werkzeuge geschrieben, die direkt auf der SKilL-basierten Darstellung arbeiten. Hier hat sich gezeigt, dass dies zumindest für einfache Analysen ohne großen Aufwand möglich ist.

Die Abdeckung von LLVM-IR durch die SKilL-basierte Darstellung wurde ausführlich getestet. Sie schafft für einen repräsentativen Datensatz erfolgreich einen Round-Trip zum Assembly-Format. Außerdem werden viele der Tests aus den LLVM-Bibliotheken mit Erfolg bestanden. Die Unterstützung von zyklischen Abhängigkeiten ohne ϕ-Knoten lässt sich bei Bedarf in zukünftigen Versionen einbauen.

Die Dateigröße der verschiedenen Serialisierungsformate und die Laufzeit der Werkzeuge wurden für einen repräsentativen Datensatz untersucht. Die Laufzeit der SKilL-Werkzeuge liegt in der gleichen Größenordnung wie die Laufzeit der LLVM-Werkzeuge. Die serialisierten Dateien sind größer als beim Bitcode-Format, aber kleiner als beim Assembly-Format. Der Hauptspeicherbedarf ist bei der Konvertierung höher als beim Assembly- und Bitcode-Format, was zu erwarten war, weil der SKilL-Graph zusätzlich im Speicher gehalten werden muss.

Die SKilL-basierte Darstellung kann also verwendet werden, um in verschiedenen Programmiersprachen mit LLVM-IR zu arbeiten. Um die Nutzbarkeit für komplexere Werkzeuge zu bestimmen, sind jedoch noch weitere Tests notwendig.

Literaturverzeichnis

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006 (zitiert auf S. 11, 13).

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck. „An Efficient Method of Computing Static Single Assignment Form“. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, S. 25–35. DOI: 10.1145/75277.75280 (zitiert auf S. 14).

[EKP+99] T. Eisenbarth, R. Koschke, E. Plödereder, J.-F. Girard, M. Würthner. „Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen“. In: *1. Workshop Software-Reengineering* (Mai 1999) (zitiert auf S. 8).

[Fel17a] T. Felden. „Änderungstolerante Serialisierung großer Datensätze für mehrsprachige Programmanalysen“. Diss. 2017. DOI: 10.18419/opus-9661 (zitiert auf S. 7, 16, 18, 19, 29, 54).

[Fel17b] T. Felden. *The SKilL Language V1.0*. Techn. Ber. Universität Stuttgart, 2017 (zitiert auf S. 16, 17, 25, 30, 34).

[FW16] T. Felden, M. Wittiger. „Migrating Bauhaus from IML to SKilL“. In: *Softwaretechnik-Trends* 36.2 (2016) (zitiert auf S. 8).

[Har14] F. Harth. *Plattform- und sprachunabhängige Serialisierung mit SKilL*. Diplomarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. 2014. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3665&engl=0 (zitiert auf S. 17).

[Har16] R. Harth. *Anbindung von SKilL an Haskell*. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. 2016 (zitiert auf S. 17).

[Har84] W. A. Harrison. „Applying Mccabe's complexity measure to multiple-exit programs“. In: *Software: Practice and Experience* 14.10 (Okt. 1984), S. 1004–1007. DOI: 10.1002/spe.4380141009 (zitiert auf S. 38).

[LA04] C. Lattner, V. Adve. „LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation“. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, März 2004, S. 75–86. DOI: 10.1109/CGO.2004.1281665 (zitiert auf S. 7, 13, 14).

[LLM+18] B. Liblit, D. Liew, I. A. Mason, T. Ravitch, B. Dutertre. *Whole Program LLVM*. 2018. URL: https://github.com/travitch/whole-program-llvm (zitiert auf S. 18, 19).

[LLVa] LLVM. *Developer Policy*. URL: https://releases.llvm.org/5.0.1/docs/DeveloperPolicy.html (zitiert auf S. 9).

[LLVb] LLVM. *Documentation*. URL: https://releases.llvm.org/5.0.0/llvm_doxygen-5.0.0.tar.xz (zitiert auf S. 25, 27, 36).

[LLVc] LLVM. *Frequently Asked Questions*. URL: https://releases.llvm.org/5.0.1/docs/FAQ.html (zitiert auf S. 22).

[LLVd] LLVM. *Language Reference Manual*. URL: https://releases.llvm.org/5.0.1/docs/LangRef.html (zitiert auf S. 7, 11–14, 22, 27).

[LLVe] LLVM. *MemorySSA*. URL: https://releases.llvm.org/5.0.1/docs/MemorySSA.html (zitiert auf S. 25).

[LLVf] LLVM. *Programmer's Manual*. URL: https://releases.llvm.org/5.0.1/docs/ProgrammersManual.html (zitiert auf S. 15, 16, 23, 24, 29).

[LLVg] LLVM. *Tutorial*. URL: https://releases.llvm.org/5.0.1/docs/tutorial/ (zitiert auf S. 40).

[LLVh] LLVM. *Writing an LLVM Pass*. URL: https://releases.llvm.org/5.0.1/docs/WritingAnLLVMPass.html (zitiert auf S. 19).

[McC76] T. J. McCabe. „A Complexity Measure". In: *IEEE Trans. Softw. Eng.* 2.4 (1976), S. 308–320. DOI: 10.1109/TSE.1976.233837 (zitiert auf S. 38, 39).

[Plö16] E. Plödereder. *Programmanalysen und Compilerbau*. Vorlesung: Universität Stuttgart. 2016 (zitiert auf S. 11).

[Prz14] D. Przytarski. *Performance-Evaluation einer sprach- und plattformunabhängigen Serialisierungssprache*. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. 2014. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-0106&engl=0 (zitiert auf S. 17).

[Ung14] W. Ungur. *Nutzbarkeitsevaluation einer sprach- und plattformunabhängigen Serialisierungssprache*. Diplomarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. 2014. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3603&engl=0 (zitiert auf S. 17).

Alle URLs in diesem Dokument wurden zuletzt am 02. 05. 2018 geprüft.

Abkürzungsverzeichnis

API application programming interface.

ASCII American Standard Code for Information Interchange.

AST abstrakter Syntaxbaum (englisch *abstract syntax tree*).

CFG Kontrollflussgraph (english *control flow graph*).

DFS depth-first search.

ELF Executable and Linkable Format.

IR Zwischendarstellung (english *intermediate representation*).

JIT just-in-time.

JVM Java Virtual Machine.

LSB least significant bit.

MB Megabyte.

PR problem report.

RAUW replace all uses with.

RISC reduced instruction set computer.

SKiIL Serialization Killer Language.

SLOC source lines of code.

SSA static single assignment.

SWIG Simplified Wrapper and Interface Generator.

WLLVM Whole Program LLVM.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift