Evaluierung und Implementierung eines LLVM-Backends für die erweiterbare Programmiersprache MOSTflexiPL

Bachelorarbeit

Autor:
Lukas Pietzschmann (76010)

Erstgutachter:
Prof. Dr. habil. Christian Heinlein

Zweitgutachter:
Prof. Dr. Winfried Bantel

15. August 2022

Hochschule Aalen
Studiengang Informatik - Software Engineering

Eidesstattliche Erklärung

Ich versichere, dass ich diese Ausarbeitung selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung (Zitat) kenntlich gemacht. Das Gleiche gilt für beigefügte Skizzen und Darstellungen.

Hüttlingen, den 15. August 2022

Ort, Datum

Lukas Pietzschmann

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abstract

Betrachtet man die jährliche StackOverflow Developer Survey stellt man fest, dass kompilierte Programmiersprachen in der Kategorie „Most Loved“ einen immer höheren Anteil annehmen und in den letzten vier Ausgaben der Umfrage stets sechs bis sieben der Top zehn beliebtesten Sprachen kompilierte waren. Auch große Firmen wie Apple und Google scheinen mit Swift (seit 2014) beziehungsweise Kotlin (seit 2017) zu Maschinencode übersetzende Sprachen zu präferieren. Während interpretierte Sprachen sicher nie ihre Anwendungsgebiete verlieren werden, ist der reine Performance-Vorteil kompilierten Sprachen schwer abzusprechen.
Diese Arbeit setzt sich damit auseinander, ob auch für die extrem flexible Programmiersprache MOSTflexiPL das Folgen dieses Trends zur Maschinencodeerzeugung umsetzbar ist. Dazu werden im ersten Teil der Arbeit verschiedene Ansätze der Codeerzeugung erläutert und miteinander verglichen. Der Hauptteil der Arbeit wird dann anschließend eine „Proof of Concept“-Implementierung unter der Verwendung des modernen Compiler-Framework LLVM präsentieren.

1 Einleitung

1.1 Motivation

Das LLVM-Projekt stellt eine Infrastruktur zur Entwicklung von sowohl traditionellen Compilern als auch von compilerbasierten Tools wie Just-In-Time Compilern, Profilern, Speicher-Sandboxing-Tools, statische Analyse-Tools und viele mehr zur Verfügung. Und auch wenn LLVM ursprünglich an der Universität Illinois entwickelt wurde, ist es schon lange kein kleines Forschungsprojekt mehr. C, C++, Haskell, Swift, Rust und diverse weitere große Programmiersprachen besitzen Compiler, die auf dieser Bibliothek aufbauen.
Diese Arbeit soll zeigen, dass (oder ob) LLVM auch für kleinere Forschungssprachen wie MOST-flexiPL eine gute Wahl ist. Dabei besteht die Herausforderung dieser Arbeit im Umgang mit der extrem flexiblen Natur dieser Programmiersprache. Somit müssen Möglichkeiten entwickelt werden, extrem abstrahierende und flexible Strukturen der Quellsprache in extrem statische und unflexible Strukturen der Zielsprache zu überzuführen.

1.2 Abgrenzung

Die hier vorgestellte Implementierung wird keinen Wert auf das Erzeugen eines möglichst effizienten Programms legen. Auch wenn immer wieder kleinere Gedanken zur Optimierung des generierten Codes eingestreut werden, ist das Hauptziel hier klar das Entwickeln eines robusten, gut funktionierenden Backends. Somit ist diese Implementierung vor allem ein Proof of Concept. Außerdem soll der generierte Code so eigenständig wie möglich sein. Es ist nicht das Ziel der Arbeit bereits übersetzte Bibliotheken wie libC zur Laufzeit einzubinden, um Komplexität zu vermeiden. Die Komplexität soll viel mehr genutzt werden, um neue interessante Ideen zur Lösung der diversen aufkommenden Herausforderungen hervorzubringen.

2 Möglichkeiten zur Erzeugung von Maschinencode

Auch wenn sich diese Arbeit primär mit der Verwendung der LLVM-Bibliothek auseinandersetzt, ist diese sicher nicht die einzige Möglichkeit nativen Maschinencode zu erzeugen. Grob kann man die meisten Möglichkeiten in drei verschiedene Klassen einteilen:

- Direktes Erzeugen von Maschinencode
- Indirektes Erzeugen von Maschinencode mithilfe einer Hochsprache
- Indirektes Erzeugen von Maschinencode mithilfe einer Zwischensprache

2.1 C-Backend

Ein erster möglicher Ansatz ist es, einfach eine beliebige Hochsprache zu verwenden, die wiederum zu nativem Code übersetzt werden kann. Während es hier natürlich etliche Möglichkeiten wie beispielsweise Haskell, Rust oder Go gibt, ist die Programmiersprache C dabei wohl die beliebteste. Aufgrund ihrer geringen Abstraktionen und Nähe zur Hardware, ist C flexibel genug, um wohl die meisten, wenn nicht sogar alle Konzepte höherer Programmiersprachen abzubilden. Ein weiterer Vorteil ist die breite Unterstützung an Zielplattformen. Alleine die GNU-Compiler-Collection unterstützt 50 verschiedene Architekturen, wobei die wichtigsten und bekanntesten wie arm, x86 oder riscv selbstverständlich vertreten sind [GCC]. Ein allerdings nicht zu vernachlässigender Nachteil ist die relativ lange Übersetzungszeit (für weitere Details siehe Unterabschnitt 7.2 - Übersetzungszeit - C vs. LLVM-IR). Diese führt, wie David A. Terei in *An LLVM Backend for GHC* schreibt, nicht einmal zu besonders gut optimiertem und performantem Assemblycode. Als Schuldigen sieht Terei hier allerdings weniger den GNU C-Compiler als vielmehr die inhärent unübliche Struktur des generierten C-Codes, auf die Compiler wohl nicht spezialisiert sind [TC10, S. 2].
Um diese beiden Nachteile jeweils zu beseitigen, existieren Projekte wie der Tiny C-Compiler, der zwar die Übersetzungszeit drastisch verringert, aber dadurch wohl noch unperformanteren Assemblycode für wesentlich weniger Architekturen erzeugt. Ein tatsächlich sehr vielversprechender Ansatz ist die Programmiersprache C--. Diese wurde speziell als C ähnliche Zwischensprache entwickelt, die durch das Weglassen oder Verändern einiger C-Features sowohl das Übersetzten zu nativem Code, als auch das Erzeugen selbst einfacher gestalten soll. Allerdings ist C-- eine nicht sonderlich weit verbreitete Sprache und somit ist die tatsächliche Weiterentwicklung und Optimierung leider nicht garantiert [TC10, S. 1].

2.2 Assembly-Codegenerator

Ein weiterer Ansatz, die Nachteile der C-Code-Generierung zu umgehen, könnte das direkte Erzeugen von Maschinencode sein. Da der Entwickler hier die volle Kontrolle über den erzeugten Assemblercode hat, kann dieser direkt für die jeweilige Quellsprache optimiert werden. Dies hat zur Folge, dass ein händisch geschriebener Assembly-Codegenerator tendenziell effizienteren Code produzieren kann, als ein C-Compiler es für nicht idiomatischen C-Code könnte [TC10, S. 2]. Allerdings ist es wesentlich mehr Aufwand einen Codegenerator für nativen Maschinencode zu pflegen beziehungsweise zu verwalten, da es hierzu detailliertes Wissen über die Zielarchitektur benötigt. Somit kann man zwar in kürzerer Übersetzungszeit effizienteren Code erzeugen, hat aber nicht den Vorteil von Haus aus diverse Architekturen unterstützen zu können, sondern muss für jedes neue Zielsystem einen Großteil des Codes umschreiben.

2.3 LLVM-IR-Backend

Ein Backend, das zu einer dedizierten Zwischensprache übersetzt, kann nun die Vorteile von sowohl dem C-Backend als auch des Assebly-Codegenerators verbinden, ohne zwingend auch deren Nachteile zu übernehmen. Wie schon in Unterabschnitt 2.1 - C-Backend angerissen, gibt es hierfür zwar vereinfachte Sprachen wie C--, die aber oft den großen Nachteil haben Nieschensprachen zu sein.
Die Zwischensprache des LLVM Projekts hat diesen Nachteil aufgrund der hohen Nutzerzahl[LLVb] nicht! Verwendet man diese Zwischensprache, klinkt man sich gleichzeitig auch in die komplette Toolchain von Linker über Debugger bis hin zu statischen Analysetools ein. Wobei ein LLVM verwendendes Projekt somit direkt von jeder Optimierung an einer beliebigen Stelle in dieser Toolchain profitiert.
LLVM bietet nicht nur schnelle Übersetzungszeiten, relativ simple Codeerzeugung, auf die später in der Arbeit noch ausführlich eingegangen wird, und Unterstützung für diverse Architekturen, sondern auch eine große Community, die dem Projekt seine Langlebigkeit fast schon garantiert. Ob es aber nicht nur theoretisch, sondern auch praktisch sinnvoll ist eine extrem flexible Sprache wie MOSTflexiPL nach LLVM-IR zu übersetzen, wird sich vor allem noch in Abschnitt 4 - Implementierung zeigen.

3 Grundlagen

3.1 Eigenschaften der LLVM-IR

Die LLVM-IR (**I**ntermediate **R**epresentation) ist sozusagen das Herz des kompletten LLVM Projekts:

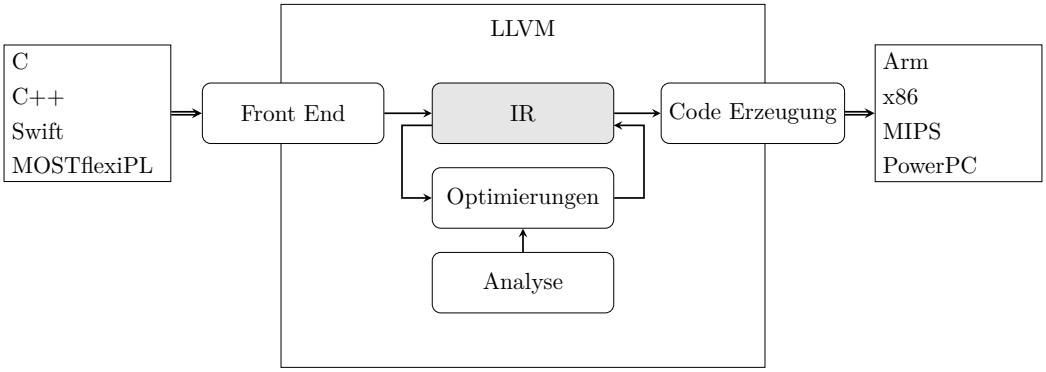


Abbildung 1: LLVM Infrastruktur
Quelle: C. Lattner

Wie Abbildung 1 - LLVM Infrastruktur zeigt, werden alle Optimierungen oder Transformationen des Codes auf der LLVM Zwischensprache ausgeführt. Dadurch, dass LLVM vom Start (der Generierung der IR) bis zum Ende (der Erzeugung des Maschinen-Codes) eingebunden ist, kann man eine „lebenslange Analyse und Transformation beliebiger Programme“[LA04, S. 2] erreichen. Diese Fähigkeit ist eine der großen Errungenschaften des LLVM Projekts.

Im ersten Schritt, den diese Arbeit primär behandeln wird, erzeugt jedes Backend[1] genau diese Zwischensprache. Dabei können drei äquivalente Formen erzeugt werden [LLVa, Introduction]:

- LLVM-Assembly
- LLVM-Bitcode
- In-Memory Repräsentation

LLVM Assembly ist die wohl leserlichste Variante. Immer wenn hier Beispiele von IR-Code gezeigt werden, ist dieser in der Assembly-Form abgebildet. Sie ähnelt sehr normalem Assembler-Code, enthält zusätzlich allerdings auch statische Typ-, Kontroll- und Datenfluss-Informationen. Die Bitcode und in-Memory Abbildungen sind nicht dazu gedacht von Menschen gelesen zu

[1] Die Grenze zwischen Front- und Backend ist für diese Arbeit der abstrakte Syntrax-Baum. Ergo gehört die hier entwickelte Codegenerierung zum Backend des Compilers.

werden. Bitcode Dateien benötigen deutlich weniger Speicher und können unter anderem auch dadurch effizienter eingelesen werden. Sobald also zwei eigenständige Tools LLVM-IR untereinander austauschen müssen, sollten dazu eher Bitcode- als Assembly-Dateien verwendet werden. Die in-Memory Repräsentation ist die wichtigste für diese Arbeit beziehungsweise wohl grundsätzlich für ein Compiler-Backend. Sie wird während des Compile-Vorgangs erzeugt und kann beide vorher besprochenen Darstellungen erzeugen. Auf ihr laufen außerdem auch alle selbst entwickelten oder in LLVM bereits enthaltenen Optimierungen. Dadurch muss nicht bei jeder Transformation eine Datei geschrieben und eingelesen werden, sondern es können alle Datenstrukturen im Speicher wieder beziehungsweise weiter verwendet werden.

Wie sieht nun aber die Syntax der LLVM-IR konkret aus? Um diese Frage zu klären, wurde das folgende, zugegebenermaßen etwas künstlich wirkende, MOSTflexiPL-Programm übersetzt:

Listing 1: Beispiel MOSTflexiPL-Programm

```
1 i: int?;
2 i =! read;
3 print if ?i > 10 then
4             i =! 10
5       else
6             ?i - 1
7       end
```

Auch wenn dieses Programm wohl ohne bereits vorhandene MOSTflexiPL Kenntnisse recht gut lesbar sein sollte, hier eine kurze Zusammenfassung: Es wird eine Variable `i` durch den Benutzer über die Konsole initialisiert und anschließend mit dem Literal `10` verglichen. Ist `i` echt größer als `10`, wird `i` auf `10` gesetzt und der `if`-Ausdruck liefert `10` als sein Ergebnis. Andernfalls wird `i` um `1` dekrementiert als Ergebnis zurückgegeben. Zum Schluss wird das Resultat des `if`-Ausdrucks via `print` auf der Konsole ausgegeben.

Aus diesem Programm wurde dann folgender IR-Code generiert[2]:

Listing 2: Beispiel LLVM-IR-Code

```
1  @scanf_format = global [3 x i8] c"%d\00
2  @prinf_format = global [5 x i8] c"%*d\0A\00
3
4  declare extern ccc i32 @scanf(i8* %0, ...)
5  declare extern ccc i32 @printf(i8* %0, ...)
6
7  define linkonce fastcc i32 @main() {
8    %i_ptr = alloca i32, align 4
9    store i32 0, i32* %i_ptr, align 4
10   %0 = alloca i32, align 4
11   %1 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]*
         @scanf_format, i32 0, i32 0), i32* %0)
12   %2 = load i32, i32* %0, align 4
13   store i32 %2, i32* %i_ptr, align 4
14   br label %if
15 if:
16   %i = load i32, i32* %i_ptr, align 4
17   %3 = icmp sgt i32 %i, 10
18   br i1 %4, label %then, label %else
19 then:
20   store i32 10, i32* %i_ptr, align 4
21   br label %merge
22 else:
23   %i1 = load i32, i32* %i_ptr, align 4
24   %4 = sub i32 %i1, 1
25   br label %merge
26 merge:
27   %5 = phi i32 [ 10, %then ], [ %4, %else ]
28   %6 call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]*
         @prinf_format, i32 0, i32 0), i32 0, i32 %5)
29   ret i32 0
30 }
```

Alle jeweils zusammen gehörenden Zeilen wurden hier in derselben Farbe dargestellt.

Auf den ersten Blick ist gleich die bereits angesprochene Ähnlichkeit zu echtem Maschinen-Code zu erkennen. Einige Befehle, wie beispielsweise `sub` in Zeile 24 oder `br` in Zeile 14 existieren so oder zumindest so ähnlich auch in diversen Prozessor-Befehlssätzen. Diese Nähe zur Maschinensprache sorgt vor allem dafür, dass sich sehr viele (oder idealerweise alle) Konzepte einer Hochsprache sauber auf die IR abbilden lassen [LLVa, Introduction].

[2]Tatsächlich wurden ein paar Details entfernt, um den Code hier etwas verständlicher zu halten.

3.1.1 Funktionen

Der erzeugte LLVM-IR-Code hat aber neben den genannten Parallelen zu echtem Maschinen-Code auch eine deutliche Parallele zu vielen Hochsprachen: die Main-Funktion.
Dem Kopf dieser Funktion (`define linkeonce ccc i32 @main`) lassen sich drei Eigenschaften entnehmen, von denen tatsächlich nur eine zwingen anzugeben ist. `i32` stellt den Rückgabetypen der Funktion dar. Dieser muss bei jeder Funktion explizit gesetzt werden, da, wie zu Beginn dieses Kapitels bereits beschrieben, die LLVM-IR eine typisierte Sprache ist. `i32` ist dabei ein 32 Bit breiter ganzzahliger Typ, der allerdings keine Information darüber enthält, ob ein Wert vorzeichenbehaftet oder vorzeichenlos zu interpretieren ist. Befehle, bei denen dies einen Unterschied macht (beispielsweise `div`), haben in der Regel zwei Varianten, die jeweils mit „U“ und „S“ gepräfixt sind (`udiv` beziehungsweise `sdiv`).
Eine Stelle weiter links in der Signatur befindet sich die Calling-Convention, mit der die Funktion aufgerufen werden muss. `fastcc` entspricht dabei der Fast-Calling-Convention. Diese versucht einen Funktionsaufruf so performant wie möglich zu machen, ohne dabei einem bestimmten ABI (**A**pplication **B**inary **I**nterface) entsprechen zu müssen. Dies wird beispielsweise durch das Übergeben von so vielen Parametern wie mögliche via Register anstelle des Stapels erreicht. Ein Nachteil dieser Konvention ist das Fehlen der Unterstützung variadischer Funktionen. Da es aber an keiner Stelle in dieser Arbeit einen Vorteil hätte variadische Funktionen zu generieren, wird diese Calling-Convention hier als Standard für alle Funktionen verwendet, die nicht explizit eine andere verlangen. Ein Beispiel für eine solche Funktion wird später in diesem Kapitel noch vorkommen.
Noch eine Stelle weiter links befindet sich die Art, wie die Funktion später gelinked werden soll. Die Main-Funktion ist hier die einzige Funktion, die mit `linkonce` gelinked wird. Da das hier beschriebene Backend immer nur ein Modul generiert, von dem keine Funktionen nach außen verfügbar gemacht werden müssen, werden alle anderen Funktionen werden `private` gelinked.

Die Main-Funktion liefert hier ab Zeile acht im Kompilat auch direkt eine Implementierung nach. Bei dieser fällt auf, dass sie explizit in einzelne basic Blocks aufteilt ist. Ein basic Block ist eine stets sequenziell und immer komplett ausgeführte Abfolge an Befehlen. Ausschließlich der letzte Befehl eines basic Blocks darf beziehungsweise muss ein Sprungbefehl sein. Daraus folgt, dass auch eine `void` zurückgebende Funktion pro forma mit einer `ret`-Instruktion terminieren muss. Alle Blöcke formen zusammen einen Kontroll-Fluss-Graphen, der zumindest in diesem Beispiel relativ direkt die Struktur des ursprünglichen Programms abbildet. Lediglich der `merge`-Block am Ende der Verzweigung hat keine direkte Repräsentation im Quell-Programm. Wozu dieser Block benötigt wird, wird später in Unterunterabschnitt 3.1.3 - Kontrollstrukturen und Phi-Instruktionen noch aufgegriffen.

Funktionen müssen allerdings nicht immer direkt implementiert werden. Wie C, C++ und viele andere Sprachen unterstützt auch die IR Prototypen. Die in Zeile zwei und drei im Quell-Programm verwendeten Operatoren `read` und `print` werden so beispielsweise nicht direkt implementiert, sondern in den Zeilen vier und fünf im Kompilat nur deklariert. Dabei wird dann anstelle von `define` das Schlüsselwort `declare` verwendet. Abseits dieses Schlüsselwortes enthält die Signatur allerdings die exakt selben Bestandteile wie die Main-Funktion. Wie die Namen `scanf` und `printf` bereits vermuten lassen, handelt es sich dabei um die Funktionen der C-Bibliothek. Dieses Backend liefert also selbst keine Implementierung für diese beide Operatoren.
Mit diesen beiden Funktionen sieht man nun auch ein Beispiel dafür, wann nicht die Fast-Calling-Convention verwendet werden darf. Um zum Schluss die generierte Objekt-Datei gegen libC linken zu können, müssen diese beiden Funktionen ergo auch der C-Calling-Convention folgen, weswegen hier dann auch explizit `ccc` als Konvention angegeben wird.

3.1.2 Globale und lokale Variablen

Globale Variablen in LLVM sind grundsätzlich nichts anderes als eine benannte Region im Speicher. Anders als für normale Zeiger wird der Speicher für globale Variablen allerdings schon zur Übersetzungszeit alloziert[3][LLVa, Global Variables]. Wie in den ersten beiden Zeilen des IR-Codes zu sehen, werden globale Werte mit einem `@`-Symbol kenntlich gemacht. Ansonsten verhalten sich diese Variablen praktisch äquivalent zu Zeigern.

Stellt man sich vor, der Code-Ausschnitt aus Listing 1 - Beispiel MOSTflexiPL-Programm sei eingebettet in einen umschließenden Scope (alle verwendeten Variablen sind also keine globalen), findet man zwei verschiedene Arten von lokalen „Dingen“:

- Temporäre Werte
- Echte Variablen

MOSTflexiPL macht es dem Benutzer einfach, indem einem Ausdruck allgemeine Ausdrücke als Parameter übergeben werden können. Die LLVM-IR, wie wohl auch alle anderen Assemblersprachen, erlaubt das allerdings nicht! Hier müssen Zwischenergebnisse erst ausgerechnet werden und dann als einfache Werte übergeben werden. Der simple Ausdruck `?i - 1` (siehe Zeile sechs des MOSTflexiPL-Beispiels) besteht intern also eigentlich aus drei einzelnen Schritten:

[3] Ähnlich, wie in C der Speicher für `int arr[5]` im Gegensatz zu `int* arr = (int*)malloc(5 * sizeof(int))` auch schon zur Übersetzungszeit zugewiesen wird.

1. Wert der Variablen `i` bestimmen
2. Den ermittelten Wert dem Minus-Operator übergeben
3. Den nun verringerten Wert als Ergebnis zurückliefern

Der im Kompilat türkis eingefärbte Ausschnitt zwischen Zeile 24 und 25 stellt die ersten beiden Schritte dar.

Temporäre Werte

Unschwer erkennbar ist, dass Schritt zwei direkt auf Zeile 25 abgebildet wird. Hier wird der Wert von `i` mithilfe der `sub`-Instruktion um `1` verringert und danach in `%4` geschrieben. `%4` ist ein Register! Diese Register korrespondieren allerdings in keinster Weise zu echten Registern des Hardware-Registersatzes. LLVM verwendet stattdessen sogenannte virtuelle Register. Von diesen stehen dem Entwickler beliebig viele zur Verfügung. Erst wenn LLVM Maschinencode erzeugen wird, werden diese virtuellen Register auf echte Register oder in den Speicher abgebildet. Um die beliebigen Namen dieser Register von globalen Variablen unterscheiden zu können, werden diese, beziehungsweise auch basic Blocks und alle weiteren lokalen „Dingen“, mit dem Präfix `%` versehen [SP15, S. 5].
Eine grundlegende Fähigkeit haben LLVM-Register allerdings nicht:

Listing 3: SSA-Form verletzender LLVM-IR-Code

```
1 %i = add i32 4, 2
2 %i = add i32 5, 3  ; <- Error
```

SSA-Form: SSA steht für **s**tatic **s**ingle **a**ssignment, was voraussetzt, dass ein Register statisch nur einmalig bei der Ausführung der beschreibenden Instruktion beschrieben wird. Da in obigem Code-Beispiel das Register `%i` allerdings noch ein zweites Mal beschrieben wird, folgt dieses Beispiel nicht der SSA-Form und ist somit auch kein gültiges LLVM-IR-Programm. Achtung: SSA bedeutet nicht, dass ein Register auch dynamisch nur einmalig beschrieben werden darf. So ist es vollkommen erlaubt beispielsweise während der Laufzeit einer Schleife ein Register mit derselben Instruktion mehrmals zu beschreiben, da dies statisch gesehen weiterhin nur einmalig passiert!

SSA ist auf der Programmiersprachen-Ebene also in etwa vergleichbar mit register-renaming auf Ebene des Prozessors. So werden auch hier eventuell künstliche Namensabhängigkeiten direkt vermieden, was Optimierungen wie sparse conditional constant propagation oder value numbering wesentlich vereinfacht[Muc97, S. 372] oder andere Optimierungen erst möglich macht.
Wenn Register durch die Voraussetzung dieser Form allerdings nur einmal beschreibbar sind, stellt sich die Frage, wie genau veränderbare Variablen realisiert werden.

Veränderbare lokale Variablen

Wenn in Zeile 24 für Schritt eins der Wert der Variablen `i` bestimmt werden soll, wird dies nicht unmittelbar über das Lesen eines Registers namens `i` getan, sondern indirekt über einen `load`-Befehl der Adresse aus `i_ptr`. Veränderbare Variablen werden also nicht unmittelbar in Registern gespeichert, sondern im Speicher abgelegt. Wie genau wird nun aber der Speicher für `i` beschafft? Beziehungsweise wie genau wird Speicher auf dem Stack beschafft? Die Antwort findet man an der rot markierten Stelle der Deklaration. Hier befindet sich eine sehr an C's `void *malloc(size_t size)` erinnernde Instruktion. `alloca` alloziert aber eben nicht auf dem Heap, sondern reserviert Speicher auf dem Stack der aktuell ausgeführten Funktion.
Zusammengefasst lässt sich also sagen, dass Zwischenergebnisse immer in Registern stehen, während echte Variablen, wie wohl auch erwartet, auf dem Stack abgelegt werden.

3.1.3 Kontrollstrukturen und Phi-Instruktionen

Um den kleinen Trick, den dieses Kapitel vorstellt, zu motivieren, werden im Folgenden nur die letzten fünf Zeilen aus Listing 1 - Beispiel MOSTflexiPL-Programm betrachtet. Eine naive Übersetzung dieses Ausschnittes zu LLVM IR könnte folgendermaßen aussehen:

Listing 4: Naive Umsetzung von Kontrollstrukturen

```
1  entry:
2    %retval = alloca i32, align 4            ; <Entfernen>
3    %0 = icmp sgt i32 %i, 10
4    br i1 %0, label %then, label %else
5  then:
6    store i32 %i, i32* %retval, align 4      ; %retval = i32 %i
7    br label %end
8  else:
9    store i32 10, i32* %retval, align 4      ; %retval = i32 10
10   br label %end
11 merge:
12   %1 = load i32, i32* %retval, align 4     ; <Entfernen>
```

In Zeile zwei wird eine lokale Variable auf dem Stack platziert, welche in Zeile sechs beziehungsweise neun ihren jeweiligen Wert erhält und ganz am Ende in `%i` für weitere Verwendungen zur Verfügung steht. Was man intuitiv wohl lieber schreiben würde, ist in die betreffenden Zeilen jeweils als Kommentar daneben eingetragen. Die `store`-Instruktion in Zeile neun sollte also zu einer einfachen Zuweisung in ein Register werden. Aufgrund der SSA-Voraussetzung kann diese Idee allerdings direkt verworfen werden, da das Register `%retval` mehrmals von unterschiedlichen Instruktionen beschrieben wird. Trotzdem wäre es aber erstrebenswert einen Mittelweg zwischen diesen beiden Varianten zu finden, da Zugriffe auf Register wohl immer effizienter sind als ein Zugriff auf den Speicher über `load` und `store`.
Die Lösung ist die phi-Instruktion, die in Listing 2 - Beispiel LLVM-IR-Code in Zeile 28 auch tatsächlich verwendet wird.

Diese Instruktion kann in Abhängigkeit des zuletzt ausgeführten basic Blocks genau einen Wert auswählen.

```
1 %5 = phi i32 [ 10, %then ], [ %4, %else ]
```

Hier wird also der Wert `10` gewählt, wenn aus dem basic Block `%then` an diese Stelle gesprungen wurde beziehungsweise der Wert aus dem Register `%4`, wenn `%else` der zuletzt ausgeführte Block war. Grundsätzlich können dabei beliebig viele Wert-Block-Paare angegeben werden, wobei zwei aber wohl der häufigste Fall ist (zumindest in dieser Arbeit).
Diese Instruktion kann neben einfachen Fallunterscheidungen beispielsweise auch für Schleifen verwendet werden. Im Folgenden wird eine `for`-Schleife über `min` bis `max` dargestellt:

Listing 5: Schleifen in LLVM-IR

```
1  ; for(int i = min; i < max; i++) ...
2  entry:
3    ...
4    %entrycond = icmp sgt i32 %max, %min
5    br i1 %entrycond, label %for.body, label %for.end
6  for.body:
7    %i = phi i32 [ %inc, %for.body ], [ %min, %entry ]
8    ...
9    %inc = add i32 %i, 1
10   %exitcond = icmp eq i32 %inc, %argc
11   br i1 %exitcond, label %for.end, label %for.body
12 for.end:
13   ...
```

Hier wird in Zeile sieben der für die aktuelle Iteration korrekte Wert von `i` bestimmt. In der ersten Iteration (es wurde von `%entry` nach `%for.body` gesprungen), wird der Startwert `%min` gewählt. Für alle weiteren Iterationen (es wurde von Ende von `%for.body` zum Anfang von `%for.body` gesprungen) wird der um `1` erhöhte letzte Wert von `%i` gewählt.
Dieses Beispiel veranschaulicht außerdem noch einmal, was bereits im Absatz SSA-Form erwähnt wurde. Die Register `%i`, `%inc` und `%exitcond` werden zwar zur Laufzeit mehrmals beschrieben (zumindest falls $\%max >= \%min + 2$ gilt), relevant ist aber nur, dass sozusagen beim Lesen des Programmcodes die Register nur einmal auf der linken Seite vorkommen. Daraus folgt, dass die SSA-Form hier eben nicht verletzt ist.
Die letzte interessante Beobachtung, die man an diesem Beispiel machen kann, ist, dass im Gegensatz zu den meisten Hochsprachen Register auch vor ihrer Deklaration schon verwendet werden können. Solange LLVM statisch verifizieren kann, dass `%inc` alle seine Verwendungen dominiert, die Deklaration also während der Laufzeit immer vor der ersten Verwendung stattfindet, ist auch das Verwenden von statisch noch nicht existenten Registern erlaubt.

3.2 MOSTflexiPLs AST

Der in diesem Abschnitt untersuchte AST (**A**bstract **S**yntax **T**ree) basiert auf folgendem MOSTflexiPL-Code:

```
1  print 1 + 2
```

Für dieses Programm spannt der Parser den folgenden Baum auf:

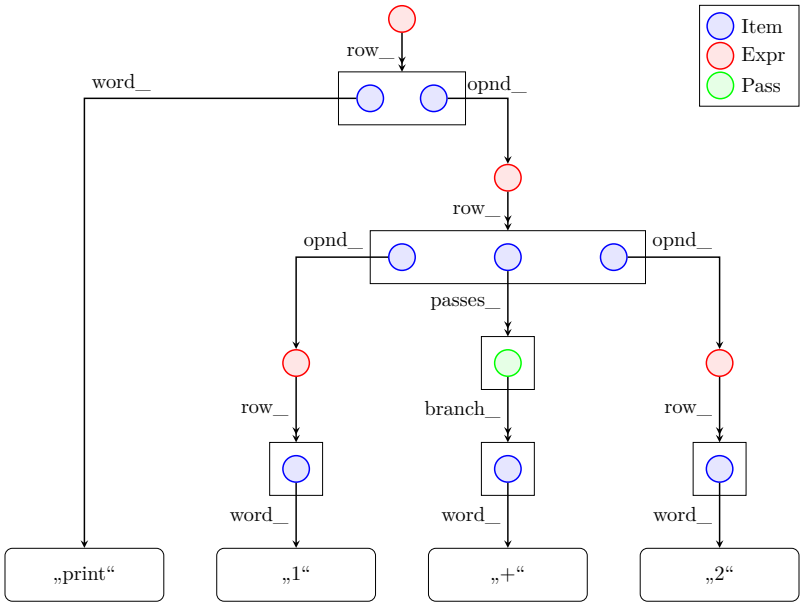
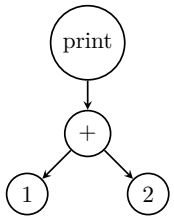


Abbildung 2: Abstrakter Syntaxbaum des Ausdrucks `print 1 + 2`



Auf den ersten Blick ist der AST wesentlich tiefer als man eventuell erwartet hätte. Der wohl simpelste Graph, der diesen Code abbilden könnte, ist hier zur rechten abgebildet. Dieser besteht ausschließlich aus den beiden Operatoren (`print` und `+`) und ihren Operanden (`+`-Operator, Literale `1` und `2`). Versucht man diesen Baum nun über Abbildung 2 - Abstrakter Syntaxbaum des Ausdrucks `print 1 + 2` zu legen, stellt man fest, dass diese minimale Form auch in dem vom MOSTflexiPL-Parser erzeugten AST enthalten ist. So repräsentieren die unteren drei rot gefärbten Knoten genau die drei Knoten aus dem Graphen rechts. Diese drei Knoten stellen Objekte des offenen Typs `Expr` dar. Da MOSTflexiPL ausschließlich aus Ausdrücken besteht, ergo keine Statements kennt, ist `Expr` auch der zentrale Typ des ASTs. Ein `Expr`-Objekt enthält eine ganze Reihe an Informationen, wie beispielsweise den angewandten Operator oder den Resultattyp des Ausdrucks. Die für dieses Backend allerdings wichtigste Information ist die Reihe (Attribut `row_`) des Ausdrucks. Sie enthält eine Liste aller Operanden eines Ausdrucks. Der Wurzel-Ausdruck des obigen Beispiels enthält genau zwei Operanden. Einerseits das Wort „print“ selbst und andererseits den `+`-Ausdruck, dessen Ergebnis ausgegeben werden soll. Äquivalent dazu besteht die Reihe des `+`-Ausdrucks indirekt aus den beiden Int-Literal-Ausdrücken, die die `1` und `2` produzieren, und wieder aus dem gelesenen „+“ selbst. Die Elemente einer Reihe sind alles Objekte des Typs `Item`. Ein `Item` besitzt neben den beiden bereits gesehenen Attributen `word_` für den gelesenen Text und `opnd_` für einen Operanden (mit Typ `Expr`), noch das Attribut `passes_`. Dieses bildet ein sehr einzigartiges Konzept der Programmiersprache MOSTflexiPL ab.

Klammern: Genauer wird auf dieses Konzept noch an entsprechenden Stellen in Abschnitt 4 - Implementierung eingegangen. Hier also erst einmal nur das grobe Konzept. Achtung: „normale“ gruppierende Klammern existieren natürlich auch, diese sind an dieser Stelle allerdings nicht gemeint! MOSTfexiPL unterscheidet zwischen drei Arten von Klammern:

Runde Klammer (Alternative):

> Diese Klammer bietet mehrere Möglichkeiten, aus denen bei einem Durchlauf ausgewählt werden kann. Es muss allerdings immer eine der Möglichkeiten gewählt werden.

Eckige Klammer (Option):

> Eine optionale Klammer entspricht grundsätzlich einer Alternative, allerdings kann diese Klammer auch nicht durchlaufen werden.

Geschweifte Klammer (Wiederholung):

> Wiederholungs-Klammern haben, wie auch die beiden vorigen, verschiedene Möglichkeiten, die nun jedoch auch mehrmals durchlaufen werden können. Mehrmals meint dabei auch nullmal.

Die Klammern `{ [ a | b ] ( c | d ) }` würden also beispielsweise die Eingabe `a c b d c` akzeptieren.

Um diese extrem flexiblen Klammerstrukturen im AST darstellen zu können, benötigt es das Attribut `passes_` des Typs `Item`. Hier werden alle Durchläufe durch eine Klammer abgespeichert. „Jeder solche Durchlauf (Typ Pass) enthält die Position der jeweils durchlaufenen Alternative in der Folge aller Alternativen (Attribut choice) sowie rekursiv die zu dieser Alternative gehörende Reihe von Einträgen (Attribut branch)“[Hei22a, S. 4]. Im Beispiel aus Abbildung 2 wird für den Zugriff auf das gelesene „+“ der Weg über `passes_` und `branch_` gewählt, da der Plus-Operator tatsächlich nicht nur für das Plus angewendet werden kann, sondern dank der Klammer `( + | - )` in seiner Implementierung auch für Subtraktionen verwendet wird. Genau für diese Alternative wird im AST eine Sequenz `passes_` mit einem Element (die tatsächlich gewählte Möglichkeit) erstellt.

4 Implementierung

4.1 Arbeiten mit dem AST

Bevor Code-Generierung selbst behandelt werden kann, soll es in diesem Abschnitt zuerst noch um die Interaktion mit den vom Parser aufgestellten AST gehen.

4.1.1 Offene Typen

Selbst für versierte C++-Programmierer könnte die Art und Weise, wie auf einzelne Elemente des ASTs zugegriffen wird, recht ungewöhnlich wirken. Das liegt wohl vor allem an der verwendeten DSL (**D**omain **S**pecific **L**anguage), die auf dem Konzept der offenen Typen basiert. Offene Typen sind ein Ansatz zur Lösung des sogenannten Expression-Problems. Dabei geht es um die Kombination der Vorteile objektorientierter Sprachen mit den Vorteilen funktionaler Sprachen. So soll es möglichst einfach sein, einer Abstraktion nicht nur neue Darstellungen hinzuzufügen, sondern auch ihre Verhaltensweisen zu erweitern. Offene Typen geben ihrem Benutzer also die Möglichkeit Elemente (Attribute und Funktionen) nach und nach hinzuzufügen, aber abzufragen als wären sie zentralisiert an einer Stelle deklariert.

Die für den MOSTflexiPL-Compiler verwendete Bibliothek `libCH` stellt für die Verwendung ihrer offenen Typen verschiedene Makros und Klammer-Operatoren zur Verfügung:

Listing 6: Offene Typen

```
1 TYPE(Student)
2 ATTR1(matr_nr_, Student, unsigned)
3 ATTRN(namen_, Student,  const char*)
4 Student lukas = Student(matr_nr_, 76010)(namen_, "Lukas", "Pietzschmann");
5 const char* v_name = lukas(namen_)[CH::A];
6 const char* n_name = lukas(namen_)[CH::A + 1];
7 std::cout << v_name << " " << n_name << ": " << lukas(matr_nr_) << std::endl;
```

In den ersten drei Zeilen wird der Typ `Student` mit den Attributen `matr_nr_` und `namen_` angelegt. `namen_` ist dabei ein mehrwertiges Attribut und kann wie eine List verwendet werden. Die verbleibenden Zeilen zeigen, wie auf die deklarierten Attribute zugegriffen werden kann. Sowohl eine einfache Abfrage als auch eine Zuweisung geschieht über den `operator()`. Für die Abfrage wird diesem Operator der Name des gewünschten Attributs übergeben (Zeile fünf bis sieben). Wurde zum Zeitpunkt der Abfrage noch kein Wert für das gefragte Attribut hinterlegt, wird der wohldefinierte Wert `nil` zurückgegeben. Möchte man einem Attribut einen Wert zuweisen, wird auch dafür zuerst der Name übergeben, gefolgt von dem neuen Wert beziehungsweise den neuen

Werten im Falle eines mehrwertigen Attributes (Zeile vier).
Allerdings ist nicht nur das Arbeiten mit Objekten eines offenen Typs etwas ungewohnt Auch mehrwertige Attribute beziehungsweise Sequenzen des Schablonen-Typs `CH::seq<T>` haben ihre Besonderheiten. So werden Sequenzen nicht mit Indizes $i \in \mathbb{N}$, sondern mit Objekten des Typs `posA` indiziert. „Die Konstante [CH::]A mit Typ posA bezeichnet den Anfang einer beliebigen Sequenz, d. h. die Position vor dem ersten Element.“[Hei22b, S. 18]. Mit den Operatoren `+` und `-` kann ein `posA`-Objekt um beliebig viele Stellen nach rechts beziehungsweise links geschoben werden, um den gewünschten Index zu generieren.

4.1.2 Traversieren des ASTs

Das Traversieren des abstrakten Syntaxbaums funktioniert über eine Art Visitor-Pattern. Dabei wird jedem Operator während seiner Erstellung eine Funktion zugewiesen, die dafür zuständig ist, genau für diesen Operator LLVM-IR-Code zu erzeugen. Hierfür muss dem offenen Typen `Oper` zuerst ein neues Attribut namens `code_gen_` hinzugefügt werden. Dies geschieht in der Datei `code_gen.h`, die zusammen mit `code_gen.cxx` alle für dieses Backend relevanten Funktionen enthält. Dank offener Typen ist das Hinzufügen des neuen Attributs simpel in einer Zeile möglich:

```
1 using CodeGenFun = llvm::Value* (*)(Expr);
2 ATTR1(code_gen_, Oper, CodeGenFun)
```

Haben dann alle Operatoren zu Beginn des Übersetzungsprozesses die korrekte Funktion zugewiesen bekommen, wird diese erst wieder nach Abschluss des Parsens verwendet. Der Parser liefert, ein korrektes MOSTflexiPL-Programm vorausgesetzt, genau einen Ausdruck des Typs `Expr` zurück, der den kompletten Quelltext abbildet. An dieser Stelle kann nun die Code-Erzeugung beginnen. So kann mit ...

```
1 auto code_gen_function = expr(oper_)(code_gen_);
```

... die zu dem Ausdruck (beziehungsweise dem Operator des Ausdrucks) gehörende Generierungsfunktion abgefragt werden und anschließend mit ...

```
1 code_gen_function(expr);
```

... aufgerufen werden. Dieses Abfragen und Aufrufen wird in der Funktion `Value* gen_expr_code_internal(Expr expr)` verpackt.
Genau diese Funktion beziehungsweise diese beiden Zeilen ähneln etwas dem am Anfang dieses Abschnittes angesprochenen Visitor-Pattern, wobei `gen_expr_code_internal` ungefähr das

Äquivalent zu `accept` ist. Es ist nicht bekannt, welcher Ausdruck tatsächlich hinter `expr` steckt beziehungsweise welche konkrete Generierungsfunktion in `code_gen_` gespeichert ist. Da aber während der Erstellung der Operatoren immer die korrekte Funktion übergeben wurde, findet der Ausdruck an dieser Stelle eben trotzdem den „richtigen Weg“. Durch diesen Aufruf von `code_gen_function(expr)` wird also am Ende beispielsweise `print_gen` aufgerufen. `print_gen` wird dann wiederum für den Operanden des `print`-Ausdrucks die Generierungsfunktion aufrufen. Auch dieser Operand hat möglicherweise selbst wieder Operanden, für die dann auch wieder `gen_expr_code_internal` aufgerufen wird. Und so weiter. Folgendes Aufruf-Diagramm zeigt grob den eben beschriebenen Ablauf für das Beispiel aus Unterabschnitt 3.2 - MOSTflexi-PLs AST:

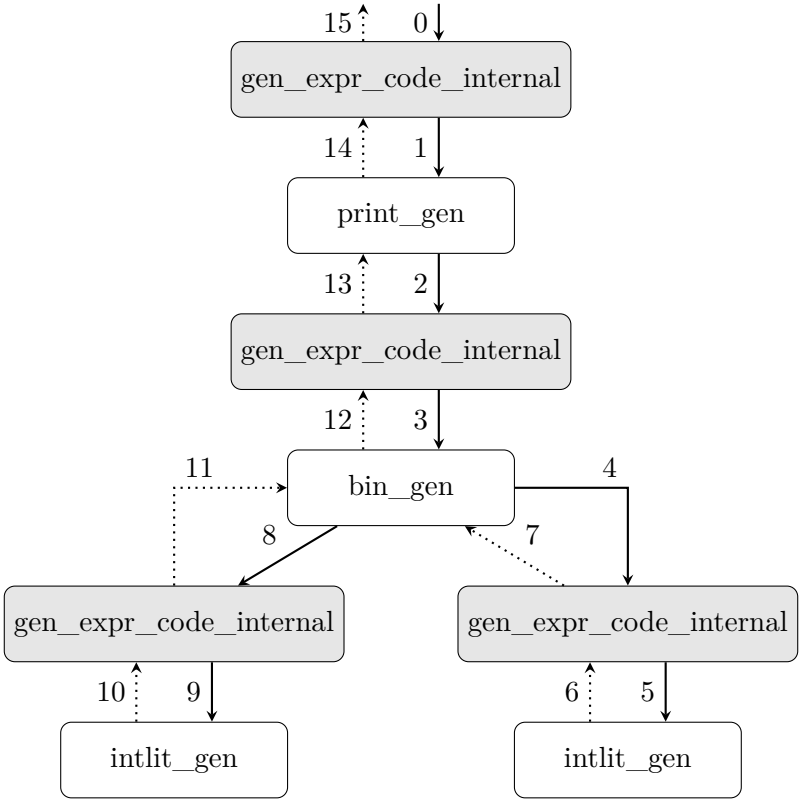


Abbildung 3: Ablauf-Diagramm für rekursive Aufrufe von Generierugsfunktionen

4.2 LLVMs API

Während sich die vorigen Kapitel vor allem mit der Eingabe (dem AST) und der Ausgabe (dem LLVM-IR-Code) des Backends beschäftigt haben, soll es hier nun um das Verbindungsstück der beiden Enden gehen.
LLVM bietet eine extrem mächtige, aber trotzdem erstaunlich simple API, um über den kompletten Lebenszyklus der LLVM-IR mit eben dieser zu interagieren. Die für dieses Kapitel relevanten Schnittstellen zur Erzeugung des Quelltextes werden primär über zwei globale Objekte zur Verfügung gestellt.

Ganz zentral ist die Klasse `Module`. Jedes Modul enthält eine Liste an globalen Variablen, Funktionen, eine Symboltabelle und verschiedene Daten über die Eigenschaften der Zielarchitektur. Auch besitzt ein Modul den kompletten Speicher, der vom Entwickler für die IR alloziert wurde. Wird also einem Modul ein Zeiger auf einen basic Block übergeben, der wiederum weitere Zeiger auf diverse Instruktionen enthält, gehören all diese Zeiger und der dahinter liegende Speicher dem Modul und der Entwickler muss sich nicht selbst um die Freigabe kümmern. Dies ist der Hauptgrund, wieso in diesem Projekt an nahezu allen Stellen mit normalen Zeigern und nicht mit Smartpointern gearbeitet wird.
Wie in Unterunterabschnitt 3.1.1 - Funktionen bereits nebenbei erwähnt, erzeugt dieses Backend ausschließlich ein Modul, weswegen ein globales Objekt der Klasse `Module` (`llvm_module`) ausreicht.

Das zweite zentrale, ebenfalls globale Objekt `ir_builder` ist vom Typ `IRBuilder<>`. Dieses Objekt bietet eine einfache und einheitliche Schnittstelle zum Einfügen von Instruktionen in den aktuellen basic Block. Funktionen der Klasse `IRBuilder<>` folgen typischerweise dem Muster `Value* Create<Instruktions-Name>(<Instruktions-Argumente>, <Ergebnis-Name>)`. So sieht die Signatur der Funktion zur Erstellung eines `load`-Befehls beispielsweise folgendermaßen aus:

```
1 Value* CreateLoad(Type* <Res-Typ>, Value* <Adresse>, StringRef <Res-Name> = "")
2 // Erzeugte Instruktion: %<Res-Name> = load <Res-Typ>, <Adresse>
```

Die Möglichkeit ein Ergebnis über den hier letzten Parameter zu benennen wird in der Implementierung dieser Arbeit sehr häufig genutzt, um den erzeugten LLVM-IR-Code leichter nachvollziehen zu können. Werden hier Code-Ausschnitte aus der Implementierung gezeigt, wird dieser Name aus Platzgründen allerdings weggelassen!

Neben diesen beiden globalen Objekten gibt es noch einen dritten essenziellen Container namens `Value`. Dieser ist die Basisklasse für alle berechenbare Werte. Sie wird also überall verwendet, wo ein beliebiger Wert erwartet oder erzeugt wird. Man kann sich ein Objekt des Typs `Value` wie die zur Übersetzungszeit existierende Repräsentation von zur Laufzeit existierenden

Werten vorstellen. Dabei können mit diesem Typ alle Werte dargestellt werden, die ein virtuelles Register enthalten sind. Man spricht dabei von „first-class“ Werten [LLVa, Type System]. Möglich sind also Zahlen oder Adressen, aber nicht Strukturen oder Typen.

4.3 Abstraktionen über häufige Funktionen

Anstatt LLVMs API direkt zu verwenden wurden im Rahmen dieser Arbeit diverse weitere Hilfsfunktionen geschrieben, die häufig verwendetes auf eine dem Projekt zugeschnittene Art verfügbar machen. Außerdem wurden einige Funktionen entwickelt, die bestimmte Konzepte der Programmiersprache MOSTflexiPL auf Ebene der IR abbildbar machen.
Die folgenden Abschnitte sollen einen kleinen Einstieg in die entwickelten Hilfsfunktionen geben, damit später der Fokus ausschließlich auf dem jeweils behandeltem Konzept liegen kann.

Bei all diesen Funktionen ist es wichtig, stets zwischen Übersetzungs- und Laufzeit zu trennen. Alle hier vorgestellten Funktionen er- oder beschaffen Werte durch Emittieren von Befehlen. Ergo arbeiten sie nur mit der Compilezeit-Repräsentation des erst zur Laufzeit existierenden Wertes! Da man aber oft schnell dazu verfällt zu denken, dass Dinge zur Übersetzungszeit geschehen, obwohl dies nicht der Fall ist, sollte man diese Arbeit und die von ihr vorgestellten Datenstrukturen und Codeausschnitte immer mit bedacht auf diesen Fallstrick lesen!

4.3.1 Abstraktionen über LLVM

Funktionen dieser Kategorie finden sich im Namensraum `utils`.

Die wohl meistverwendete Funktion aus diesem Namespace ist `ConstantInt* get_integer_constant(int constant, int width = INTEGER_WIDTH, bool is_signed = true)` . Wie der Name bereits sagt, wird hier ein konstanter ganzzahliger Wert mit der angegebenen Breite in Bits zurückgegeben. Achtung: auch boolesche Konstanten können hiermit erstellt werden, indem die Breite 1 verwendet wird! Da MOSTflexiPL aktuell keine anderen primitiven Typen neben natürlichen Zahlen und booleschen Werten kennt, werden alle Konstanten und Literale über diese Funktion erstellt.

Neben dem Erstellen von Konstanten bietet der Namensraum auch zwei Funktionen zum Erstellen von Typen: `IntegerType* get_integer_type(int width = INTEGER_WIDTH)` und `PointerType* get_ptr_type(Type* underlying_type, int indirections = 1)`. Wird `get_integer_constant` mit Breite `n` aufgerufen, hat der Resultatwert exakt den Typen, den ein Aufruf an `get_integer_type` ebenfalls mit Breite `n` liefert.

Neben dem Erstellen von Konstanten und Typen ist die Abfolge von Funktionsaufrufen zum Einfügen von Instruktionen in einen basic Block ein weiteres extrem häufig vorkommendes Pattern. Hierfür bietet der Namensraum `utils` die Funktion `void generate_on_basic_block(BasicBlock* bb, Lambda lambda, bool reset_insert_point = false)`, die neben dem Block auch ein Lambda verlangt, dessen Code auf dem übergebenen Block ausgeführt wird. Dies wird erreicht, indem mit den Zeilen ...

```
1 ir_builder->SetInsertPoint(basic_block);
2 lambda()
```

... der Cursor des IRBuilders zuerst auf den übergebenen Block gesetzt wird und anschließend der übergebene Callback ausgeführt wird. Mit einem zusätzlichen booleschen Wert an letzter Stelle kann `generate_on_basic_block` angewiesen werden eben diesen Cursor nach Ausführung des Lambdas wieder an die alte Position zurückzusetzen. Im Regelfall ist dies nicht nötig, aber folgende Situation würde ohne diese Option falschen Code erzeugen:

```
1 define fastcc i32 @main() {
2     ...
3     |
4 }
5 define i32 @pow(i32 base, i32 exp) {
6     ...
7     ret i32 %res|
8 }
```

In der Main-Funktion wurde beliebiger Code erzeugt, bis die Codegenerierung auf einen Potenz-Ausdruck gestoßen ist. Diese Stelle wird in obigem Ausschnitt durch die blaue Markierung angedeutet. Für die Potenz erzeugt dieses Backend eine eigene Funktion, die aber erst vor ihrem ersten Aufruf generiert wird. Ist die Generierung der `pow`-Funktion dann abgeschlossen, ist der Cursor des IRBuilders an der rot markierten Position. Nun darf aber die folgende `call`-Instruktion eben nicht an dieser Stelle eingefügt werden! Stattdessen muss der Cursor an die blaue Markierung zurückgesetzt werden, was genau durch das Übergeben von `true` als letzten Parameter an `generate_on_basic_block` erreicht werden kann.

Die nächste Gruppe an Funktionen wird vor allem für das Arbeiten mit LLVMs Strukturen von Bedeutung sein. So werden von `utils` auch vereinfachte Funktionen für Schreib- und Lesezugriffe auf diese geboten. Der erste Schritt ist dabei immer das Ausführen einer `getelementptr`-Instruktion (kurz: gep). Diese nimmt neben dem Typ der Struktur in die indiziert werden soll und einem Zeiger auf die Struktur selbst, auch eine Liste an Indizes, mithilfe welcher die Adresse auf das gewünschte Element berechnet wird.

Folgendes Beispiel verdeutlicht die Verwendung der verschiedenen Parameter:

Listing 7: GEP-Beispiel C-Code

```
1 struct A { int i; struct A* as; };
2 int main(){
3     struct A a;
4     a.i;
5     a.as[2];
6 }
```

Dieses recht kurze C-Programm resultiert in folgendem LLVM-IR Code:

Listing 8: GEP-Beispiel LLVM-IR-Code

```
1  %struct.A = type { i32, %struct.A* }
2  define i32 @main() {
3    %a = alloca %struct.A, align 8
4    %i = getelementptr inbounds %struct.A, %struct.A* %a, i32 0, i32 0
5    %0 = load i32, i32* %i, align 8
6    %as = getelementptr inbounds %struct.A, %struct.A* %a, i32 0, i32 1
7    %1 = load %struct.A*, %struct.A** %as, align 8
8    %arrayidx = getelementptr inbounds %struct.A, %struct.A* %1, i64 2
9    ret i32 0
10 }
```

Die für diesen Abschnitt interessanten Zeilen sind vier und fünf des Quellcodes, da sie sowohl das Abfragen eines einzelnen Elements `i` als auch das Abfragen mit einer zusätzlichen Indirektion (`as[2]`) enthalten. In den olivgrünen Zeilen des Kompilats ist die erste Abfrage enthalten. Während die ersten beiden Parameter der `gep`-Instruktion recht offensichtlich sind, würde man in der Liste der Indizes wohl eher nur eine `0` erwarten. Die erwartete `0` ist die zweite in der Liste und meint den Index des Elements in der Struktur, die erste `0` indiziert allerdings nicht in die Struktur hinein, sondern über die Struktur hinweg. Der Zugriff auf das zweite Element aus `as` verdeutlicht diesen Unterschied noch einmal. Während Zeile sechs mit den erwarteten beiden Indizes erst einmal nur einen Zeiger auf die Liste selbst errechnet, ist der erste Index aus der Instruktion in Zeile acht nicht mehr null. Der übergebene Zeiger `%1` zeigt auf das erste Element der Liste und um nun nicht in dieses Element hineinzuindizieren, sondern den Zeiger zwei Elemente nach rechts zu verschieben, wird mit der `2` als ersten Index genau dieses Indizieren über das Element hinweg erreicht. Diese Besonderheit des ersten Indexes der `getelementptr`-Instruktion muss also auch beim Aufruf der Hilfsmethoden `extract_from` und `insert_into` beachtet werden.

Während beide entwickelten Hilfsfunktionen als Erstes eine `gep`-Instruktion emittieren, führt `extract_from` anschließend einen `load`-Befehl aus, `insert_into` hingegen wird eine `store`-Instruktion verwenden, um nicht den errechneten Wert zu laden, sondern den übergebenen Wert an der errechneten Stelle zu speichern. Beide Funktionen sind variadisch, um das explizite Erstellen einer Index-Liste beim Aufruf zu vermeiden und es möglich zu machen, alle Indizes nacheinander als Parameter anzugeben.

4.3.2 Abstraktionen über MOSTflexiPL Konzepte

MOSTflexiPL-Werte

Ein zentrales Konzept aus MOSTflexiPL ist das Existieren verschiedener Arten von Werten. Dabei wird zwischen drei verschiedenen Arten differenziert, wobei ein Wert immer nur zu einer dieser drei Kategorien gleichzeitig gehören kann:

Natürliche Werte:
: Natürliche Werte sind ganz normale boolesche oder ganzzahlige Werte. Beispiele für Werte dieser Art sind das Literal `1` oder das Ergebnis des Ausdrucks `1 < 2`.

Synthetische Werte:
: Synthetische Werte sind im Gegensatz zu natürlichen eigentlich keine wirklichen Werte. Sie entstehen beispielsweise „durch Konstantendeklarationen ohne explizite Initialisierung wie z. B. Person : type, p : Person oder auch i : int“[Hei22a, S. 20]. Die zentrale Eigenschaft synthetischer Werte ist dabei, dass sie immer eindeutig sind. Dadurch kommen solche Werte typischerweise an Stellen vor, an denen der konkrete Wert eines Objekts irrelevant ist, die Einzigartigkeit des Objekts allerdings größte Bedeutung hat.

Nil-Werte:
: Im Gegensatz zu synthetischen Werten sind nil-Werte wieder ein äußerst verbreitetes Konzept. Ein solcher Wert weißt immer auf die Abwesenheit von etwas hin. Ganz äquivalent zu `null` aus Java oder `nullptr` aus C++. So erhalten uninitialisierte Variablen beispielsweise nil als ihren Wert.

Im Gegensatz zu den vorher besprochenen Klammern, wird für das Darstellen von allgemeinen MOSTflexiPL-Werten keine eigene Struktur erstellt, die dann mit den jeweiligen Eigenschaften gefüllt wird. Dieses Vorgehen hätte zwei große Nachteile: zum einen müsste jede Abfrage eines Wertes über einen Speicherzugriff stattfinden und zum anderen würde man dadurch auch die Möglichkeit verlieren, triviales Constant-Folding zu betreiben.
Anstelle dieses naiven Ansatzes wurde folgendes Vorgehen gewählt:

nil	synt	nat	<Wert>		
34	33	32	31	...	0

Tabelle 1: Darstellung allgemeiner MOSTflexiPL-Werten

So existiert für jede oben genannte Eigenschaft ein eigenes Flag. Diese drei Flags werden zusammen hinter die höherwertigen Bits des tatsächlichen Wertes gehängt, um somit einen 35 Bit breiten MOSTflexiPL-Wert zu erzeugen. Auch wenn es auf den ersten Blick eher ungewohnt wirkt, erlaubt es LLVM tatsächlich Integer beliebiger Breite zu verwenden. Wobei der hier verwendete Typ `i35` während der Maschinencode-Generierung wahrscheinlich zu einem 64 Bit Integer aufgerundet wird. Dieses Vorgehen besitzt keinen der Nachteile der Struktur-Variante, da es offensichtlich keinen Speicherzugriff mehr benötigt, um auf einfache Werte zuzugreifen, weswegen dann auch triviales Constant-Folding wieder möglich ist. Selbst das Auslesen der einzelnen Flags oder des tatsächlichen Wertes kann teilweise zur Übersetzungszeit durch geeignete bitweise Operatoren passieren.

Angenommen es liegt folgender Wert vor: $\underbrace{001}_{Flags}\underbrace{00000000.00000000.00000000.00101010}_{echter Wert}$. Nun kann über eine einfache Bitmaske abgefragt werden, ob das gewünschte Flag gesetzt ist:

$$\begin{array}{r} 001.00000000.00000000.00000000.00101010 \\ \&001.00000000.00000000.00000000.00000000 \\ \hline 001.00000000.00000000.00000000.00000000 \end{array}$$

Ist das Ergebnis echt größer 0, ist das mit `001` assoziierte Flag `nat` gesetzt. Ergo kann hier der in den unteren 32 Bit gesetzte Wert als Integer interpretiert werden. Im Falle eines synthetischen Wertes würden diese unteren Bits eine eindeutige ID enthalten.
Neben dieser Überprüfung (`Value* is(Value* value, int flag)`) bietet der Namensraum `flx_value` noch folgende weitere Hilfsfunktionen an:

`Value* get(int flag, int value)`:

`get` erzeugt einen neuen MOSTflexiPL-Wert mit dem angegeben Flag und dem eigentlichen Wert. Hierbei ist es wichtig, dass der eigentliche Wert 0 ist, falls das Flag `nil` gesetzt ist, da `nil`-Werte stets als negative Wahrheitswerte interpretiert werden. Das eigentliche Erstellen des MOSTflexiPL-Wertes geschieht dann durch eine linksschiebe-Operation um 32 Stellen von `flag` und ein anschließendes Addieren von `value`.

`Value* promote_to_flx_value(Value* value, int flag)`:

Diese Hilfsfunktion wandelt einen einfachen Wert zu einem MOSTflexiPL-Wert mit dem übergebenen Flag um, indem `value` mithilfe der Instruktion `zext` links mit Nullen auf 35 Bit aufgefüllt wird und anschließend die um 32 Stellen nach links geschobenen Flags dazugezählt werden.

`Value* truncate_to_raw_value(Value* value)`:

Diese Funktion macht exakt das Gegenteil der vorherigen. Die akzeptiert einen MOSTflexiPL-Wert und schneidet mit der Instruktion `trunc` die oberen drei Bits ab, um lediglich den tatsächlichen Wert zurückzulassen. Soll also beispielsweise `1 + 2` ausgewertet werden, muss vorher zwingend diese Funktion angewandt werden, da die oberen drei Flag-Bits der beiden Literale ansonsten das Ergebnis verfälschen würden.

Die einzelnen Flags sind über das Enum `enum { nat = 1, synt = 2, nil = 4 }` implementiert, um sicherzustellen, dass ein Flag immer nur genau eine Stelle der drei reservierten Bits belegt.

4.4 Codegenerierungs-Prolog

Bevor die in Unterunterabschnitt 4.1.2 - Traversieren des ASTs bereits grob behandelte Funktion `gen_expr_code_internal` mit dem von Parser generierten Ausdruck aufgerufen werden kann und damit die Codegenerierung angetreten wird, muss zuerst noch ein gewisses Grundgerüst, einerseits in der IR selbst, andererseits aber auch im Backend, aufgesetzt werden. Dies wird in der Funktion `gen_expr_code` getan. Diese Funktion stellt die einzige Schnittstelle der Codeerzeugung nach außen dar, daher auch das Postfix „_internal“ der anderen `gen_expr_code` Funktion.

Das Backend muss im ersten Schritt seine globalen Objekte initialisieren. Hierzu wird zuerst das Modul initialisiert, wozu ein `Context` benötigt wird. Was genau dieser enthält, ist absichtlich[4] nicht klar dokumentiert, man kann sich einen Kontext allerdings wie einen Container vorstellen, der den kompletten globalen Zustand des „LLVM Systems“ enthält. Ein Objekt dieser Klasse wird über den parameterlosen Konstruktor erstellt.
Nun kann das Modul erstellt werden. Hierfür wird dem Konstruktor an zweiter Stelle der eben beschriebene Kontext übergeben. Zusätzlich benötigt jedes Modul eine eindeutige ID. Da dieses Projekt nur mit einem Modul arbeitet, sinnvolle Namen für verschiedene Module also nicht allzu relevant sind und die ID eine Zeichenkette verlangt, wurde hier „MOSTflexiPL“ als ID gewählt. Folgender Code-Ausschnitt zeigt die Initialisierung:

Listing 9: Initialisierung der relevanten globalen Objekte

```
1 std::unique_ptr<LLVMContext> llvm_context_temp = std::make_unique<LLVMContext>();
2 std::unique_ptr<Module> llvm_module_temp = std::make_unique<Module>("MOSTflexiPL
    ↪ ", *llvm_context_temp);
3 std::unique_ptr<IRBuilder<>> ir_builder_temp = std::make_unique<IRBuilder<>>(*
    ↪ llvm_context_temp);
4 llvm_module = llvm_module_temp.get();
5 ir_builder = ir_builder_temp.get();
6 current_context = ValueContext::mk_ptr();
```

Auch wenn die globalen Objekte normale Zeiger sind, werden zu ihrer Initialisierung Smartpointer verwendet. Dies hat den klassischen Vorteil von automatischer Speicherverwaltung: da `gen_expr_code` an mehreren Stellen verlassen werden kann, ist es so deutlich einfacher sicherzustellen, dass die hinter den Zeigern liegenden Objekte stets korrekt aus dem Speicher genommen werden. Achtung: die normalen Zeiger `llvm_module` und `ir_builder` zeigen nach den Aufrufen der Destruktoren von `llvm_module_temp` und `ir_builder_temp` ins Nichts. Dies stellt allerdings keine Fehler in der Programmierung dar, da ein Benutzer der Codegenerierung ausschließlich mit `gen_expr_code` interagieren darf und diese Funktion die Zeiger direkt zu ihrem Beginn wieder korrekt setzt. Das Verhalten eines händischen Aufrufs aller anderen Funktionen ist undefiniert und würde durch die falsch gesetzten Zeiger mit großer Sicherheit in einem Segmentation-Fault enden!
Die in der letzten Zeile initialisierte Variable `current_context` steht mit dem von LLVM verwendetem Kontext in keinem Zusammenhang. Die Existent dieses Kontextes wird später in Unterunterabschnitt 4.5.3 - Variablen-Abfrage noch wichtig sein.
Mit der Initialisierung dieser Variablen ist die Konfiguration des Backends beendet. Der nächste Schritt ist das Konfigurieren und Aufsetzten des Moduls.

[4] Das Wort *opaquely* aus „It opaquely owns [...]“ aus der offizitellen Dokumentation (`https://llvm.org/doxygen/classllvm_1_1LLVMContext.html#details`) lässt darauf schließen, dass es für den Entwickler eines Backends wohl nicht relevant ist, was genau dieser Kontext nun enthält

Um die Konfiguration des Moduls so variabel wie möglich zu gestalten, kann der Funktion `gen_expr_code` ein Lambda übergeben werden, das das Modul als Referenz erhält und es somit ganz frei konfigurieren kann.
Die wichtigsten Schritte sind dabei das Übergeben von Informationen über die Zielarchitektur. Diese Informationen bestehen aus zwei Teilen: dem Target-Triple und dem Datenlayout. `x86_64-redhat-linux-gnu` ist beispielsweise das Target-Triple für eine RedHat basierte Linux-Distribution, laufend auf einem Intel x86_64 System. Um aus diesem Triple nun ein passendes Datenlayout zu gewinnen, benötigt LLVM allerdings noch ein paar weitere Informationen. So muss beziehungsweise kann man zusätzlich angeben, auf welche konkrete Prozessor-Generation man abzielt oder welche Funktionen diese hat. Dieses Projekt verwendet der Einfachheit halber `"generic"` als CPU ohne jegliche Angabe an zusätzlichen Funktionen. Aus all diesen Informationen lässt sich zum Schluss eine `TargetMachine` erstellen, die den Charakter der konfigurierten Architektur beschreibt. Mit den beiden Zeilen ...

```
1 module.setDataLayout(target_machine->createDataLayout());
2 module.setTargetTriple(target_triple);
```

... werden alle Informationen in das Modul übertragen. Der erzeugte LLVM-IR-Code wird dadurch dann folgende Zeilen enthalten:

```
1 target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
    ↪ :16:32:64-S128"
2 target triple = "x86_64-redhat-linux-gnu"
```

Während das Target-Triple exakt dem oben genannten Beispiel gleicht, ist das Datenlayout recht kryptisch. Hier sind Informationen wie das Stack-Alignment (hier: 128Bit) oder ob das Little oder Big-Endian Format genutzt wird (hier: Little-Endian) enthalten.
Diese beiden Zeilen sind zwar nicht zwingend nötig, aber sehr empfehlenswert, da auf ihnen alle Architektur orientierten Optimierungen beruhen[LLVd, The Basics].
Mit dem Setzen der beiden Zeilen und dem Terminieren des Lambdas ist die Konfiguration des Moduls beendet und es kann im nächsten Schritt das erste Mal LLVM-IR-Code erzeugt werden.

Während MOSTflexiPL es dem Programmierer erlaubt Ausdrücke auch auf globale Ebene, außerhalb von Funktionen zu schreiben, ist die LLVM-IR in dieser Hinsicht limitiert. Hier ist es nicht möglich beliebige Instruktionen außerhalb einer Funktion zu platzieren, weswegen vor der Codegenerierung für die eigentliche Eingabe zuerst eine Main-Funktion in das Modul geschrieben werden muss. Diese Funktion wird dann später jeglichen Top-Level-MOSTflexiPL-Code enthalten. Die Einstiegs-Funktion `main` zu nennen bringt zwei Vorteile mit sich: zum einen ist beim Lesen des IR-Codes direkt erkennbar, welchen Zweck diese Funktion erfüllt. Zum anderen wird der Linker automatisch dafür sorgen, dass die Funktion mit dem Namen `main` zum Starten an

`__libc_start_main`[5] übergeben wird. Die Details dieses Prozesses sind für das Thema dieser Arbeit allerdings wohl nicht sonderlich relevant. Wichtig ist dabei nur die folgende Erkenntnis: die LLVM-IR bietet selbst keine Möglichkeit zu bestimmen, mit welcher Funktion das erzeugte Programm einsteigt. Das festzulegen ist ausschließlich die Aufgabe des Linkers, der dazu mit den korrekten Parametern instruiert werden muss. Da hier das Standardverhalten, nämlich das Linken mit libC und dem Verwenden einer Funktion namens `main` das erwünschte ist, müssen keine weiteren Maßnahmen beim späteren Linken beachtet werden.

Das Erstellen der Main-Funktion funktioniert wie folgt:

Listing 10: Generierung der main-Funktion #1

```
1 BasicBlock* main_entry = BasicBlock::Create(llvm_module->getContext());
2 predef_functions::main()->getBasicBlockList().push_back(main_entry);
3 utils::generate_on_basic_block(main_entry, [&]() {
4   Value* res = gen_expr_code_internal(expr);
5   ir_builder->CreateRet(flx_value::truncate_to_raw_value(res));
6 });
```

Im ersten Schritt wird der erste basic Blocks der Funktion erstellt. Nach dem Erstellen des Blocks wird er der Main-Funktion einfach mit `push_back` hinzugefügt. Das tatsächliche Erstellen der Funktion wird durch die Hilfsfunktion `main` aus dem Namensraum `predef_functions` versteckt.

Listing 11: Generierung der main-Funktion #2

```
1 if(Function* function = llvm_module->getFunction(predef_function_names::main))
2   return function;
3 Type* return_type = utils::get_integer_type();
4 std::vector<Type*> args = {};
5 bool is_variadic = false;
6 FunctionType* function_type = FunctionType::get(return_type, args, is_variadic);
7 Function* function = Function::Create(function_type, Function::LinkOnceAnyLinkage
    ↪ , predef_function_names::main, llvm_module);
8 function->setCallingConv(CallingConv::Fast);
9 return function;
```

Um sicherzustellen, dass auch bei mehrfachem Aufrufen von `predef_functions::main` nicht mehrere Main-Funktionen generiert werden, wird in der ersten Zeile abgeprüft, ob das Modul bereits eine Funktion namens `main` kennt, ist das der Fall, wird die bereits im Modul gespeicherte Funktion zurückgegeben. Andernfalls werden die bereits in Unterunterabschnitt 3.1.1 - Funktionen besprochenen Eigenschaften gesetzt und die daraus erzeugte Funktion zurückgeliefert.

[5] `__libc_start_main` wird vom Linker (vorausgesetzt, es wurde kein anderes Verhalten explizit verlang) direkt an den Start eines Programms gelinked, um die C-Bibliothek zu initialisieren und letztendlich die Main-Funktion des eigentlichen Programms zu starten.

Wenn die eben beschriebene Funktion in Listing 10 - Generierung der main-Funktion #1 in Zeile zwei terminiert und dabei der Kopf der zu generierenden Funktion in das Modul eingefügt wurde, kann ab Zeile drei der Inhalt erzeugt werden. Dazu werden alle Instruktionen, in den vorher erstellten und eingefügten basic Block geschrieben, hierfür wird über den ersten Aufruf von `gen_expr_code_internal` die Codeerzeugung für das Eingabeprogramm angestoßen. Ab dann folgt das Backend dem in Unterunterabschnitt 4.1.2 - Traversieren des ASTs beschriebenen Pfad.
Wurde der AST abgearbeitet, muss die Main-Funktion zum Schluss noch durch eine return-Instruktion beendet werden.

Nun enthält das Modul den kompletten LLVM-IR-Code und kann im letzten Schritt weiter verarbeitet werden. Bevor dieser letzte Schritt allerdings behandelt wird, soll es nun um die Details der Implementierung der verschiedenen Operatoren gehen.

4.5 Vordefinierte Operatoren

Alle bisher verwendeten Operatoren wie `print` oder `+` und `-` sind bereits vordefiniert und die Implementierung ihrer Generierungsfunktion ist größtenteils recht trivial. Aus diesem Grund werden in diesem Kapitel nicht alle 27 bereits existierenden Operatoren[6] beschrieben, sondern ein Paar ausgewählte, die nach und nach komplexere Konzepte einführen.

4.5.1 Nacheinanderausführung

Ein sehr simpler Einstieg in die Codegenerierung für vordefinierte Operatoren ist die Generierungsfunktion `sequ_gen`.

Listing 12: sequ_gen

```
1 Value* sequ_gen(Expr expr) {
2   Expr lhs = expr(row_)[CH::A + 1](opnd_);
3   Expr rhs = expr(row_)[CH::Z](opnd_);
4   gen_expr_code_internal(lhs);
5   return gen_expr_code_internal(rhs);
6 }
```

Die einzige Aufgabe des Semikolon-Operators ist es nacheinander seinen linken und rechten Operanden auszuwerten, wobei das Ergebnis des rechten Operanden auch das Ergebnis des Operators selbst ist. Diese Beschreibung lässt sich eins zu eins auf die Implementierung abbilden.

[6] Eine Liste aller vordefinierten Operatoren kann unter Unterabschnitt 7.1 - MOSTflexiPL-Operatoren gefunden werden.

4.5.2 Ausgabe

Während die Implementierung des Semikolon-Operators, sowie die der allermeisten Operatoren, auch von diesem Backend erzeugt wurde, wird für den `print`-Operator anders verfahren. Anstelle selbst Code zu erzeugen, der wohl in einem Aufruf des write-Systemcalls resultieren würde, wird hier lediglich die aus libC bekannte Funktion `printf` wiederverwendet, um an dieser Stelle unnötige Komplexität zu vermeiden. Vor allen bei der Implementierung benutzerdefinierter Operatoren werden später in dieser Arbeit noch weitere Stellen aufkommen, an denen es sinnvoll erscheint und wohl außerhalb einer wissenschaftlichen Arbeit auch wirtschaftlich sinnvoll wäre, weitere Abstraktionen aus libC (oder eventuell sogar C++'s STL) zu verwenden. Allerdings wird außer für die Aus- und Eingabe an keiner weiteren Stelle die Abkürzung über eine bereits existierende Bibliothek gewählt, da es explizit das Ziel dieser Arbeit ist eine „Proof of Concept"-Implementierung zu entwickeln, die zeigt, dass es ausschließlich unter Verwendung der LLVM-IR möglich ist für eine so extrem abstrahierende Sprache Code zu erzeugen. Während die händische Implementierung von `print` und `read` wohl wenig neue Erkenntnisse geliefert hätte und es somit auch in Ordnung ist hier diese Abkürzung zu wählen, soll die anspruchsvolle und erkenntnisreiche Implementierung benutzerdefinierter Operatoren später nicht abgekürzt werden!

`print` ist also wie folgt implementiert:

Listing 13: print_gen

```
1  Item printee = expr(row_)[CH::A + 1];
2  Value* printee_v = gen_expr_code_internal(printee(opnd_));
3  Value* width_v;
4  if(Expr width = expr(row_)[CH::Z](passes_)[CH::A](branch_)[CH::Z](opnd_))
5    width_v = flx_value::truncate_to_raw_value(gen_expr_code_internal(width));
6  else
7    width_v = utils::get_integer_constant(0);
8  Value* is_nat = flx_value::is(printee_v, flx_value::nat);
9  Value* string = ir_builder->CreateSelect(is_nat, utils::construct_string("%*d\n",
       ↪  "prinf_format"), utils::construct_string("\n", "empty_printf"));
10 std::vector<Value*> args = {string, width_v, flx_value::truncate_to_raw_value(
       ↪ printee_v)};
11 ir_builder->CreateCall(predef_functions::printf(), args);
12 return printee_v;
```

Nachdem in Zeile zwei der Code für den ersten Operanden erzeugt wurde, wird als nächstes abgefragt, ob auch eine `width` angegeben wurde. So kann der Ausgabe-Operator also auch folgendermaßen aufgerufen werden: `print 42 width 5`. Dabei wird sichergestellt, dass die Ausgabe mindestens `5` Zeichen lang ist. Wurde keine Breite angegeben, wird in Zeile sieben `0` als eben

diese Breite verwendet. Bevor der erste Operand mit der angegebenen Breite allerding ausgegeben werden kann, muss im nächsten Schritt überprüft werden, ob er `nil` ist. Dazu wird eine der in Unterunterabschnitt 4.3.2 - Abstraktionen über MOSTflexiPL Konzepte beschriebenen Funktionen verwendet. Abhängig von diesem Ergebnis wird `print` mithilfe der `select`-Instruktion unterschiedlichen Text ausgeben. Dieser Befehl hat den Vorteil einen von zwei Werten, basierend auf einem Wahrheitswert auswählen zu können, ohne dafür einen Sprung auf Ebene der IR einfügen zu müssen (ähnlich wie der ternäre Operator diverser Hochsprachen).
Wurden die Argumente für `printf` nun zusammengestellt, kann die Funktion aufgerufen werden. Das eigentliche Erstellen des Prototyps ist wie schon bei der main-Funktion hinter einer Hilfsfunktion versteckt. Diese muss zwei Dinge beachten: Zum einen muss die Calling-Convention auf `ccc` (**C**-**C**alling-**C**onvention) gesetzt werden, um die korrekte Übergabe der Parameter an libC's `printf` sicherzustellen. Weiterhin muss der Prototyp `external` gelinked werden, da ansonsten ausschließlich im generierten Modul nach seiner Implementierung gesucht wird, die dort natürlich nicht gefunden werden würde. Werden diese beiden Dinge beachtet, wird die tatsächliche Ausgabe der übergebenen Argumente von der C-Bibliothek übernommen.

4.5.3 Variablen-Abfrage

Die bisher gezeigten Operatoren mussten selbst keine zur Übersetzungszeit feststehenden Informationen verwalten. Alle Operatoren, die direkt mit Variablen oder Konstanten arbeiten, müssen das allerdings schon. Hierfür werden von diesem Backend sogenannte Kontexte aufgestellt. Detailliert wird auf diese Datenstrukturen noch in Unterabschnitt 4.6 - Benutzerdefinierte Operatoren eingegangen. Hier also erst einmal nur das grobe Konzept:

Ein Kontext wird von der Klasse `ValueContext` repäsentiert und enthält unter anderem zwei wichtige Elemente:

`named_values`:
: Hier wird dem Namen einer Variable oder Konstanten unter anderem der zugehörige Wert (Typ `Value*`) zugeordnet. Dabei ist es wichtig, dass dies nie unmittelbare Werte sind, sondern immer Zeiger auf eine Speicherstelle (siehe Unterunterabschnitt 3.1.2 - Globale und lokale Variablen).

`parent`:
: Hier wird, falls vorhanden, der umschließende Kontext gespeichert. Wird in `named_values` der gewünschte Name nicht gefunden, wird weiter in diesem Kontext gesucht.

Die Generierungsfunktion `query_gen` muss nun mit diesen Kontexten arbeiten, um die gewünschte Speicherstelle der gewünschten Variable oder Konstanten zu finden.

Listing 14: query_gen

```
1 Value* query_gen(Expr expr) {
2   std::stringstream name;
3   name << expr(row_)[CH::Z](opnd_)(row_)[CH::A](word_);
4   auto* alloca = current_context->lookup_value(name.str());
5   if(alloca == nullptr)
6     return flx_value::get(flx_value::nil);
7   return ir_builder->CreateLoad(utils::get_integer_type(FLX_VALUE_WIDTH), alloca
      ↪ ->value);
8 }
```

Die in Zeile vier verwendete Variable `current_context` zeigt, wie der Name schon sagt, immer auf den aktuell zu verwendenden Kontext. Diese Variable wird an entsprechenden Stellen so verändert, dass diese Eigenschaft stets gegeben ist. Die Methode `lookup_value` wird, falls der Name nicht im aktuellen Kontext gefunden wurde, so lange die Kette an umschließenden Kontexten entlang laufen bis der globale Kontext erreicht oder der Name gefunden wurde. Wird aus ersterem Grund abgebrochen, ist der Name im aktuellen Kontext nicht sichtbar und es wird der Nullzeiger zurückgegeben. Dies wird in Zeile fünf abgeprüft und entsprechend behandelt. Wieso hier `nil` zurückgegeben und kein Fehler geworfen wird, wird im Verlauf der Arbeit noch einmal aufgegriffen.
Wird das Symbol aber schon gefunden, enthält die Variable `alloca`, wie weiter oben bereits erwähnt, den Wert nicht unmittelbar, sondern einen Zeiger auf ihn[7]. Im letzten Schritt muss nun der Wert an der Adresse des Zeigers geladen werden, um ihn als Ergebnis der Generierungsfunktion zurückgeben zu können.

4.6 Benutzerdefinierte Operatoren

Wie die Beispiele aus dem letzten Kapitel aufgezeigt haben, gibt es an manchen Stellen der Implementierung vordefinierter Operatoren zwar ein paar Dinge zu beachten, die Hauptkomplexität dieser Arbeit liegt aber wohl nicht darin. So sind Schleifen, Operatoren zur Ein- und Ausgabe oder zur arithmetischen Verrechnung von Operanden wirklich gut verstandene und keine einzigartigen Konzepte und lassen sich deswegen auch direkt in LLVM-IR-Code abbilden. Benutzerdefinierte Operatoren und vor allem deren beliebig geschachtelte Klammern in Signatur und Implementierung sind allerdings kein verbreitetes Konzept und erlauben es dem Benutzer Schleifen und Bedingungen durch sehr prägnanten und kurzen MOSTflexiPL-Code auszudrücken. Diese beiden Eigenschaften sorgen für eine sehr komplexe Codeerzeugung und den Hauptaufwand dieser Arbeit.

[7] Natürlich enthält nicht `alloca` selbst den Zeiger, sondern der Zeiger wird von dem LLVM-Wert den `alloca` enthält repräsentiert.

Um die Möglichkeiten von Klammern in Operatoren zu verstehen, wird für dieses Kapitel folgendes Beispiel betrachtet:

Listing 15: calc.flx

```
1 calc [minus] (x:int) { (plus|minus) (y:int) } -> (int =
2   s : int?;
3   [ s =! -x | s =! x ];
4   { ( s =! ?s + y | s =! ?s - y ) };
5   ?s
6 );
7 calc minus 1 plus 2 minus 3
```

Die Signatur des Operators kann wie folgt über den Aufruf des Operators gelegt werden:

$$\underbrace{\underbrace{calc}_{calc}\ \underbrace{[minus]}_{minus}\ \underbrace{(x : int)}_{1}\ \{\ \underbrace{\underbrace{(plus|minus)}_{plus,\ minus}\ \underbrace{(y : int)}_{2,\ 3}\ }\ \}}_{plus\ 2\ minus\ 3}$$

Um in der Implementierung des Operators abfragen zu können, welche Klammer wie oft durchlaufen wurde und welchen Wert ein bestimmter Parameter in einem bestimmten Durchlauf hatte, kann MOSTflexiPL-Code in die Klammer-Struktur aus der Signatur geschachtelt werden. Dabei kann die Beziehung zwischen Signatur und Implementierung durch folgende Abbildung dargestellt werden:

Abbildung 4: Beziehung zwischen Signatur- und Implementierungs-Klammern

Würde man die Implementierungs-Klammern nach diesem Bild und dem tatsächlichen Aufruf aus Listing 15 - calc.flx nun expandieren, würde der folgende Code entstehen:

Listing 16: Expandierte Implementierungs-Klammern

```
1 s =! x;          // x = 1
2 s =! ?s + y0;    // y0 = 2
3 s =! ?s - y1     // y1 = 3
```

Für Klammern muss also nicht nur gespeichert werden, ob, wie oft und welche ihrer Alternativen beziehungsweise Optionen durchlaufen wurde, sondern auch mit welchen Werten Parameter in einer jeweiligen Iteration belegt sind, um diese dann in diesem Beispiel für y_0 und y_1 einsetzen zu können.
Da Implementierungs-Klammern selbst auch Ausdrücke sind, evaluiert ihre Ausführung natürlich auch zu einem Ergebnis. So geben Alternativen und Optionen den Index der gewählten Alternative beziehungsweise Option von 1 beginnend zurück. Wiederholungen liefern hingegen die Anzahl ihrer Durchläufe als Ergebnis.

4.6.1 Überblick

Für die Codegenerierung der Klammern und Operatoren sind mehrere Funktionen verantwortlich, deren Zusammenspiel und die Reihenfolge ihrer Aufrufe in diesem Abschnitt besprochen werden.
Die drei zentralen Generierungsfunktionen sind:

`odecl_gen`:
: Diese Funktion wird bei der Erstellung eines Operators aufgerufen. Sie hat nicht nur Zugriff auf seine Signatur, sondern auch auf die Implementierung.

`bracket_gen`:
: Kommen in der Implementierung Klammern vor, wird für diese `bracket_gen` aufgerufen. Achtung: Klammern aus der Signatur eines Operators werden nicht von dieser Funktion behandelt!

`appl_gen`:
: Wird ein Operator, resultiert dieser Aufruf in `appl_gen`. Diese Funktion kennt alle übergebenen Parameter und weiß, welche Klammern wie durchlaufen wurden.

Die Reihenfolge der Aufrufe dieser drei Operationen wirkt auf den ersten Blick offensichtlich. So könnte man annehmen, dass zuerst die Funktion generiert wird (`odecl_gen`), während der Generierung Code für alle Implementierungs-Klammern erzeugt wird (`bracket_gen`) und zuletzt die erzeugte Funktion aufgerufen wird (`appl_gen`). Und auch wenn diese Reihenfolge die in anderen Sprachen wahrscheinlich am häufigsten implementierte ist, wurde hier ein anderer Weg gewählt.

So erzeugt dieses Backend erst Code für einen Operator, wenn dieser tatsächlich aufgerufen wird. Die hier verwendete Reihenfolge ist also Aufruf → Funktions-Generierung → Klammer-Generierung[8]. Dass das Aufrufen und Erzeugen hier verdreht ist, ist bei genauerem Nachdenken allerdings nicht so ungewöhnlich wie auf den ersten Blick. So würde beispielsweise ein Interpreter auch erst die Implementierung einer Funktion abarbeiteten, wenn sie aufgerufen wurde und auch C++'s Templates werden erst instanziiert, wenn sie tatsächlich verwendet werden. Tatsächlich teilen sich diese Implementierung und Templates sehr viele Gemeinsamkeiten. So erzeugt die Instanziierung einer Schablone immer Code, der genau auf die Argumente der Instanziierung beziehungsweise des Aufrufs abgestimmt ist. Für verschiedene Instanziierungen eines Templates werden dann in voller Konsequenz auch mehrere Objekte oder Funktionen erzeugt. Genau dasselbe Verhalten weißt auch dieses Backend auf. Wird der oben deklarierte calc-Operator mehrmals mit unterschiedlichen Klammer-Durchläufen aufgerufen, wird der zweite Aufruf in einer zweiten calc-Funktion im erzeugten Code resultieren.

Während dieser Ansatz den offensichtlichen Nachteil hat, bei n Aufrufen eines Operators im schlechtesten Fall auch n teils redundante Funktionen zu generieren, hat er allerdings auch mehrere nicht so offensichtliche Vorteile.
Da beim Erzeugen der Funktion nicht nur bekannt ist, welche Klammern wie oft und mit welcher Option beziehungsweise Alternative durchlaufen wurden, sondern auch mit welchen Werten die Parameter belegt wurden, könnte ein Aufruf mit ausschließlich konstanten Argumenten (wie in Listing 15 - calc.flx) komplett zur Übersetzungszeit ausgeführt werden! Selbst wenn nicht jedes einzelne Argument zur Übersetzungszeit konstant ist, können immer noch große Teile des erzeugten Codes für Implementierungs-Klammern wesentlich besser optimiert werden, als das bei einer einzelnen generischen Funktion möglich ist. Detailliertere Gedankengänge und weitere Ideen werden zum Schluss der Arbeit in Unterabschnitt 5.1 - Weitere Optimierung des Zwischencodes behandelt.

Der größte und für diese Arbeit entscheidende Vorteil ist allerdings die wesentlich einfachere Implementierung. Genauer gesagt, die wesentlich einfachere Implementierung von in Klammern geschachtelten Parametern. Da die statische Anzahl der Parameter, sobald Wiederholungs-Klammern verwendet werden, nicht mehr der dynamischen entspricht, ist es also nicht möglich, pro MOSTflexiPL-Parameter einfach einen Funktions-Parameter der Signatur der erzeugten Funktion hinzuzufügen. Im Beispiel von oben gibt es statisch lediglich zwei Parameter (`x` und `y`). Der Aufruf `calc minus 1 plus 2 minus 3` sorgt allerdings dafür, dass durch das zweimalige Durchlaufen der geschweiften Klammer dynamisch drei Parameter nötig sind (`x`, $\texttt{y}_0$ und $\texttt{y}_1$). Da dieser Ansatz also in der Regel nicht möglich ist, wäre ein nächster Gedanke beim Aufrufen

[8] Die Reihenfolge wie die Funktionen selbst aufgerufen werden ändert sich natürlich nicht, da sich die Struktur des ASTs nicht ändert. Es ändert sich eben nur die Reihenfolge der von diesen Funktionen typischerweise ausgeführten Operationen.

der Funktion alle dynamisch benötigten Parameter in einer bestimmten Art und Weise im Speicher abzulegen und in der Funktion bei jeder Abfrage des Parameters in Abhängigkeit seines Namens und des aktuellen Durchlaufs durch eine eventuell umschließende Klammer die Adresse des Wertes im Speicher zu berechnen. Man könnte beispielsweise für jede Parameter enthaltende Klammer eine Tabelle im Speicher ablegen, in die zuerst mit dem Namen des Parameters und dann mit dem Index des aktuellen Durchlaufs durch die Klammer indiziert wird. Während es wohl kein Problem ist den aktuellen Durchlauf herauszufinden ist es eher schwer von dem Namen des Parameters auf einen Index zu schließen. In einer Hochsprache wäre die Lösung dieses Problems trivial. So implementieren wohl alle Standardbibliotheken eine Hash-Map. Damit wäre hier die Abkürzung über C's `hcreate` und `hsearch` sicher einfacher und zeitsparender gewesen. Wie in Unterabschnitt 4.5 - Vordefinierte Operatoren bereits diskutiert wurde, ist das Verwenden von Bibliotheks-Funktionen an dieser Stelle allerdings nicht das Ziel dieser Arbeit, da gezeigt werden soll, dass eine Implementierung auch ausschließlich mit den Mitteln, die die LLVM-IR bietet, möglich ist und durch diese Einschränkung auch gleichzeitig das Entwickeln neuer Ideen gefördert beziehungsweise verlangt wird.
Da die Adressberechnung eines Wertes in Abhängigkeit seines Namens, dem aktuellen Klammer-Durchlauf und den genannten Limitationen also nicht implementierbar scheint, wird eben die Funktion nicht bei ihrer Deklaration erstellt, sondern erst bei ihrem Aufruf und somit ausgenutzt, dass zu diesem Zeitpunkt alle Informationen über Klammern und Parametern bekannt sind.

Die folgende Implementierung ist in vier Teile gegliedert. Der gesamte Prozess startet in Unterunterabschnitt 4.6.2 - Aufsetzten der Funktion. Hier wird gezeigt, was ein Aufruf von `odecl_gen` bewirkt, wie die Signatur der generierten Funktion erstellt wird und wie die passenden Parameter für diese Signatur erstellt werden. Unterunterabschnitt 4.6.3 - Erzeugung der Funktion beschäftigt sich im nächsten Schritt mit dem Körper der erzeugten Funktion. Der wohl interessanteste Teil der Implementierung eines Operators ist das Erzeugen der Implementierungs-Klammern. Das Vorgehen hierfür wird in Unterunterabschnitt 4.6.4 - Erzeugung von Implementierungs-Klammern beschrieben, bevor die Funktion zum Schluss in Unterunterabschnitt 4.6.5 - Aufrufen der erzeugten Funktion aufgerufen wird.

4.6.2 Aufsetzten der Funktion

Dieser Schritt beginnt mit dem Aufruf der Generierungsfunktion `odecl_gen`. Deren Aufgabe ist aufgrund der Vertauschung von Aufruf und Erzeugung sehr simpel. Wird später der Operator aufgerufen, muss zur Generierung der Funktion der vom Parser für diesen Operator erzeugte Ausdruck vorliegen. Da `appl_gen` auf diesen allerdings keinen direkten Zugriff hat, ist es `odecl_gen`'s Aufgabe diesen Ausdruck so abzuspeichern, dass er später wieder gefunden werden kann.

Dazu nutzt dieses Backend die eindeutige ID aus, die jedes Objekt eines offenen Typen bei seiner Erstellung erhält. So ist die ID des Operators sowohl bei seiner Deklaration als auch bei seinem Aufruf vorhanden.

Listing 17: odecl_gen

```
1 Value* odecl_gen(Expr expr) {
2   CH_id_t id = expr(expt_)(opers_)[CH::A](orig_).id;
3   func_utils::available_functions_to_instantiate.try_emplace(id, expr);
4   return flx_value::get(flx_value::nil);
5 }
```

Die Map `available_functions_to_instantiate` aus dem Namensraum `func_utils` ist genau die Datenstruktur, auf die beim Aufruf des Operators später wieder zugegriffen wird, um den zugehörigen Ausdruck zu finden. Welche Elemente hinter `expt_` und `opers_` stecken, ist hier nicht allzu wichtig. Zu beachten ist an Stellen wie dieser allerdings die ID eines Operators immer über sein `orig_` Attribut abzufragen, da dieses, falls der Operator verändert wurde, immer auf den ursprünglichen Operator verweist und somit bei Identitätsoperationen benutzt werden muss. So wird auch später in `appl_gen` die ID wieder über dieses Attribut abgefragt!

Die Generierungsfunktion für den Operator-Aufruf ist wesentlich komplexer als die eben beschriebene. Sie besteht aus insgesamt vier Schritten:

1. Erzeugen der Funktions-Argumente
2. Generieren der Funktion und erzeugen von Kontext-Strukturen
3. Funktions-Argumenten mithilfe der erzeugten Kontext-Strukturen vervollständigen
4. Funktion aufrufen

Da Schritt zwei in seine eigene Funktion ausgelagert ist und erst im nächsten Abschnitt beschrieben wird, besteht der Hauptaufwand `appl_gen`s also in Schritt eins.

Für die Erzeugung der Funktions-Argumente und vor allem auf für die spätere Generierung der Funktion müssen mehrere Datenstrukturen aufgebaut werden. Um diese zu verstehen, muss zuerst geklärt werden, welchen Aufbau die generierte Signatur haben wird, auch wenn das eigentliche Generieren der Signatur in `appl_gen` noch gar nicht geschieht.

Hierzu wird der Kopf des oben in Listing 15 gezeigten calc-Operators zusammen mit seinem Aufruf aus demselben Listing betrachtet:

```
1 calc [minus] (x:int) { (plus|minus) (y:int) }
2 ...
3 calc 1 plus 2 minus 3
```

Dieser Aufruf wird in folgenden Parametern resultieren:

Listing 18: Beispiel Funktions-Signatur

```
1 %bracket* %0, i35* %x.ptr.0, %bracket* %1, i35* %y.ptr.0, i35* %y.ptr.1
```

Um diese Parameter-Liste zu erzeugen, wird eine Mischung aus den in Unterunterabschnitt 4.6.1 - Überblick beschriebenen Möglichkeiten verwendet. So werden sowohl MOSTflexiPL-Parameter als auch Klammern auf höchster Ebene direkt zu Funktions-Parametern umgesetzt. Für geschachtelte Klammern wird eine Baumstruktur an `bracket`-Objekten erzeugt, deren Wurzel zwingend ein Top-Level-Parameter ist, welcher wiederum direkt in der Parameter-Liste zu finden ist. Für in Klammern geschachtelte Parameter wird allerdings ein anderes Vorgehen gewählt. Da beim Aufruf des Operators bekannt ist, ob und wie oft Klammern mit Parametern durchlaufen wurden, kann hier für jeden dynamisch existierenden Parameter ein neuer Parameter in die Signatur der Funktion geschrieben werden. So würden hier die Parameter `%y.ptr.0` und `%y.ptr.1` beim Aufruf der Funktion mit den Werten `1` und `2` belegt werden.

Um nun die korrekten Argumente für die Übergabe an die erzeugte Funktion zu erstellen, verwaltet `appl_gen` drei Listen:

```
1 std::vector<AllocaInst*> oper_brackets;
2 std::vector<AllocaInst*> oper_params;
3 std::vector<func_utils::AllocaBracket> abrackets;
```

Während die ersten beiden Listen Klammern beziehungsweise Parameter auf höchster Ebene enthalten[9], enthält die Dritte Objekte des Typs `AllocaBracket`, um geschachtelte Klammern darzustellen. Hierbei ist die Unterscheidung zwischen der Struktur, die sich der Compiler selbst aufbaut und der, die später während der Laufzeit des erzeugten Programms aufgebaut wird wichtig. Während `oper_brackets` und `oper_params` Repräsentationen von Werten enthalten, die erst zur Laufzeit existieren werden, enthält `abrackets` Objekte, die der Compiler für die eigene Verwendung erstellt und die zur Laufzeit nicht existieren.
Die `AllocaBracket`-Klasse enthält neben der ID der Klammer, der Anzahl der möglichen Alternativen und der `alloca`-Instruktion, die das zugehörige `bracket`-Objekt im Speicher angelegt

[9] `AllocaInst` ist dabei eine Unterklasse von der bereits bekannten Klasse `Value` aus der LLVM-Bibliothek.

hat, auch eine Liste an allen Durchläufen durch die Klammer. Ein Durchlauf (Typ `AllocaRun`) speichert die tatsächlich durchlaufene Option, eine Liste an geschachtelten Klammern und eine Liste an, in diesem Durchlauf, enthaltenen Parameter. Beide Listen enthalten ihre Elemente jeweils in der Reihenfolge, in der sie im Quelltext vorkommen.
Symmetrisch zu diesen beiden Klassen werden später noch `ContextBracket` und `ContextRun` aufkommen. Diese repräsentieren paarweise logisch dieselben Dinge, allerdings für eine andere Stufe in der Funktions-Erzeugung. Wenn es in Unterunterabschnitt 4.6.3 - Erzeugung der Funktion um das Aufstellen von Kontexten geht, werden die hier erzeugten `AllocaBracket`- und `AllocaRun`-Objekte nach und nach in `ContextBracket`- und `ContextRun`-Objekte überführt. Da das Erzeugen der Argumente und das Generieren der Funktion jeweils sehr unterschiedliche Informationen über Klammern und ihre Durchläufe benötigen, ist somit auch durch die Verwendung unterschiedlicher Klassen eine klare Grenze zwischen den Schritten geschaffen.

Für obiges Beispiel (calc-Operator) würde `abrackets` am Ende von `appl_gen` folgende Objekte enthalten:

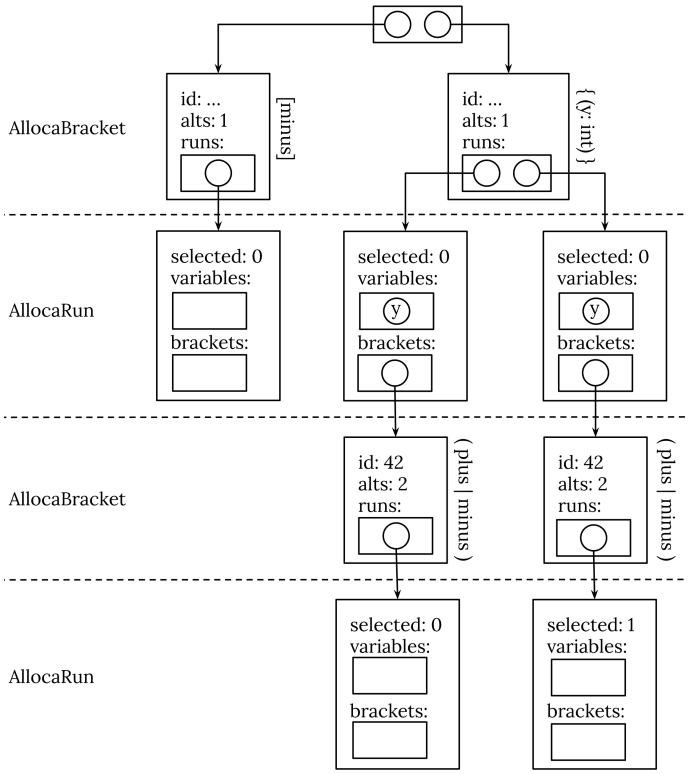


Abbildung 5: Von `appl_gen` erzeugte Klammer-Struktur

Achtung: Der Parameter-Name `y` steht nicht wirklich in der Liste an Variablen, da diese Namen in `appl_gen` noch nicht bekannt sind. Anstelle der Namen stehen dort Objekte des Typs `AllocaInst*`!

An dieser Baumstruktur kann man gut die Art und Weise wie alle Klammern durchlaufen worden sind erkennen. So hat die Wiederholung (oben rechts) klar erkennbar zwei Durchläufe, in denen jeweils einmal die Alternative durchlaufen wurde. In der untersten Reihe erkennt man, dass die Alternative beim ersten Mal mit ihrer ersten Möglichkeit (Index 0) und das zweite Mal mit der zweiten Möglichkeit (Index 1) durchlaufen wurde.
Ein wichtiges Detail, das diese Abbildung nur andeutet, dreht sich um die Identität der verschiedenen Klammern. Die beiden Klammern auf der ersten Ebene können nie dieselben sein, da logischerweise zwei Top-Level-Klammern immer unterschiedliche sind. Die beiden Klammern auf der dritten Ebene sind allerdings schon dieselben. Auch wenn sie durch zwei verschiedene Objekte repräsentiert werden, weißt die identische ID darauf hin, dass sie aus derselben MOSTflexiPL-Klammer entstanden sind. Trotzdem ist es wohlgemerkt möglich, dass beide Objekt eine unterschiedliche Anzahl an Durchläufen besitzen! Wenn später über das Erzeugen von Implementierungs-Klammern gesprochen wird, wird das Finden von Klammern mit demselben Ursprung noch einmal relevant werden.

Während nun nach und nach diese Struktur im Speicher des Compilers erstellt wird, werden im selben Algorithmus parallel LLVM-IR-Instruktionen emittiert, die später äquivalent verschachtelte `bracket`-Objekte im Speicher des laufenden Programms erzeugen werden. Die aus diesem Aufbau resultierenden Wurzel-Knoten werden dann beim Aufruf des Operators als Argumente übergeben. Im Beispiel aus Listing 18 - Beispiel Funktions-Signatur sind das die Argumente `%0` und `%1` an den Positionen null und zwei in der Parameter-Liste.
Der in LLVM-IR implementierte Typ `bracket` ist wie folgt definiert:

```
1 %bracket = type { i32*, i32, i1, %bracket***}
```

Da dieser Typ für alle Klammer-Arten verwendet wird, werden bestimmte Attribute für bestimmte Klammern nie gesetzt, da sie auch nie abgefragt werden. Die beiden Zeigertypen müssen hier als Listen interpretiert werden. So verbirgt sich hinter `i32*` eine Liste an Indizes der gewählten Option pro Durchlauf. Diese Liste muss zwingend dieselbe Länge haben wie `bracket***`, da hier in der ersten Dimension alle Durchläufe und in der zweiten alle in diesem Durchlauf durchlaufenen Klammern gespeichert sind. Die zweite Dimension dieser Liste repräsentiert hier also dieselben Daten, wie das `brackets` Feld der C++-Klasse `AllocaRun`. Die beiden verbleibenden Elemente `i32` und `i1` enthalten die Anzahl der Durchläufe, falls eine Wiederholungs-Klammer vorliegt, und einen booleschen Wert, ob die Klammer durchlaufen wurde, falls eine eckige Klammer vorliegt.
Die folgenden Code-Ausschnitte zeigen exemplarisch das Emittieren von Befehlen zur Erstellung einer Wiederholungs-Klammer. Die Erzeugung für Optionen und Alternativen folgt prinzipiell demselben Vorgehen. Der einzige Unterschied ist, dass beide nur (maximal) einen Durchlauf beachten müssen und dadurch etwas simpler in der Implementierung sind. Der hier gezeigt Code

liegt in einer zu `appl_gen` lokalen Funktion, die über eine `for`-Schleife sequenziell alle Klammern und Parameter auf der aktuellen Ebene abarbeitet. Wird in dieser Schleife eine geschweifte Klammer erkannt, wird also folgender Code ausgeführt:

Listing 19: Erstellung eines `bracket`-Objekts #1

```
1  auto generate_call_args = [&](CH::seq<Item> items, CH::seq<Part> parts)->Value* {
2  // ...
3  AllocaInst* bracket_ptr = ir_builder->CreateAlloca(predef_structures::bracket());
4  utils::insert_into(bracket_ptr, *item(passes_), 0, 1);
5  Value* selected_option_ptr = nullptr;
6  Value* runs_ptr = nullptr;
7  if(*item(passes_) > 0) {
8    runs_ptr = ir_builder->CreateAlloca(utils::get_ptr_type(predef_structures::
       ↪ bracket(), 2), utils::get_integer_constant(*item(passes_));
9    // Platz für *item(passes_) viele Elemente reservieren
10   selected_option_ptr = ir_builder->CreateAlloca(utils::get_integer_type(), utils
       ↪ ::get_integer_constant(*item(passes_)));
11 }
```

Im ersten Schritt wird das zu füllende Klammer-Objekt erstellt und direkt im Anschluss mit der Anzahl an Durchläufen gefüllt. Hierzu wird die in Unterunterabschnitt 4.3.1 - Abstraktionen über LLVM beschriebene Funktion `insert_into` verwendet, um die gegebene Information an Index `1` in das Objekt einzutragen. Neben der Anzahl der Durchläufe werden zusätzlich die beiden Listen `i32*` und `bracket***` benötigt. Da für diese kein Speicher reserviert werden muss, falls die Klammer nicht durchlaufen wurde, werden die nötigen Variablen hier nur mit der zugehörigen `alloca`-Instruktion initialisiert, falls tatsächlich Durchläufe vorliegen.
Die folgende Schleife ist der wichtigste Teil dieses Ausschnitts.

Listing 20: Erstellung eines `bracket`-Objekts #2

```
1  for(int j = 0; j < *item(passes_); ++j) {
2    Pass pass = passes[CH::A + j];
3    current_abracket->runs.emplace_back(pass(choice_) - CH::A);
4    utils::insert_into(selected_option_ptr, pass(choice_) - CH::A, j);
5    Value* brackets = generate_call_args(pass(branch_),part(alts_)[pass(choice_)]);
6    if(brackets != nullptr)
7      utils::insert_into(runs_ptr, brackets, j);
8  }
```

Hier wird für jeden Durchlauf rekursiv die in ihm enthaltene Klammer-Struktur aufgebaut und das Resultat passend in die aktuelle Klammer eingefügt.
Zuerst wird allerdings die in diesem Lauf ausgewählte Option gesichert. Achtung: `current_abracket` hat nichts mit dem LLVM-Typ `bracket` zu tun. Diese Variable wird für den Aufbau der

in Abbildung 5 - Von `appl_gen` erzeugte Klammer-Struktur beschriebenen Struktur verwendet, die der Compiler selbst verwendet. Der Aufruf zu `insert_into` für die Klammer, die im Speicher des laufenden Programms liegen wird, folgt erst in der nächsten Zeile. Daraufhin wird, wie eingangs beschrieben, alle inneren Klammern des aktuellen Durchlaufs rekursiv erstellt. Dieser Aufruf liefert als Ergebnis eine eindimensionale Liste, die an passender Stelle (Index `j`) in die Liste aller Durchläufen eingefügt werden muss, um die benötigte zweidimensionale Struktur zu erstellen. Dieses Einfügen passiert allerdings nur, falls der aktuelle Durchlauf tatsächlich geschachtelte Klammern besessen hat.

Listing 21: Erstellung eines `bracket`-Objekts #3

```
1 if(*passes > 0) {
2   utils::insert_into(bracket_ptr, selected_option_ptr, 0, 0);
3   utils::insert_into(bracket_ptr, runs_ptr, 0, 3);
4 }
```

Zum Schluss werden die in der Schleife erstellten Listen in die Klammer eingefügt und somit ist das Objekt mit allen nötigen Informationen gefüllt.
Am Ende einer Iteration der eingangs genannten Schleife über alle Klammern und Parameter einer Ebene, muss das eben erstellte `bracket`-Objekt selbst noch einer Liste, die alle `bracket`-Objekte dieser Ebene speichert (`bracket_ptrs`), hinzugefügt werden. Am Ende aller Iterationen wird dann mit ...

```
1 Value* all_brackets = ir_builder->CreateAlloca(utils::get_ptr_type(
      ↪ predef_structures::bracket()), utils::get_integer_constant(bracket_ptrs.
      ↪ size()));
2 for(int i = 0; i < bracket_ptrs.size(); ++i)
3   utils::insert_into(all_brackets, bracket_ptrs.at(i), i);
4 return brackets;
```

... diese Liste in eine zur Laufzeit existierende Liste umgewandelt, die dann auch als Ergebnis der lokalen Funktion zurückgeliefert werden kann.

Durch diesen Algorithmus werden während der Laufzeit des erzeugten Programms für den Aufruf der Wiederholungs-Klammer des calc-Operators diese Objekte erstellen:

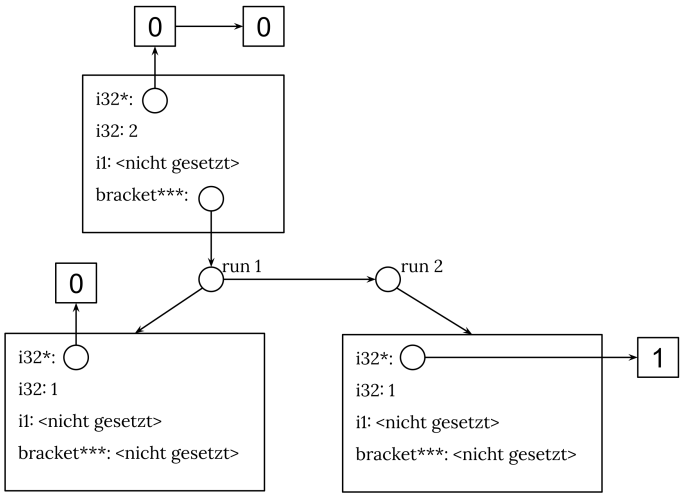


Abbildung 6: Während der Laufzeit erzeugte Klammer-Struktur

Das Objekt hinter der obersten Box wird beim Aufruf des Operators an den Parameter `%1` übergeben werden.

Ein weiterer recht simpler Schritt, der hier noch stattfinden muss, ist das Erstellen der Argumente für alle Parameter. Hierzu wird in derselben Schleife, in der auch Klammern erzeugt wurden, für jeden Parameter eine `alloca`-Instruktion erstellt, die für einen Top-Level-Parameter in die Liste `oper_args` eingetragen wird oder andernfalls der Liste an Variablen des aktuellen `AllocaRun`-Objekts hinzugefügt wird. Achtung: auch wenn alle Parameter unabhängig von ihrer Tiefe in Klammern direkt in die Signatur der Funktion geschrieben werden, wird hier noch die geschachtelte Struktur beibehalten, da diese für das spätere Erzeugen von Kontexten noch relevant sein wird!

Da nun die Klammerstrukturen für die Übersetzungs- und Laufzeit aufgebaut sind, kann der nächste Schritt stattfinden: das Erzeugen der eigentlichen Funktion. Da jede Funktion einen eigenen Kontext erhält, muss dieser vor der Generierung hier noch erstellt werden. Dieser neue Kontext bekommt als Parent-Kontext den derzeit aktuellen gesetzt und wird anschließend als neuer aktueller Kontext gespeichert.

4.6.3 Erzeugung der Funktion

Für das Erzeugen der Funktion ist die Hilfsfunktion `instantiate_function` zuständig. Diese benötigt insgesamt drei Parameter:

- Die ID des Operators, der den MOSTflexiPL-Operator darstellt
- Die von `appl_gen` aufgebaute Klammer-Struktur (s. Abbildung 5 - Von `appl_gen` erzeugte Klammer-Struktur)
- Eine Liste an `ContextBracket`-Objekten als Referenz-Parameter

Letzterer wird von `instantiate_function` nicht gelesen, sondern nach und nach beschrieben. Wie zu Beginn des letzten Kapitels bereits kurz angerissen, repräsentiert die Klasse `ContextBracket` dasselbe wie die Klasse `AllocaBracket`. Ergo hat auch die von `instantiate_function` erzeugte Struktur einen äquivalenten Aufbau wie `abrackets`. Trotz dieser Ähnlichkeit ist der Inhalt der einzelnen Knoten aber teilweise ein anderer. So enthält eine `ContextBracket` weiterhin die ID der zugehörigen Klammer und einen `Value`-Zeiger auf die, die Klammer erstellende Instruktion. Für Klammern auf höchster Ebene wird zusätzlich der Wert `index` gesetzt. Dieser enthält den Index der Klammer in der Parameter-Liste der erzeugten Funktion. Anstelle einer Liste an `AllocaRun`s werden alle Durchläufe hier in einer Liste an `ContextRun`s gespeichert. Während in einem `AllocaRun`-Objekt alle Parameter noch in einer Liste an `alloca`-Instruktionen gespeichert waren, enthält ein Durchlauf nun einen richtigen Kontext, der neben den Instruktionen nun unter anderem auch die Namen der Parameter enthält. Das einzige wirklich neue Datenelement in dieser Klasse ist `ascend`. Dieser boolesche Wert ist in diesem Beispiel lediglich für die eckige Klammer (`[minus]`) gesetzt, denn ausschließlich diese Klammern können die damit abgebildete Eigenschaft besitzen. Betrachtet man folgenden Operator:

```
1 oper [abc (a: int)] {def [(b: int)]} -> (int = ...)
```

So würde man annehmen, dass die Parameter `a` und `b` ausschließlich in ihren jeweiligen Implementierungs-Klammern sichtbar sind. Grundsätzlich gilt diese Regel auch, allerdings wird für eckige Klammern mit nur einer Option eine Ausnahme gemacht, sodass die beiden Parameter auch in ihrem umschließenden Kontext sichtbar sind. Um auf `a` zuzugreifen, muss also keine Implementierungs-Klammer verwendet werden und `b` kann auch in der umschließende Wiederholung verwendet werden. Wird die Option beim Aufruf des Operators nicht durchlaufen, werden die Parameter mit dem Wert `nil` belegt. Um nun genau dieses Verhalten zu ermöglichen, wird bei Options-Klammern mit einer Option dieses Flag gesetzt, sodass ihr Kontext später in den Umschließenden verschoben werden kann.

All die genannten Eigenschaften sind in der erzeugten Baum-Struktur wie folgt gesetzt:

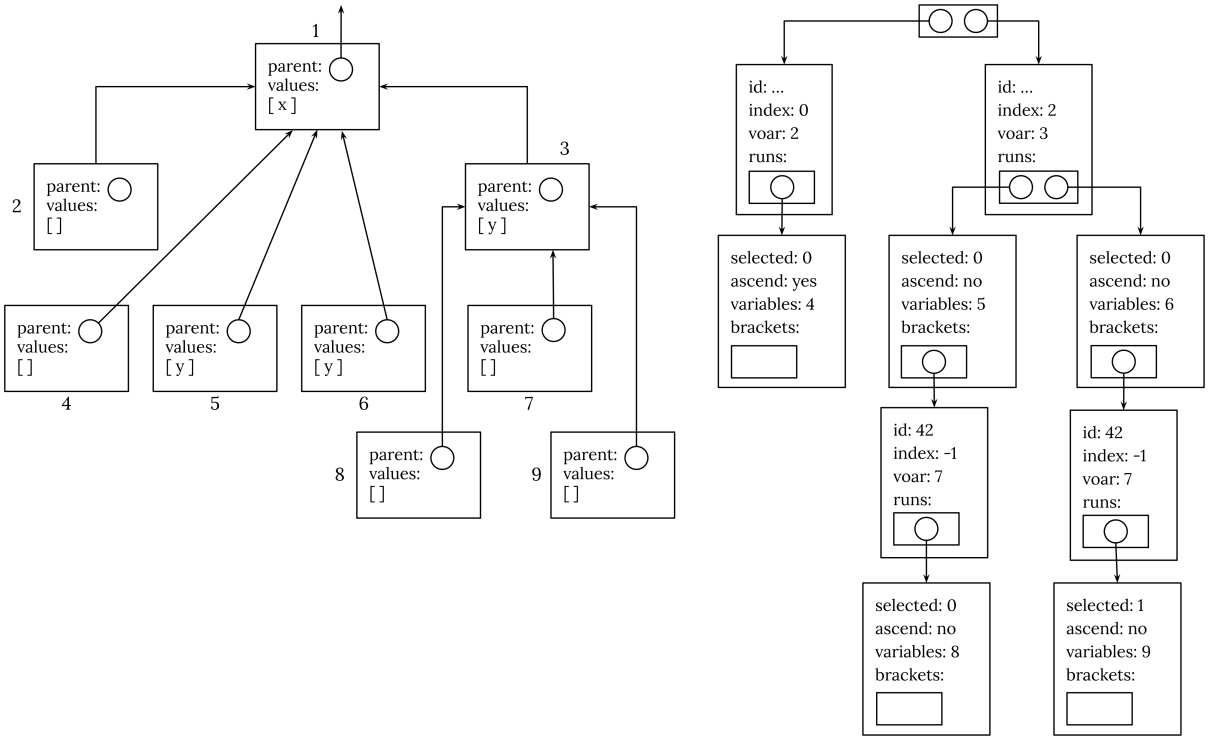


Abbildung 7: Von `instantiate_function` erzeugte Klammer-Struktur

Links werden hier zusätzlich alle erzeugten Kontexte und ihr Verhältnis untereinander dargestellt. Verwendet ein Knoten der rechten Struktur einen dieser Kontexte, wird dieser durch seine Nummer referenziert.
Schnell fällt hier auf, dass eine Klammer noch das etwas kryptisch benannte Attribut voar besitzt. Dieses Akronym steht hier für „**V**alues **o**ver **a**ll **r**uns“ und ist ebenfalls ein Kontext. Dieser wird vor allem bei Wiederholungs-Klammern relevant, da in ihm zu jedem Parameter, der in einem beliebigen Durchlauf der Klammer vorkommt, ein zusätzlicher Wert abgespeichert ist, der immer mit dem korrekten Wert des aktuellen Durchlaufs belegt ist. Betrachtet man folgendes Beispiel:

```
1 oper { (a (a: int) | b (b: int)) } -> (int ...);
2 oper a 1 a 2 b 3
```

So wird hier der voar-Kontext der Alternative zwei Einträge enthalten, `a` und `b`. Der Wert beider Einträge wird standardmäßig auf den Wert des ersten Vorkommnisses gesetzt. Somit ist also `a` mit `1` und `b` mit `3` belegt. Nun werden für jede Iteration der Wiederholung die Werte überschrieben. Für die erste bleibt der Wert von `a` bestehen und `b` wird auf nil gesetzt. In der zweiten Iteration wird `a 2` annehmen und `b 3`. Wie genau das Abändern des Wertes während

der Laufzeit ohne das Vorhandensein der Kontexte passiert, wird in Unterunterabschnitt 4.6.4 - Erzeugung von Implementierungs-Klammern behandelt.

Die Hauptaufgabe von `instantiate_function` ist nun im ersten Schritt die von `appl_gen` übergebene Struktur in die in Abbildung 7 beschriebene zu überführen, parallel alle nötigen Kontexte und ihre Beziehung untereinander aufzubauen und die Signatur der Funktion zu erzeugen. Hierzu wird wieder das Erstellen einer Wiederholungs-Klammer und eines Parameters betrachtet.

Listing 22: Erstellen eines `ContextBracket`-Objekts #1

```
1 auto gen_function_signatur = [&](CH::seq<Pass> passes, ValueContext*
      ↪ parent_context) -> void {
2 // ...
3 int index = -1;
4 if(level == 0)
5   index = new_param(utils::get_ptr_type(predef_structures::bracket()));
6 bool has_pass = setup_cbracket(bracket_index, index, parent_context);
7 if(!has_pass)
8   break;
9 dynamic_function_name << "r" << current_abracket->runs.size();
```

Wie bereits in `appl_gen` befindet sich der Code für die Erzeugung in einer lokalen Hilfsfunktion, die mithilfe einer `for`-Schleife über alle Elemente der aktuellen Ebene iteriert und für eine Wiederholungs-Klammer den dargestellten Code ausführt.

Alles beginnt damit, dass ein neuer Funktions-Parameter mit `new_param` in die Parameter-Liste geschrieben wird, falls es sich um eine Top-Level-Klammer handelt. `setup_cbracket` wird als Nächstes ein neues `ContextBracket`-Objekt erstellen und es an geeigneter Stelle in die gesamte Struktur einfügen. Zusätzlich sorgt diese Funktion dafür, dass der voar-Zeiger für Klammern mit derselben ID immer auch auf denselben Kontext verweist. Ist es das erste Mal, dass `setup_cbracket` eine bestimmte Klammer-ID sieht, muss ein neuer voar-Kontext erstellt werden, der den übergebenen Kontext als Parent-Kontext gesetzt bekommt. Zum Schluss dieser Hilfsfunktion wird genau dann `false` zurückgegeben, falls die gerade erstellte Klammer keinen Durchlauf hat. Ist das der Fall, wird in Zeile neun abgebrochen und die umschließende Schleife wird mit dem nächsten Element fortfahren. Daraus folgt, dass auch für eine nicht durchlaufene Klammer ein `ContextBracket`-Objekt erstellt wird! Dies gilt ebenfalls für `abrackets` und dir Struktur, die zur Laufzeit aufgebaut wird.
Der letzte Schritt vor der Erstellung der einzelnen Durchläufe ist das Erzeugen eines eindeutigen Namens. Diesen Namen wird später die Funktion erhalten und anhand dieses Namens wird auch bestimmt werden, ob für einen bestimmten Aufruf eine neue Funktion instanziiert werden muss oder ob eine bereits existierende wiederverwendet werden kann. Der erzeugte Name muss

also Informationen darüber enthalten, welche Klammer wie oft durchlaufen wurde und welche Option beim Durchlaufen jeweils gewählt wurde. Keine Rolle spielt es allerdings, mit welchen Werten Parameter belegt wurden! Theoretisch würde es auch keine Rolle spielen, welche Option bei einem Durchlauf gewählt wurde, da Implementierungs-Klammern unabhängig der ausgewählten Option generiert werden. In Unterabschnitt 4.8 - Optimierung des Zwischencodes wird es allerdings von Bedeutung sein, dass diese Information im Namen enthalten ist.
Namen werden nach folgendem Schema erzeugt:

- Durchlaufene Wiederholung → „r“ + Anzahl der Durchläufe
- Durchlaufene Option → „o“ + Anzahl der Durchläufe (Immer 1)
- Durchlaufene Alternative → „a“ + Anzahl der Durchläufe (Immer 1)
- Pro Durchlauf → Ausgewählte Alternative
- Pro Parameter → „p“

Zusätzlich wird jeder Name mit der ID des Operators gepräfixt, sodass der Aufruf des calc-Operators aus Listing 15 `0x2f6a030o10pr20a10p0a11p` als Namen erzeugt[10].

Nun folgt das Abarbeiten aller Durchläufe der Wiederholungs-Klammer.

Listing 23: Erstellen eines `ContextBracket`-Objekts #2

```
1  for(int i = 0; i < current_abracket->runs.size(); ++i) {
2    setup_crun(i, parent_context);
3    dynamic_function_name << current_arun->selected_option;
4    if(int selected_option = current_arun->selected_option; selected_option == 0)
5      gen_function_signatur(pass(branch_)[CH::A + 2](opnd_)(row_)[CH::A](passes_),
           ↪ ccurrent_cbracket->voar);
6    else {
7      gen_function_signatur(pass(branch_)[CH::A + 3](passes_)[CH::A +
           ↪ selected_option - 1](branch_)[CH::A + 1](opnd_)(row_)[CH::A](passes_),
           ↪  current_cbracket->voar);
8    }
9  }
```

Zu Beginn wird mit `setup_crun` ein neues `ContextRun`-Objekt erstellt und in die Liste aller Durchläufe der aktuellen Klammer eingefügt. `setup_crun` kümmert sich auch um das korrekte Setzen des Parent-Kontexs für den neu angelegten Durchlauf. Falls bereits ein Durchlauf existiert, wird der Parent-Kontext des ersten Durchlaufs gewählt, andernfalls der an `setup_crun` übergebene. Dies führt dazu, dass alle Laufe immer denselben Eltern-Kontext besitzen. Als letz-

[10] `0x2f6a030` ist dabei die ID und `o10pr20a10p0a11p` der den Aufruf beschreibende Teil.

tes optionales Argument kann das weiter oben angesprochene `ascend` Flag übergeben werden. Dieses wird im Falle einer optionalen Klammer folgendermaßen gesetzt:

```
1 setup_crun(0, parent_context, current_abracket->alts == 1);
```

Da alle anderen Klammer-Typen dieses Flag nicht verwenden, ist es standardmäßig auf `false` gesetzt.
Im letzten Schritt wird, wie auch schon in `appl_gen`, die lokale Funktion für den Inhalt der aktuellen Klammer rekursiv aufgerufen. Die den rekursiven Aufruf umschließende Verzweigung dient lediglich dazu, das korrekte Element des ASTs auszuwählen.

Auch in dieser Funktion läuft das Erzeugen der anderen beiden Klammer-Typen etwas simpler ab, da auch hier nur (maximal) ein Durchlauf betrachtet werden muss.

Das Erstellen eines Parameters zeigt nun zum Schluss noch, welche Elemente der Kontext einem Parameter-Namen zuordnet.

Listing 24: Erstellen eines Parameters

```
1  param_name << pass(branch_)[CH::A + 1](passes_)[CH::A](branch_)[CH::A](passes_)[
       ↪ CH::A](branch_)[CH::A](word_);
2  int index = new_param(utils::get_ptr_type(utils::get_integer_type(35)));
3  if(level == 0)
4    top_level_param_context.named_values.try_emplace(parameter_name.str(), index);
5  else {
6    AllocaInst* alloca = current_arun->variables.at(param_index_per_selected_option
       ↪ [current_arun->selected_option]);
7    current_crun->variables->named_values.try_emplace(param_name.str(), index,
       ↪ alloca);
8    if(!current_cbracket->voar->named_values.contains(param_name.str()))
9      current_cbracket->voar->named_values.try_emplace(param_name.str(), index);
10 }
```

Zeile zwei zeigt das, was bereits mehrmals im Text erwähnt wurde, nun auch im Programm-Code. So wird `new_param` unabhängig der aktuellen Tiefe in Klammern aufgerufen und somit wird für einen MOSTflexiPL-Parameter eben immer ein Platz in der Parameter-Liste der Funktion reserviert. Lediglich der Kontext, in den der Parameter eingetragen wird, ist von der Tiefe abhängig. So gibt es für Top-Level-Parameter einen eigenen Kontext. Alle anderen werden sowohl in den Kontext des aktuellen Durchlaufs (Zeile sieben) als auch in den voar-Kontext eingetragen (Zeile neun). In den Kontext des Durchlaufs wird neben dem Namen auch der Index, an dem der Parameter in der Signatur der Funktion steht, geschrieben. Einträge des voar-Kontexts enthalten diesen Index auch. Dabei ist dies immer der Index des ersten Vorkommnisses des Parameters. So würde der voar-Kontext der Wiederholung aus Listing 15 - calc.flx für den Parameter `y` den Index

drei enthalten. Dieser wird später genutzt, um den Wert des voar-Parameters zu initialisieren. Der Index in einem normalen Kontext wird hingegen später unter anderem in `appl_gen` dazu verwendet, das zugehörige Argument an der richtigen Stelle zu übergeben. Um nun neben dem richtigen Index auch das richtige Argument für den späteren Aufruf der Funktion in `appl_gen` zu finden, wird dieses aus der vorher aufgestellten Struktur `abrackets` in Zeile sechs extrahiert und danach ebenfalls dem Kontext hinzugefügt. Somit findet `appl_gen` nach dem Terminieren von `instantiate_function` alle nötigen Informationen auch in `cbrackets` wieder und wird damit `abrackets` nicht mehr verwenden. Letztere Struktur dient also nur als Hilfe zum Aufbauen ersterer.

Der erste Schritt nach dem Terminieren der lokalen Unterfunktion ist das Durchsuchen des LLVM-Moduls nach dem erzeugten Namen. Wird dabei keine Funktion gefunden, wurde der Operator also noch nie mit diesem bestimmten Durchlaufs-Muster aufgerufen und die Funktion muss somit neu erstellt werden. Dazu wird als Rückgabe-Typ immer ein 35 Bit breiter Integer gewählt. Die Parameter-Liste entsteht aus der von Aufrufen zu `new_param` erstellten Liste.
Liest man Listing 24 - Erstellen eines Parameters noch einmal aufmerksam, fällt auf, dass dort an keiner Stelle der tatsächliche Wert des Parameters (Attribut `value`) gesetzt wird. Für den Wert eines MOSTflexiPL-Parameters soll der passende Parameter aus der Signatur der Funktion gewählt werden. Da die Funktion bis eben aber noch nicht existiert hat, konnte auch noch kein Parameter aus ihrer Signatur ausgewählt und in den Kontext eingetragen werden. Da dies nun möglich ist, wird über alle Klammern und alle deren Durchläufe aus `cbrackets` iteriert. Dabei werden für jeden Lauf alle in ihm abgespeicherten Variablen und vor allem deren gespeicherter Index abgefragt. Zur Erinnerung: dieser Index ist genau die Stelle, an der der gewünschte Parameter in der Funktion zu finden ist. Ergo kann nun mit ...

```
1 generated_function->getArg(index);
```

... der Parameter abgefragt und anschließend in den Kontext eingetragen werden. Dieses Vorgehen wird für alle in einem Durchlauf geschachtelten Klammern und später auch für den Kontext an Top-Level-Parametern wiederholt, sodass am Ende jedem MOSTflexiPL-Parameter ein Funktions-Parameter zugeordnet wurde.
Achtung: Hat die Funktion bereits existiert, muss der Index nicht mit dem Parameter rücksubstituiert werden, da dies nur für das Erzeugen der Implementierung von Bedeutung ist, dass ein korrekter Wert gesetzt ist. `appl_gen` wird später nur noch auf den Index und das zu dem Parameter aus `abrackets` übernommene Argument zugreifen. Tatsächlich werden alle folgende Änderungen an `cbrackets` ausschließlich ausgeführt, falls die Funktion nicht bereits existiert.

Der letzte größere Schritt vor dem Generieren der Implementierung besteht in der Verarbeitung des `ascend` Flags. Hierzu werden wieder alle Klammern aus `cbrackets` und deren jeweilige Durchläufe durchgegangen. Für jeden einzelnen Durchlauf wird dann folgender Code ausgeführt:

```
1  auto ascend_contexts = [&](std::vector<ContextBracket>& cbrackets) {
2  // ...
3  if(run.ascend_in_contexts) {
4    encl_run->named_values.insert(run.variables->named_values.begin(), run.
       ↪ variables->named_values.end());
5    encl_voar->named_values.insert(bracket.voar->named_values.begin(), bracket.voar
       ↪ ->named_values.end());
6    run.variables->named_values.clear();
7  }
8  encl_run = run.variables;
9  encl_voar = bracket.values_over_all_runs;
10 ascend_contexts(run.brackets);
```

Hier wird, falls das Flag denn gesetzt ist, in Zeile zwei und drei jeweils der umschließende Durchlaufs- und voar-Kontext mit den Elementen des Kontexts der aktuellen eckigen Klammer erweitert. Hierbei sollte es nie vorkommen, dass bereits existierende Parameter in den umschließenden Kontexten überschrieben werden, da der Parser bei einer solchen Situation abbrechen würde. Es kann also keine Situation entstehen, in der sowohl die Option als auch ihr umschließender Kontext einen Parameter namens `a` besitzen, da ein Zugriff auf `a` nun mehrdeutig wäre. Da nun alle Parameter in ihre Zielkontexte verschoben wurden, kann der Kontext der Option gelöscht werden. Dabei bleiben alle Parameter natürlich auch in einer Implementierungs-Options-Klammer über den `parent`-Kontext sichtbar! Im letzten Schritt wird die Hilfsfunktion rekursiv für alle im aktuellen Durchlauf enthaltenen Klammern aufgerufen, wobei vorher die Zeiger auf die beiden umschließenden Kontexte korrekt gesetzt werden müssen.

Bevor nun mit der Generierung der Implementierung begonnen werden kann, werden in den aktuellen Kontext noch alle Elemente aus dem Kontext an Top-Level-Parametern eingefügt, damit auch diese sichtbar sind. Nun wird das erste Mal Code in den Körper der Funktion eingefügt.

Dabei wird, wie auch bei allen anderen bisher besprochenen LLVM-IR-Funktionen, zuerst ein neuer basic Block erstellt, in den der komplette Code geschrieben wird. Der erste LLVM-IR-Code der in ihm stehen wird initialisiert alle Werte aus allen voar-Kontexten. Zur Erinnerung: in diese wurde bisher nur der Name und ein Index `i` geschrieben. Nun wird auch der tatsächliche Wert des Parameters eingefügt. Hierfür wird durch einen `alloca`-Befehl Speicher reserviert, in den der geladene Wert des `i`-ten Parameters gespeichert wird. Zum Schluss wird die `alloca`-Instruktio als Wert des Parameters im Kontext abgelegt.

Nun kann der Teil des ASTs abgearbeitet werden, der die Implementierung des Operators enthält. Hierfür wird `instantiate_function` die Funktion `gen_expr_code_internal` aufrufen. Dabei wird es wohl so gut wie immer vorkommen, dass der Operator Implementierungs-Klammern in seinem Körper enthält. Wie LLVM-IR-Code für diese erzeugt wird, zeigt das folgende Kapitel.

4.6.4 Erzeugung von Implementierungs-Klammern

Die Idee hinter der Code-Erzeugung für die drei verschiedenen Arten an Implementierungs-Klammern ist, im Gegensatz zu ihrer Implementierung, recht simpel. So könnte man folgenden Pseudocode formulieren:

Listing 25: Pseudocode einer runden Implementierungs-Klammer

```
1 selected_alternative = current_bracket.runs[0].selected_alternative
2 switch(selected_alternative)
3   case 0: // Code für die erste Alternative erzeugen
4   case 1: // Code für die zweite Alternative erzeugen
5   ...
6 end
```

Alternativen haben den einfachsten Aufbau. Dadurch, dass sie immer einmal durchlaufen werden, reicht es, die für diesen einen Durchlauf gewählte Alternative abzufragen und je nach Resultat zu einem anderen Block zu springen.

Listing 26: Pseudocode einer eckigen Implementierungs-Klammer

```
1  has_run = current_bracket.has_run
2  has_else = current_bracket.has_else
3  if(has_run)
4    selected_option = current_bracket.runs[0].selected_option
5    switch(selected_option)
6      case 0: // Code für die erste Option erzeugen
7      case 1: // Code für die zweite Option erzeugen
8      ...
9    end
10 else if(has_else)
11   // Code für den Else-Teil erzeugen
12 end
```

Im Gegensatz zu Alternativen muss bei der optionalen Klammern vor der Auswahl der gewählten Option abgefragt werden, ob überhaupt eine Option gewählt wurde. Ist dies der Fall, kann dasselbe Verfahren wie in dem vorherigen Listing verwendet werden. Andernfalls muss überprüft

werden, ob ein Else-Teil in der Implementierungs-Klammer angegeben wurde. Dieser wird, wie alle anderen Teile auch, durch das Zeichen „|“ abgetrennt und wird ausschließlich ausgeführt, wenn keine Option gewählt wurde.

Listing 27: Pseudocode einer geschweiften Implementierungs-Klammer

```
1 number_of_runs = current_bracket.number_of_runs
2 for(i = 0; i < number_of_runs; ++i)
3   selected_option = current_bracket.runs[i].selected_option
4   switch(selected_option)
5     case 0: // Code für die erste Option erzeugen
6     case 1: // Code für die zweite Option erzeugen
7     ...
8   end
9 end
```

Eine Wiederholung ähnelt einer Alternative. Nur wird hier nicht nur ein einzelner Durchlauf betrachtet, sondern durch die `for`-Schleife aus Zeile zwei, mehrere. Dabei muss die ausgewählte Option natürlich auch immer aus dem aktuellen Durchlauf abgefragt werden.

Um eine grobe Vorstellung von der Umsetzung der Struktur des Pseudocodes zu LLVM-IR zu bekommen, wird im Folgenden die Implementierung einer Alternative betrachtet.

Zu Beginn muss während der Laufzeit abgefragt werden, welche Alternative in der Klammer durchlaufen wurde. Dies wurde dank `appl_gen` in einem `bracket`-Objekt gespeichert und kann somit durch die folgenden beiden Zeilen abgefragt werden.

```
1 Value* chosen_alternatives = utils::extract_from(ptr_to_current_bracket, 0, 0);
2 Value* chosen_alternative = utils::extract_from(chosen_alternative_per_run, 0);
```

Dabei wird zuerst ein Zeiger auf die Liste an Indizes für alle Durchläufe beschafft und anschließend das 0te Element aus dieser Liste ausgewählt.
Zur Umsetzung des `switch`-Statements muss in LLVM keine Sprungtabelle per Hand erstellt werden, da die Bibliothek selbst auch eine `switch`-Instruktion zur Verfügung stellt. Diese wird mit ...

```
1 ir_builder->CreateBr(select_alternative_bb);
2 utils::generate_on_basic_block(select_alternative_bb, [&]() {
3   SwitchInst* switch_inst = ir_builder->CreateSwitch(chosen_alternative_v,
      ↪ merge_alternatives_bb);
4   for(int i = 0; i < options.size(); ++i)
5     switch_inst->addCase(utils::get_integer_constant(i), options.at(i).block);
6 });
```

... in ihren eigenen basic Block geschrieben. Dabei wird neben dem Integer-Wert, in dessen Abhängigkeit der Ziel-Block ausgewählt wird, auch ein default-Block angegeben. Zu diesem sollte hier allerdings nie durch die `switch`-Instuktion gesprungen werden, da für jeden möglichen Wert von `chosen_alternative_v` ein `case`-Eintrag existiert! Im Anschluss an das Emittieren des Befehls werden ihm alle Möglichkeiten hinzugefügt. Dabei wird die Liste `blocks` durchlaufen, in der vorher für alle Optionen der Klammer der zugehörige Ausdruck und ein basic Block abgespeichert wurden.
Im nächsten Schritt kann der IR-Code für alle Optionen generiert werden. Dafür wird zuerst der aktuelle Kontext auf den voar-Kontext der aktuellen Klammer gesetzt und anschließend über die gerade genannte Liste an Optionen iteriert und innerhalb von `generate_on_basic_block` für jeden Ausdruck `gen_expr_code_internal` aufgerufen. Wichtig ist am Ende jeder Iteration den gespeicherten basic Block mit ...

```
1 alternative.block = ir_builder->GetInsertBlock();
```

... zu aktualisieren, da die Codegenerierung der Alternative weitere Blöcke eingefügt haben könnte und der gespeicherte immer auf den letzten Block der Alternative verweisen muss.
Die letzte Instruktion des Blocks für die jeweilige Alternative ist ein unbedingter Spring zu `merge_alternatives_bb`.
In diesem wird die ausgewählte Alternative um eins erhöht und in der C++-Variablen `return_value` gespeichert, um sie später als Ergebnis der Implementierungs-Klammer zurückzuliefern. Damit ist die Generierung für die Alternative abgeschlossen.

Auf den ersten Blick wirkt die Umsetzung dieses Pseudocodes zu LLVM-IR recht simpel. Allerdings wurden zwei größere Herausforderungen hier ausgelassen:

- Wie wird die Adresse geschachtelter Klammern, vor allem bei Vorkommen von Wiederholungen korrekt berechnet?
- Wie können die korrekten Werte von Parametern in einer Wiederholung gefunden werden, wenn die aufgebauten Kontexte nur zur Übersetzungszeit existieren?

Berechnung der Adresse der aktuellen Klammer

Um die erste Frage zu beantworten, wird in der C++-Variablen `Value* ptr_to_current_bracket` zu Beginn jedes Aufrufs der Generierungsfunktion die C++-Repräsentation des zur Laufzeit existierenden Zeigers auf die aktuelle Klammer gespeichert. Um deren Adresse zu berechnen, folgt die Hilfsfunktion `get_ptr_to_current_bracket` folgendem Algorithmus:

	Die, die aktuelle Klammer (direkt oder indirekt) umschließende Top-Level-Klammer aus der Parameter-Liste der Funktion zwischenspeichern
	Entspricht die zwischengespeicherte Klammer der aktuelle Klammer?
	Den korrekten Durchlauf durch die zwischengespeicherte Klammer auswählen
	Die korrekte Klammer des Durchlaufs auswählen und zwischenspeichern
	Den zwischengespeicherten Wert als Ergebnis zurückgeben

Abbildung 8: Algorithmus zur Berechnung der Adresse der des aktuellen `bracket`-Objekts

Ist die gesuchte Klammer eine Top-Level-Klammer, wird die Schleife nicht ausgeführt und es wird der im ersten Schritt gefundene Parameter zurückgeliefert.
Um die Schleife zu realisieren, benötigt `get_ptr_to_current_bracket` drei Parameter:

`iterations_through_repetitions`:
: In dieser Liste werden Paare an IDs und Indizes der Durchläufe aller umschließenden Wiederholungs-Klammern gespeichert, mit denen die aktuelle Klammer erreichbar ist. Mit diesen Indizes wird also durch die erste Dimension der Liste `bracket***` indiziert.

`indices_through_brackets`:
: Diese Liste enthält für jede Klammer einen Index, der ihre Position in ihrer umschließenden Klammer anzeigt. Ihre Elemente indizieren also durch die zweite Dimension der Liste `bracket***`

`bracket_stack`:
: Auch dieser Parameter ist eine Liste. Hier werden zu jeder umschließenden Klammer deren Typ und deren ID abgespeichert. Die Schleife kann nun also genau `bracket_stack.size() - 1` mal durchlaufen werden. Die Länge muss dabei um `1` verringert werden, da die äußerste Klammer bereits vor der Schleife gefunden wurde und daher jetzt nicht mehr beachtet werden muss.

Da die Schleife des Struktogramms und somit auch die Indizierung durch `bracket`-Objekte zur Laufzeit geschieht, müssen `iterations_through_repetitions` und `indices_through_brackets` Objekte des LLVM-Typs `Value*` enthalten. Die Elemente des Parameters `bracket_stack` werden hingegen nur zur Übersetzungszeit verwendet.
Mithilfe dieser Parameter wird `get_ptr_to_current_bracket` nun eine Reihe an `getelementptr`- und `load`-Instruktionen an den Anfang jeder Klammer emittieren.

Bevor das geschehen kann, müssen die Argumente für die beschriebenen Parameter allerdings erst erstellt werden. `iterations_through_repetitions` und `bracket_stack` sind beides statische Variablen der Generierungsfunktion `bracket_gen`. In letztere wird direkt am Anfang der Funktion die benötigte ID und Art der Klammer gespeichert und am Ende der Funktion wieder entfernt. `iterations_through_repetitions` werden nur Elemente hinzugefügt, falls eine Wiederholungs-Klammer vorhanden ist. So wird der Liste am Anfang der Code-Generierung für eine solche Klammer ein neuer Eintrag hinzugefügt und am Ende ebenfalls wieder entfernt.
`indices_through_brackets` kann nicht durch eine einfache statische Variable aufgebaut werden. Hierfür wird die komplette `cbrackets`-Struktur rekursiv durchlaufen, bis die ID der gefundenen Klammer mit der ID der aktuellen Klammer übereinstimmt. Ist dies der Fall, wird die Rekursion beendet und es kann aus jeder Rekursionsebene die Position der jeweiligen Klammer in ihrer umschließenden Klammer gelesen werden.

Setzen der korrekten Parameter-Werte für Parameter

Kommen Parameter direkt oder indirekt in einer Wiederholung vor, müssen sie in jeder Iteration der Wiederholung mit dem korrekten Wert belegt werden. Dies kann insbesondere bei mehreren geschachtelten Wiederholungs-Klammern komplex werden.

Für die folgende Implementierung ist der Unterschied zwischen den Durchläufen der Klammer und dem Vorkommen der Klammer in Durchläufen wichtig. Im Beispiel aus Listing 15 - calc.flx kommt die Alternative in zwei Durchläufen vor, hat selbst aber beide Male nur einen Durchlauf. Zum korrekten Setzen von Parameter-Werten spielt vor allem das Vorkommen der Klammer in Durchläufen eine Rolle.
So wird in `bracket_gen` zu Beginn die Liste `bracket_from_all_runs` aufgebaut. Diese hat den recht komplexen Typ `std::vector<std::pair<ContextBracket*, std::vector<std::pair<Value*, int>>>>`. Dabei wird neben der Klammer selbst eine weitere zu dieser Klammer gehörende Liste gesichert. Diese enthält die Indizes der Durchläufe aller umschließenden Wiederholungen, in denen die aktuelle Klammer vorkam. Dabei steht an zweiter Stelle des Paars der tatsächliche Index. Die erste Stelle repräsentiert hingegen den Index, der zur Laufzeit für jeden Durchlauf hochgezählt wird.

Abbildung 9 zeigt diese Struktur für Listing 15s Alternative.

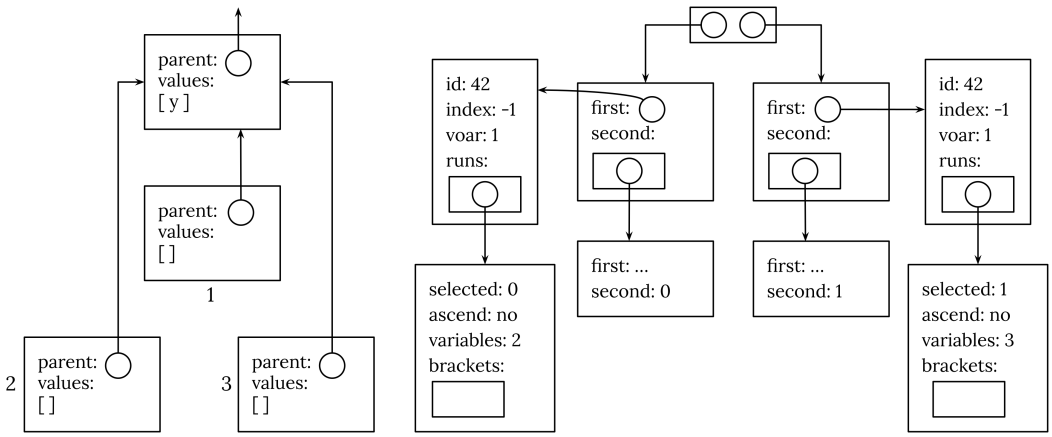


Abbildung 9: Klammer aus allen umschließenden Durchläufen

Auf der rechten Seite ist hier wieder die Struktur selbst abgebildet, während auch der linken Seite die dazugehörenden Kontexte dargestellt sind.
Bei genauerem Betrachten fällt die deutliche Ähnlichkeit zu Abbildung 7 - Von `instantiate_function` erzeugte Klammer-Struktur auf. So entsprechen die Kontexte 1, 2 und 3 den Kontexten 7, 8 und 9 aus Listing 7 und auch die beiden Klammer-Objekte mit der ID 42 sind dieselben wie in Listing 7.

Diese Struktur kann nun dazu genutzt werden, LLVM-IR-Code zu erzeugen, der zur Laufzeit alle Parameter mit den korrekten Werten belegt. Dieser Code wird zu Beginn jeder Implementierungs-Klammer stehen, wenn folgende Bedingungen zutreffen:

- Die aktuelle Klammer enthält Parameter
- Die aktuelle Klammer ist in mindestens einer Wiederholung geschachtelt, oder ist selbst eine Wiederholungs-Klammer

Ist mindestens eine Bedingung nicht erfüllt, muss kein extra Code zum Auswählen der korrekten Werte erzeugt werden! Während das für erstere bedingung wohl einfach nachvollziehbar ist, ist es für letztere nicht direkt offensichtlich. Da in Unterunterabschnitt 4.6.3 - Erzeugung der Funktion die Parameter des voar-Kontextes stets den Wert ihres erstes Vorkommnis erhalten und es ausschließlich unter der Verwendung von Wiederholungen mehr als ein Vorkommnis geben kann, enthält der Kontext bei der Verwendung von Optionen oder Alternativen bereits die korrekten Werte für den Durchlauf der Klammer.
Der generierte IR-Code wird für den Aufruf des calc-Operators dann folgenden Pseudocode ähneln:

Listing 28: Pseudocode für das Auswählen korrekter Parameter-Werte in Wiederholungen

```
1 acc = iteration_index == 0
2 y = acc ? 2 : old_y
3 acc = iteration_index == 1
4 y = acc ? 3 : y
```

Auf den ersten Blick fällt dabei auf, dass keine if-elseif-else-Struktur erzeugt wurde, sondern sequenziell der ternäre Operator verwendet wurde. Dies hat den einfachen Grund, dass es leichter zu implementieren war und der daraus entstehende IR-Code etwas kompakter ist. Sind mehrere geschachtelte Wiederholungen vorhanden, werden deren Indizes mit dem gewünschten Wert wie in Zeile eins verglichen und das Ergebnis dieses Vergleichs mit `acc` verundet.
Da eben keine Verzweigungen verwendet wurden, die Tests der Indizes also bei einer Übereinstimmung nicht vorzeitig abgebrochen werden können, dadurch also immer alle Tests und alle ternären Operatoren ausgeführt werden, ist Folgendes zu beachten: stimmt der erwartete Index nicht mit dem tatsächlichen überein, muss der ternären Operator den alten Wert des Parameters zurückliefern! Angenommen in Listing 28 liefert die Überprüfung `iteration_index == 0` `false` zurück, darf in Zeile vier der bereits korrekte Wert von `y` nicht überschrieben werden.

Zur Erstellung des LLVM-IR-Codes wird die zu `bracket_gen` lokale Funktion `setup_selects` genutzt. Diese bekommt als einzigen Parameter den Index (Typ `Value*`) der aktuellen Klammer, falls diese eine Wiederholung ist. Dies ist nötig, da dieser Index nicht in der Struktur aus Abbildung 9 vorhanden sein kann, da diese vor der Erstellung des Indexes aufgebaut wird. `setup_selects` muss dann vor jedem Durchlauf durch eine Klammer aufgerufen werden. Das heißt, der generierte Code wird zur Laufzeit einmalig für Optionen und Alternativen und mehrmals für Wiederholungen ausgeführt. Durch diese Funktion werden dann eine Reihe an `icmp eq`-, `and`- und `select`-Instruktionen emittiert. Der `select`-Befehl ist dabei das LLVM-IR-Äquivalent zu dem oben verwendeten ternären Operator.
An dieser Stelle noch ein kurzer Rückblick zu Unterunterabschnitt 4.5.3 - Variablen-Abfrage: in diesem Kapitel wurde geschrieben, dass der MOSTflexiPL -Wert Nil zurückgegeben wird, falls eine Variable im aktuellen Kontext nicht sichtbar ist. Dieses Verhalten ist für dieses Kapitel relevant! Folgendes Beispiel erklärt diese Relevanz:

```
1 oper [(f: int)] -> (int =
2   print f
3 );
4 oper
```

Da beim Aufruf des Operators die Klammer nicht durchlaufen wurde, wird der Parameter `f` nie mit einem Wert belegt und dadurch wird er auch nie in einen Kontext eingetragen. Wird nun der Körper des Operators erzeugt, wird früher oder später für Zeile zwei `query_gen` aufgerufen. Nun kann der Parameter allerdings nicht gefunden werden! Um auch in diesem Fall korrekten Code zu erzeugen, wird dann also der wohldefinierte Wert nil zurückgegeben. Dies ist der einzige Fall, dass ein Parameter nicht in einem Kontext gefunden werden kann!

4.6.5 Aufrufen der erzeugten Funktion

Wurde nun die komplette Funktion samt Inhalt und möglichen Implementierungs-Klammern erzeugt, kann sie zum Schluss auch aufgerufen werden. Dieses Kapitel kehrt damit zu der bereits aus Unterunterabschnitt 4.6.2 - Aufsetzten der Funktion bekannten Generierungsfunktion `appl_gen` zurück. Der letzte besprochene Schritt dieser Funktion war das Aufrufen von `instantiate_function`. Nun muss `app_gen` eine Liste an Argumenten für den Aufruf der generierten Funktion aufbauen.

Zur Erinnerung: die Argumente wurden von `appl_gen` bereits erstellt. Alle Top-Level-Parameter wurden dabei in `oper_params` gespeichert. Die Argumente für Klammern und geschachtelte Parameter wurden von `instantiate_function` aus `abrackets` nach `cbrackets` kopiert.
Zur Erstellung der Argumenten-Liste `std::vector<Value*> oper_args` werden zwei Schritte benötigt: zuerst werden alle Top-Level-Klammern und alle geschachtelten Parameter in die Liste eingetragen. Im Anschluss daran folgen alle Top-Level-Parameter. Die folgende lokale Hilfsfunktion wird über die komplette Struktur `cbrackets` laufen und für jedes Element entsprechend verfahren:

Listing 29: Sammeln der Funktions-Argumente #1

```
1  auto insert_params_and_brackets = [&](const std::vector<ContextBracket>&
       ↪ cbrackets) {
2    for(const ContextBracket& cbracket : cbrackets) {
3      if(cbracket.index != -1)
4        oper_args.at(cbracket.index) = cbracket.bracket_ptr;
5      for(const ContextRun& run : cbracket.runs) {
6        for(const auto& [name, value] : run.variables->named_values)
7          oper_args.at(value.index) = value.outside_alloca;
8        insert_params_and_brackets(run.brackets);
9      }
10   }
11 };
```

Ist der Index der Klammer in Zeile drei ungleich `-1` muss die Klammer eine Top-Level-Klammer sein und der in ihr gespeicherte Zeiger `bracket_ptr` in die Liste an Argumenten eingefügt werden muss. In `bracket_ptr` wurde dabei die am Anfang von `appl_gen` erstellte `alloca`-Instruktion des `bracket`-Objekts gespeichert. Im Anschluss werden alle Parameter aus allen Durchläufen der Klammer an ihrer jeweiligen Stelle in die Argumenten-Liste eingetragen. `outside_alloca` speichert auch hier wieder die von `appl_gen` erstelle Instruktion zur Erzeugung des Arguments. Zum Schluss wird die Hilfsfunktion für alle Klammern aus allen Durchläufen rekursiv aufgerufen.
Terminiert diese Funktion endgültig, steht nun an allen Stellen der Liste `oper_args` ein Element, außer an denen, die für Top-Level-Parameter reserviert sind. Diesen Fakt nutzt die nächste und letzte Schleife aus und wird genau an diesen Stellen die Elemente aus `oper_params` speichern.

Listing 30: Sammeln der Funktions-Argumente #2

```
1 auto insert_top_level_param_allocas = [&](const std::vector<Value*>& oper_params)
    ↪ {
2   int param_index = 0;
3   for(int i = 0; i < oper_args.size(); ++i) {
4     if(oper_args.at(i) == nullptr)
5       oper_args.at(i) = oper_params.at(param_index++);
6   }
7 };
```

Damit sind nach einem Aufruf beider Hilfsfunktionen alle Argumente korrekt in die Liste einsortiert und die Funktion kann mit ...

```
1 ir_builder->CreateCall(instantiated_operator, oper_args);
```

... als letzte Aktion der Generierungsfunktion `appl_gen` aufgerufen werden.

4.6.6 Zusammenfassung

Nachdem auf den letzten Seiten sehr viel über Datenstrukturen, abstrakte Ideen und den groben Ablauf der Implementierung geschrieben wurde, allerdings nie konkreter LLVM-IR-Code gezeigt wurde, soll dies hier nun nachgeholt werden. Somit ist dieses Kapitel eine knappe Zusammenfassung der Generierung von benutzerdefinierten Operatoren und soll helfen, eine Brücke zwischen der Struktur des LLVM-IR-Codes und den beschriebenen Konzepten zu schaffen.

Da das Beispiel aus Listing 15 - calc.flx deutlich zu viel LLVM-IR-Code generiert, wird für dieses Kapitel ein etwas simplerer Operator verwendet:

```
1 oper { a (a: int) | b (b: int) } -> (int=
2   {print a | print b}
3 );
4 oper a 1
```

In den folgenden Abschnitten wird nun der aus diesem Quell-Programm resultierende LLVM-IR-Code in genau der Reihenfolge gezeigt, in der er tatsächlich in der erstellten Datei steht.

Erstellen der Argumente

Listing 31: LLVM-IR zur Erstellung der bracket-Strukturen (`appl_gen`)

```
1  %mult.ptr.0 = alloca %bracket, align 8
2  %mult.num.passes.0 = getelementptr %bracket, %bracket* %mult.ptr.0, i32 0, i32 1
3  store i32 1, i32* %mult.num.passes.0, align 4
4  %a.ptr = alloca i35, align 8
5  store i35 4294967297, i35* %a.ptr, align 8
6  %mult.brackets.sel.0 = alloca i32, align 4
7  %mult.selopt.0 = getelementptr i32, i32* %mult.brackets.sel.0, i32 0
8  store i32 0, i32* %mult.selopt.0, align 4
9  %mult.selopt.ptr.0 = getelementptr %bracket, %bracket* %mult.ptr.0, i32 0, i32 0
10 store i32* %mult.brackets.sel.0, i32** %mult.selopt.ptr.0, align 8
11 %mult.brackets.ptr.0 = alloca %bracket**, align 8
12 %mult.brackets.ptr.1= getelementptr %bracket, %bracket* %mult.ptr.0, i32 0, i32 3
13 store %bracket*** %mult.brackets.ptr.0, %bracket**** %mult.brackets.ptr.1,align 8
14 %run.brackets.0 = alloca %bracket*, align 8
15 %1 = getelementptr %bracket*, %bracket** %run.brackets.0, i32 0
16 store %bracket* %mult.ptr.0, %bracket** %1, align 8
```

Hier wird direkt in Zeile eins das Klammer-Objekt erstellt und anschließend in Zeile drei die Anzahl an Durchläufen der Klammer an die berechnete Adresse eingetragen. Zeile vier reserviert Speicher für den Parameter `a`. Für `b` wird hier nie Speicher beschafft, da dieser dynamisch nie vorkommt. `a` wird mit dem auf den ersten Blick komisch wirkenden dezimalen Wert `4294967297` initialisiert. Dies ist allerdings nur die dezimale Darstellung des natürlichen MOSTflexiPL-Werts 1. Die nächsten fünf Zeilen (sechs bis zehn) speichern ab, dass im 0ten Durchlauf die 0te Option der Klammer gewählt wurde. Alle weiteren Zeilen initialisieren die in allen Durchläufen geschachtelten Klammern. Da diese hier nicht vorhanden sind, wird ausschließlich Speicher reserviert, ohne diesem auch Inhalt zuzuweisen.
All diese Instruktionen wurden von der Hilfsfunktion `generate_call_args` aus `appl_gen` erstellt.

Aufruf der Funktion

Während der Compiler als Nächstes die Funktion selbst erstellen wird, folgt in der erstellten Datei allerdings schon der Aufruf.

Listing 32: LLVM-IR zum Aufrufen der generierten Funktion (`appl_gen`)

```
1 %2 = call i35 @"0x22bc540r10p"(%bracket* %mult.ptr.0, i35* %a.ptr)
```

Hier werden nur, wie im letzten Kapitel gesehen, die gerade erstellten und initialisierten Zeiger übergeben.

Inhalt der Funktion

Die generierte Funktion beginnt mit folgenden Zeilen:

Listing 33: Generierte Funktion #1 (`instantiate_function` + `bracket_gen`)

```
1 %a.runs.0 = alloca i35, align 8
2 %2 = load i35, i35* %a.ptr.0, align 8
3 store i35 %2, i35* %a.runs.0, align 8
4 %mult.runs.0.ptr = getelementptr %bracket, %bracket* %0, i32 0, i32 1
5 %mult.runs.0 = load i32, i32* %mult.runs.0.ptr, align 4
6 br label %mult.loop.begin.0
```

In Zeile eins wird der Parameter `a` des voar-Kontextes erstellt und in Zeile drei mit dem Wert seines ersten Vorkommnisses initialisiert. Bevor in die Schleife für die Durchläufe der Wiederholungs-Klammer gesprungen werden kann, muss noch die Anzahl der Durchläufe aus der übergebenen Klammer ausgelesen werden. Dabei steckt in Zeile vier auch die, hier triviale, Berechnung der Adresse zur aktuellen Klammer. Da diese eine Top-Level-Klammer ist, kann direkt der Parameter `%0` verwendet werden. Die in Abbildung 8 - Algorithmus zur Berechnung der Adresse der des aktuellen `bracket`-Objekts gezeigte Schleife wird in diesem Beispiel also nicht durchlaufen.

Listing 34: Generierte Funktion #2 (`bracket_gen`)

```
1 mult.loop.begin.0:
2 %run.index.0 = phi i32 [ 0, %1 ], [ %run.index.inc.0, %mult.merge.opt.0 ]
3 %3 = icmp slt i32 %run.index.0, %mult.runs.0
4 br i1 %3, label %mult.loop.body.0, label %mult.loop.end.0
```

`run.index.0` enthält immer den Index der aktuellen Iteration. Dieser ist `0`, falls vom Anfang der Funktion (basic Block `%1`) an diese Stelle gesprungen wurde, beziehungsweise die inkrementierte

Version, falls vom Ende der Schleife (basic Block `%mult.merge.opt.0`) an diese Stelle gesprungen wurde. Danach folgt die Überprüfung der Abbruchbedingung. Ist der Index gleich der Anzahl an Durchläufen, wird in Zeile vier nach `%mult.loop.end.0` gesprungen. Andernfalls wird der Körper der Schleife ausgeführt.
In diesem Körper wird nun zuerst der korrekte Wert von `a` für die aktuelle Iteration bestimmt und anschließend zum Block der ausgewählten Option gesprungen.

Listing 35: Generierte Funktion #3 (`bracket_gen`)

```
1  mult.loop.body.0:
2  %4 = icmp eq i32 %run.index.0, 0
3  %5 = and i1 true, %4
4  %6 = select i1 %5, i35* %a.ptr.0, i35* %a.runs.0
5  %7 = load i35, i35* %6, align 8
6  store i35 %7, i35* %a.runs.0, align 8
7  %8 = getelementptr %bracket, %bracket* %0, i32 0, i32 0
8  %9 = load i32*, i32** %8, align 8
9  %10 = getelementptr i32, i32* %9, i32 %run.index.0
10 %11 = load i32, i32* %10, align 4
11 switch i32 %11, label %mult.loop.end.0 [
12   i32 0, label %mult.option.0
13   i32 1, label %mult.option.1
14  ]
```

Zeile zwei bis sechs bilden dabei genau den in Listinmg 28 gezeigten Code ab. Als Nächstes wird in die Liste an gewählten Optionen pro Durchlauf indiziert, um in Zeile zehn den Index des 0ten Durchlaufs zu laden. Anhand dieses Indexes wird die `switch`-Instruktion bestimmen, zu welchem Block gesprungen werden soll.

Listing 36: Generierte Funktion #4 (`bracket_gen`)

```
1  mult.option.0:
2  %a.0 = load i35, i35* %a.runs.0, align 8
3  %12 = and i35 %a.0, 4294967296
4  %13 = icmp ne i35 %12, 0
5  %14 = select i1 %13, i8* getelementptr inbounds ([5 x i8], [5 x i8]*
       ↪ @prinf_format, i32 0, i32 0), i8* getelementptr inbounds ([2 x i8], [2 x
       ↪ i8]* @empty_printf, i32 0, i32 0)
6  %15 = trunc i35 %a.0 to i32
7  %16 = call i32 (i8*, ...) @printf(i8* %14, i32 0, i32 %15)
8  br label %mult.merge.opt.0
9
10 mult.option.1:
11 %17 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([2 x i8], [2 x i8]*
       ↪  @empty_printf, i32 0, i32 0), i32 0, i32 0)
12 br label %mult.merge.opt.0
```

Beide hier gezeigte Blöcke enthalten Code für den `print`-Operator, auch wenn sie auf den ersten Blick recht unterschiedlich wirken.
In `mult.option.0` wird der Parameter `a` des voar-Kontextes geladen, um den Wert aus der aktuellen Iteration zu bekommen. Nun muss getestet werden, ob der geladene Wert MOSTflexiPLs Nil-Wert (`4294967296`) entspricht. In Abhängigkeit dieses Ergebnisses wird C's `printf` mit einem jeweils anderen Format-String aufgerufen.
Auch `mult.option.1` enthält prinzipiell denselben Code, nur konnte hier schon zur Übersetzungszeit festgestellt werden, dass der Parameter `b` nil ist, da er im Aufruf nicht vorkommt. Ergo konnte die Überprüfung komplett weggelassen werden und direkt der korrekte Format-String eingefügt werden.

Beide Blöcke springen zum Schluss zu `%mult.merge.opt.0`.

Listing 37: Generierte Funktion #5 (`bracket_gen`)

```
1 mult.merge.opt.0:
2 %run.index.inc.0 = add i32 %run.index.0, 1
3 br label %mult.loop.begin.0
```

Hier wird der aktuelle Index inkrementiert und wieder an den Beginn der Schleife gesprungen, um erneut auf die Abbruchbedingung zu testen und gegebenenfalls den Körper erneut auszuführen, oder ganz zum Ende der Funktion zu springen.

Listing 38: Generierte Funktion #6 (`bracket_gen`)

```
1 mult.loop.end.0:
2 %18 = zext i32 %mult.runs.0 to i35
3 %19 = add i35 %18, 4294967296
4 ret i35 %19
```

Hiermit ist das Ende der Funktion erreicht. So wird zum Schluss die Anzahl der Durchläufe zu einem natürlichen MOSTflexiPL-Wert umgewandelt und als Ergebnis der Funktion zurückgegeben.

4.7 Codegenerierungs-Epilog

Nachdem nun also der komplette AST traversiert wurde und jeder Knoten mithilfe der entsprechenden Generierungsfunktion zu LLVM-IR-Code in einem Modul umgewandelt wurde, kann im letzten Schritt dieses Modul zu einer ausführbaren Datei umgewandelt werden. Während es bisher um das Übersetzen von MOSTflexiPL-Code zu LLVM-IR-Code ging, behandelt dieses Kapitel nun die Übersetzung von IR zu Maschinencode. Wie bereits in Unterabschnitt 4.4 steht

der Code für diesen Schritt nicht in der Datei `code_gen.cxx`, sondern wird über einen Callback an die zentrale Funktion `gen_expr_code` übergeben. Diese Funktion wird den Callback nach der Erzeugung des Moduls mit eben diesem als Parameter aufrufen und ab dann selbst nichts mehr an dem generierten Modul ändern. Da der erzeugte Code noch beliebig stark optimiert werden kann oder auch eine komplett andere Art des Kompilierens gewählt werden kann (siehe Unterabschnitt 5.2 - Just-in-time-Kompilierung), wird dieser Schritt durch den verwendeten Callback wieder so flexibel wie möglich gestaltet.

Das hier vorgestellte Backend unterstützt verschiedene Ziel-Dateien als finale Ausgabe:

- LLVM-IR (.ll)
- Maschinencode (.s)
- Objekt- und ausführbare Datei (.o und .out)

Erzeugung einer .ll Datei:
: Dies ist wohl die einfachste Option. Ein LLVM `Modul` bietet über die öffentliche Methode `print` die Möglichkeit, die in-Memory Repräsentation zu LLVM-Bitcode umzuwandeln. Dabei muss an erster Stelle der Filedescriptor in Form eines Objekts des Typs `raw_fd_ostream` übergebenen werden.

Erzeugung einer .s Datei / Erzeugung der ausführbaren Datei:
: Assembly und ausführbare Dateien werden nicht über eine einfache print-Funktion erstellt, sondern benötigen einen Pass-Manager. Dieser wird von der in Unterabschnitt 4.4 - Codegenerierungs-Prolog erstellten Target-Machine mit einem geeigneten Pass gefüllt. So wird mit der folgenden Zeile je nach übergebenem Flag der korrekte Lauf ausgewählt.

```
1 target_machine->addPassesToEmitFile(pass_manager, file, nullptr, resultTypes
    ↪ & EXE ? CGFT_ObjectFile : CGFT_AssemblyFile);
```

  Um die Objekt-Datei im letzten Schritt ausführbar zu machen, wird ein Linker benötigt. Dabei muss zwingend mit libC gelinked werden, damit `printf` und `scanf` verfügbar sind. Anstatt selbst den Linker aufzurufen, ruft dieses Backend den Compiler auf, mit dem es übersetzt wurde. Dieser sollte in aller Regel wohl auch Objekt-Dateien linken können, oder diese Aufgabe zumindest an einen Linker weiter geben können.

Mit dem Terminieren dieses Lambdas wird dann auch der initiale Aufruf zu `gen_expr_code` und letzten Endes auch der komplette Compiler terminieren.

4.8 Optimierung des Zwischencodes

Optimierung ist ein extrem großer und wichtiger Teilbereich der Entwicklung eines Compilers. Während LLVM einige kleinere Optimierungen, wie beispielsweise triviales Constant-Folding standardmäßig auf der IR ausführt, muss der Entwickler Optimierungen für größeren Leistungsgewinn natürlich selbst programmieren. Unterabschnitt 5.1 - Weitere Optimierung des Zwischencodes wird genauer darauf eingehen, welche Möglichkeiten LLVM theoretisch bietet und welche Arten von Optimierungen für dieses Projekt wohl noch sinnvoll gewesen wären. Das Ziel dieses Kapitel ist es hingegen eine tatsächlich entwickelte Optimierung vorzustellen.

4.8.1 Optimierung benutzerdefinierter Operatoren

Die größte Quelle für unleserlichen und unperformanten IR Code, ist in diesem Backend klar das Vorbereiten eines Aufrufs eines benutzerdefinierten Operators. So erzeugt das Beispiel aus Listing 15 - calc.flx insgesamt 54 Zeilen IR-Code, der nur aus `getelementptr`- und `store`-Instruktionen besteht, um die für den Aufruf nötigen Klammerstrukturen zu erzeugen. Für komplexere Operatoren und mehrere verschiedene Aufrufe steigt diese Zahl extrem! Ergo wäre es also sowohl eine statische als auch dynamische Optimierung und dadurch sehr erstrebenswert diesen Code zu verkürzen.
Die Optimierung hierfür ist recht offensichtlich und bietet einen nur sehr geringen Nachteil. So kann abermals der Fakt ausgenutzt werden, dass der Compiler beim Erzeugen der Implementierungs-Klammern genau weiß, welche wie oft und mit welcher Option durchlaufen wurden. Wenn diese Klammern nicht mehr maximal allgemeinen Code erzeugen, sondern auf den Aufruf des Operators zugeschnitten werden, könnte das Erzeugen der Klammerstrukturen wegfallen, da diese Information nun fest in der Implementierung steht.
Anstatt also äquivalenten IR-Code zu dem in Unterabschnitt 4.6.4 vorgestellten Pseudocode zu generieren, werden alle Implementierungs-Klammer expandiert (vgl. Listing 16 - Expandierte Implementierungs-Klammern). Damit diese Optimierung die Korrektheit des generierten Codes nicht gefährdet, ist es nun umso wichtiger, dass der Name der erzeugten Funktion so exakt wie möglich abbildet, wie genau die einzelnen Klammern durchlaufen wurden. Wäre diese Abbildung zu ungenau, könnte bei einem Operator-Aufruf eine Funktion aufgerufen werden, deren Klammern nicht für diesen einen speziellen Aufruf expandiert wurden, sondern für einen möglicherweise minimal anderen, wodurch das erzeugte Programm natürlich nicht mehr das Quell-Programm korrekt repräsentieren würde.

Die Optimierung wurde in einer eigenen Funktion `opt::bracket_gen` entwickelt. Wird dem Compiler das Flag `-fexpand-brackets` aufgerufen, wird in der regulären Generierungsfunktion der Implementierungs-Klammern eben diese optimierte Variante ausgewählt und `appl_gen` an-

gewiesen keine Befehle zur Erstellung `bracket`-Objekte zu emittieren. Dabei muss der Funktion `opt::bracket_gen` neben dem Klammer-Ausdruck, der Liste der aktuellen Klammer in allen umschließenden Durchläufen (`bracket_from_all_runs`), auch der bereits bekannte `bracket_stack` übergeben werden, damit diese Funktion dieselben Datenstrukturen noch einmal aufbauen muss.
Die Implementierung ist im Gegensatz zur regulären Version von `bracket_gen` recht simpel. So wird eine Wiederholungs-Klammer wie folgt expandiert:

Listing 39: Expandieren einer Wiederholungs-Klammer

```
1 for(const ContextRun& run : bracket_from_all_runs[index].first->runs) {
2   current_context = run.variables;
3   gen_expr_code_internal(options.at(run.selected_option));
4 }
5 return flx_value::get(flv_value::nat, bracket_from_all_runs[index].first->runs.
      ↪ size());
```

Zu Beginn wird über alle Durchläufe der aktuellen Klammer iteriert. `index` ist dabei immer die Nummer der Iteration, in der die aktuelle Klammer gerade aufgerufen wird. Für jeden Durchlauf wird nun der aktuelle Kontext auf den Kontext des Durchlaufs gesetzt. Anschließend kann die Generierung für die ausgewählte Option angestoßen werden.
Während in der regulären Version von `bracket_gen` die korrekten Werte für den aktuellen Durchlauf mühsam in jeder Iteration durch Instruktionen ausgewählt werden mussten, passiert dies nun zur Übersetzungszeit.

4.8.2 LLVM Optimierungsläufe

LLVM bietet selbst auch eine Reihe an Optimierungen an. So können durch verschiedene sogenannte Passes ganz unterschiedliche Bereiche des erzeugten Codes betrachtet werden. Dabei sind für die Optimierung vor allem Transformations-Läufe interessant. Da diese aber nicht im Rahmen dieser Arbeit selbst entwickelt wurden, wird auf diese und deren Funktionsweise hier nur knapp eingegangen.

Insgesamt werden dem Entwickler 58 Transformations-Läufe angeboten. Allerdings ergibt es wohl wenig Sinn, alle zu verwenden. Stattdessen sollte mithilfe des Hintergrundwissens über die Natur des generierten IR-Codes entschieden werden, welche Optimierungen für eine bestimmte Sprache Sinn ergeben. Für den hier erzeugten Code verwendet dieses Backend bei gesetztem `-O2` Flag die folgenden Optimierungen der LLVM-Bibliothek:

mem2reg:

In diesem Durchlauf werden Speicherreferenzen zu Registerreferenzen umgewandelt. So können beispielsweise `alloca`-Instruktionen vermieden werden, wenn eine Speicherstelle nur einmalig beschrieben wird, also die Semantik einer Konstante aufweist. Diese Optimierung hat hier hohes Potenzial, da dieses Backend ausschließlich für Literale keine Allokationen vornimmt.

dce: Dead Code Elimination tut prinzipiell genau das, was der Name bereits vermuten lässt. So werden dabei aber nicht nur Instruktionen entfernt, die nicht ausgeführt werden, sondern auch Instruktionen, die keine Seiteneffekte haben, und deren Ergebnis nicht verwendet wird. Wenn beispielsweise eine Wiederholungs-Klammer durchlaufen wird, liefer diese als Resultat die Anzahl ihrer Durchläufe zurück. Dazu wird der intern verwendete `i32` Wert zum Zählen jener Durchläufe zu einem `i35` MOSTflexiPL-Wert umgewandelt. Die Instruktionen zur Umwandlung würden von diesem Pass entfernt werden, falls das Ergebnis nicht benötigt wird.

dse: Hier werden unnötige `store`-Instruktionen entfernt. Unter Umständen können in `appl_gen` bei der Vorbereitung der Laufzeit-Klammer-Strukturen unnötige Speicher-Zugriffe generiert werden. Diese können aber auch durch den Benutzer der Programmiersprache durch mehrfache Zuweisung an eine Variable, wobei nur der letzte zugewiesene Wert jemals abgefragt wird.

simplifycfg:

Dieser Pass eliminiert unnötige Kanten des Kontroll-Fluss-Graphen. Dieses Backend generiert häufig zusätzliche basic Blocks, die ausschließlich der besseren Lesbarkeit des erzeugten IR-Codes dienen. Diese, und alle weiteren Blöcke mit nur einem Vorgänger und Nachfolger, werden von diesem Durchlauf direkt in ihren Vorgänger geschrieben, um den unnötigen Sprung zu entfernen.

5 Ausblick

Während im letzten Kapitel ausschließlich Gedanken präsentiert wurden, die auch tatsächlich implementiert wurde, soll es hier nun um Weiterführendes gehen, das aber nicht mehr Teil der Implementierung ist.

5.1 Weitere Optimierung des Zwischencodes

Schon in Unterabschnitt 4.8 - Optimierung des Zwischencodes wurden Gedanken zur Optimierung des Zwischencodes genannt. Diese können allerdings noch erweitert werden.

Zur Erinnerung: der ursprüngliche Grund zur Generierung einer Funktion pro einzigartigem Operator-Aufruf war, dass es andernfalls nicht offensichtlich ist, wie die Adresse in Klammern geschachtelter Parameter berechnet werden kann. Sind allerdings keine geschachtelten Parameter in der Signatur des Operators vorhanden, ist es problemlos möglich eine einzige Funktion für diesen Operator zu emittieren und diese für alle Aufrufe zu verwenden.
Bisher wurden nur Optimierungen besprochen, die direkt während der Generierung ausgeführt werden. LLVM bietet allerdings auch die Möglichkeit, Optimierungsdurchläufe zu entwickeln. Diese können nach dem Generieren auf einem kompletten Modul oder auf jeder einzelnen Funktion ausgeführt werden. Nun könnte mithilfe mehrerer dieser Läufe das standardmäßige triviale Constant-Folding auf Operatoren erweitert werden. Hierbei wird wieder ausgenutzt, dass bei der Generierung einer Funktion alle Informationen über den Operator-Aufruf bekannt sind. So kann beim Aufruf erkannt werden, ob ausschließlich konstante Parameter übergeben wurden. Dies kann entweder eine echte MOSTflexiPL-Konstante sein, aber auch in Integer-Literal. Ist dies der Fall, ist es möglich, den kompletten Operator-Aufruf zur Übersetzungszeit auszuwerten, wodurch erst gar keine Funktion emittiert werden muss. Diese Optimierung mach das Generieren einer Funktion pro Aufruf extrem mächtig!

Dies sind nur zwei von vielen Möglichkeiten, wie der erzeugte Code noch weiter transformiert werden kann. Das meiste Potenzial bieten dabei aber tatsächlich benutzerdefinierte Operatoren und Implementierungs-Klammern, da der Rest der Programmiersprache MOSTflexiPL tatsächlich eher simpel gehalten ist.

5.2 Just-in-time-Kompilierung

Im Gegensatz zu einem traditionellen Compiler, übersetzt ein JIT nicht den kompletten Code auf einmal, sondern nur einzelne Teile, genau dann, wenn sie benötigt werden. Dies verbindet die überlegene Performance eines Compilers mit der Flexibilität eines Interpreters.
Gerade für sich schnell weiterentwickelnde Sprachen wie MOSTflexiPL kann es von großem Vorteil sein, wenn ein „Write-Compile-Run"-Zyklus so kurz wie möglich ist, oder sogar ein REPL (**R**ead-**E**val-**P**rint-**L**oop) existiert. So kann umso effizienter neuer Code getestet und die Sprache an ihre Grenzen geführt werden.

So könnte das Prinzip hinter einem MOSTflexiPL-JIT folgendermaßen aussehen[LLVc]:

1. Initial ein Modul erstellen
2. JIT initialisieren und mit `addModule` das Modul hinzufügen
3. IR-Code für den eingegebenen MOSTflexiPL-Code in eine beliebig benannte LLVM-IR-Funktion emittieren

Ab hier steht die übersetzte Funktion im Speicher zur Verfügung. Da LLVM automatisch das Application Binary Interface des generierten Codes dem des Host-Computers anpasst, kann die erzeugte Funktion aus dem Compiler-Prozess heraus aufgerufen werden.

4. Jene Funktion mit `findSymbol` und `getAddress` abfragen
5. Die hierdurch gewonnenen Adresse zu einem Funktionszeiger casten und diesen aufrufen
6. Mit Schritt eins von vorne beginnen

5.3 Erzeugen von Debugging-Symbolen

Produziert Programmcode nicht das erwartete Resultat, kann das Finden des Grunds vor allem bei größeren Programmen schnell zu einem langwierigen Prozess werden. Die Möglichkeit Programmcode nicht nur statisch betrachten zu können, sondern sein Verhalten Schritt für Schritt während der Laufzeit verfolgen zu können, lässt das Finden von Fehlern zu einem wesentlich angenehmeren Vorgang werden.
Um das Debugging von MOSTflexiPL-Code zu ermöglichen, können Debugging-Informationen über ein spezielles Format in Form von Metadaten emittiert werden.

Dabei ist das Ziel dieser Metadaten abzubilden, wie Teile des Quellprogramms in und an welcher Stelle der IR dargestellt sind. Dabei werden beispielsweise Typen, Funktion und natürlich einzelne Zeilen des ursprünglichen Programms auf die Zwischensprache abgebildet. Diese Metadaten wurden dabei so konzipiert, dass sie unabhängig des Ziel-Debuggers und dessen Repräsentation dieser Informationen verwendet werden können.

Zum Erzeugen der Debugging-Metadaten wird analog zur Klasse `IRBuilder` ein Objekt der Klasse `DIBuilder` verwendet. So könnte beispielsweise mit ...

```
1 bi_builder->createBasicType("flx-value", 35);
```

... der verwendete Typ für MOSTflexiPL-Werte erstellt werden.
Nach demselben Prinzip funktioniert auch die Erzeugung der meisten anderen nötigen Funktionen.

Eine weitere Interpretation dieses Kapitels könnte das Erzeugen von Debugging-Symbolen für den generierten IR-Code sein. Während diese Interpretation für Benutzer der Sprache MOSTflexiPL nicht sonderlich nützlich wäre, würde sie das (Weiter-)Entwickeln dieses Backends enorm vereinfachen. Häufig passiert es beim Testen des generierten Codes, dass dieser durch einen Segmentation-Fault abbricht und der Grund des Fehlverhaltens nicht offensichtlich ist. Nun muss durch mühsames abändern und anschließendes Ausprobieren des IR-Codes nach und nach der Grund gefunden werden. Glücklicherweise bietet das Tool `debugir` (`https://github.com/vaivaswatha/debugir`) genau diese Möglichkeit. Ähnlicher Code könnte auch für dieses Backend entwickelt werden und somit noch besser auf den erzeugten IR-Code abgestimmt sein, was nutzen eines Debuggers auf dem generierten LLVM-IR-Code noch effizienter gestalten würde.

6 Fazit

Zu Beginn dieser Arbeit wurden mehrere Möglichkeiten zur Generierung von Maschinen-Code vorgestellt. Dabei war neben LLVM auch unter anderem das Erzeugen von C-Code eine Option. Dieses Kapitel soll zum Schluss eine Brücke zum Anfang der Arbeit schlagen und abwiegen, ob die Implementierung eines Backends in LLVM tatsächlich die zu präferierende Option ist.

MOSTflexiPL ist sicher keine Sprache, die das Entwickeln eines solchen Backends besonders simpel gestaltet. So abstrahieren benutzerdefinierte Operatoren und vor allem ihre Klammern von so vielen Dingen, dass die Umsetzung dieser Konzepte in einer Sprache, die per Design von so wenig wie möglich abstrahiert, eine echte Herausforderung ist. Trotzdem ist die Frage, ob LLVM die geeignetste Bibliothek für dieses Vorhaben war, keine, die eindeutig beantwortet werden kann.

In einem Umfeld, in dem es ausschließlich auf das Verhältnis von Zeit zu Fortschritt ankommt, wäre LLVM für dieses Projekt sicher nicht die beste Variante. So ist es wohl an einigen Stellen deutlich einfacher Programmcode einer anderen Hochsprache zu generieren, da man so die Möglichkeit hätte wesentlich simpler Abstraktionen in den generierten Code einzufügen. So wäre es für dieses Projekt sehr hilfreich gewesen, die in Kapitel 4.6.3 für den Compiler erstellten Kontexte auch zur Laufzeit zur Verfügung zu haben.
Und auch wenn beim Schreiben einer Bachelorarbeit sicher das Verhältnis von Zeit zu Fortschritt ebenfalls eine entscheidende Rolle spielt, ist es hier aber noch wichtiger neue Dinge zu erlernen und durch Herausforderungen interessante Ideen zu produzieren. So wurden in dieser Arbeit Wege gefunden, um die Limitationen, die eine Zwischensprache wie die LLVM-IR von Haus aus hat, zu umgehen, ohne den einfacheren Weg über eine Hochsprache zu gehen. Und auch wenn das Pflegen eines solchen Backends vielleicht nicht der Weg ist, den MOSTflexiPL langfristig einschlagen wird, war das Erkunden dieses Weges sicher trotzdem ein Erfolg!

7 Appendix

7.1 MOSTflexiPL-Operatoren

Die folgende Tabelle enthält alle aktuell in MOSTflexiPL vorhandenen Operatoren, wobei die Operatoren selbst in blau hervorgehoben sind.

Operatoren		
Operator	Beispiel	Generierungs-funktion
Binäre Addition	10 + 5	bin_gen
Binäre Subtraktion	10 - 5	
Binäre Multiplikation	10 * 5	
Binäre Division	10 / 5	
Potenz	10 ^ 5	
Negation	~ true	neg_gen
Logisches Und	true & false	logic_gen
Logisches Oder	true | false	
Unäres Minus	-10	chs_gen
Fakultät	10!	fac_gen
Semikolon	10 + 5 ; 10 - 5	sequ_gen
Klammerung	(1+2)*3	paren_gen
Deklaration	a: int?	cdelc_gen
Zuweisung	a =! 10	assign_gen
Abfrage	?a	query_gen
Print	print 10 width 2	print_gen
Read	a =! read	read_gen
Vergleich	1 < 2 < 3 = 3	cmp_gen
Verzweigung	if true then print 1 elseif true then print 2 else print 3 end	branch_gen
Schleife	while true do print 1 until true do print 2 end	loop_gen

Tabelle 2: MOSTflexiPL-Operatoren #1

Operatoren		
Operator	Beispiel	Generierungs-funktion
Integer-Literal	1	intlit_gen
Boolesches-Literal	true	boollit_gen
Operator-Definition	(x:int) plus (y:int) -> (int =x + y)	odecl_gen
Operator-Anwendung	1 plus 2	appl_gen
Klammer	... {[plus|minus]} ... -> (...= {[print 1 | print 2 | print 3]}	bracket_gen
Konstanten-Auswertung	a: int; a	`const_gen`

Tabelle 3: MOSTflexiPL-Operatoren #2

7.2 Übersetzungszeit - C vs. LLVM-IR

Als simples Beispiel wurde hier dieses Projekt übersetzt. Dabei wurde sowohl mit dem GCC 11.2.0, als auch mit Clang 13.1.6 direkt zu Assemblercode, und nocheinmal mit Clang erst zu LLVM IR und ausgehend von der IR anschließend zu nativem Code übersetzt (für dieses Beispiel spielt nur der letzte Übersetzungsschritt von IR zu Assembly eine Rolle!). Alle Übersetzungsvorgänge wurden jeweils drei mal hintereinander und ohne jegliche Optimierungen durchgeführt. Die folgende Tabelle zeigt die Ergebnisse dieses Versuchs:

#	GCC (C zu Assembly)	Clang (C zu Assembly)	Clang (LLVM IR zu Assembly)
1	15.23s	14.01s	8.15s
2	15.11s	13.78s	7.88s
3	15.16s	13.81s	7.79s

Tabelle 4: Übersetzungszeiten - C vs. LLVM IR

Auch wenn dieses Beispiels sicher einige Details außen vor lässt, zeigt es wohl trotzdem eine recht deutliche Tendenz.

Literatur

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. 1997.

[LA04] C. Lattner und V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004. DOI: 10.1109/CGO.2004.1281665. URL: https://www.llvm.org/pubs/2004-01-30-CGO-LLVM.pdf.

[TC10] David A. Terei und Manuel M. T. Chakravarty. "An llVM backend for GHC". In: *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*. Hrsg. von Jeremy Gibbons. ACM, 2010, S. 109–120. DOI: 10.1145/1863523.1863538. URL: https://doi.org/10.1145/1863523.1863538.

[SP15] Suyog Sarda und Mayur Pandey. *LLVM Essentials*. Community Experience Distilled. Packt Publishing, 2015. ISBN: 9781785280801. URL: http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx%3fdirect%3dtrue%26db%3dnlebk%26AN%3d1131205%26site%3dehost-live.

[Hei22a] Prof. Dr. Christian Heinlein. *Kernteile eines MOSTflexiPL-Compilers*. 2022.

[Hei22b] Prof. Dr. Christian Heinlein. *LibCH C++-Bibliothek*. 2022.

[GCC] GCC Team. *Status of Supported Architectures from Maintainers' Point of View*. URL: https://gcc.gnu.org/backends.html (besucht am 18. 04. 2022).

[LLVa] LLVM Team. *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html (besucht am 04. 06. 2022).

[LLVb] LLVM Team. *LLVM Users*. URL: https://llvm.org/Users.html (besucht am 21. 04. 2022).

[LLVc] LLVM Team. *ORC Design and Implementation*. URL: https://www.llvm.org/docs/ORCv2.html (besucht am 09. 08. 2022).

[LLVd] LLVM Team. *Performance Tips for Frontend Authors*. URL: https://www.llvm.org/docs/Frontend/PerformanceTips.html#the-basics (besucht am 17. 07. 2022).