Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

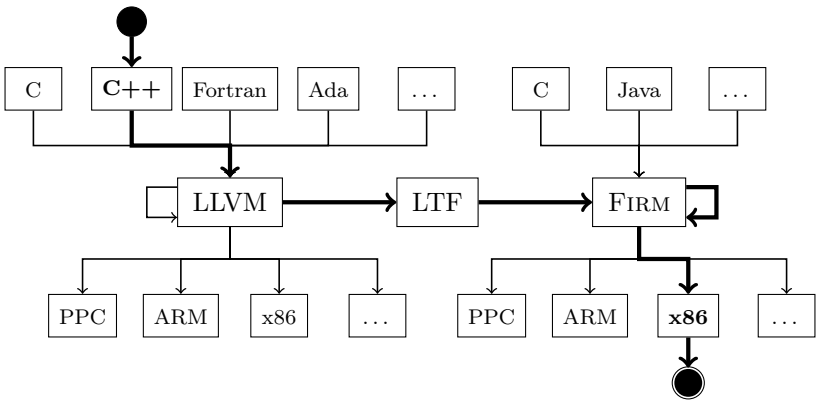
Lehrstuhl Prof. Dr.-Ing. Snelting

Transformation SSA-basierter Zwischensprachen

Studienarbeit von

Olaf Liebe

an der Fakultät für Informatik



Gutachter: Prof. Dr.-Ing. Gregor Snelting

Betreuender Mitarbeiter: Dipl.-Inform. Matthias Braun

Bearbeitungszeit: 4. Dezember 2009 – 30. März 2010

Zusammenfassung

Das an der University of Illinois at Urbana Champaign ins Leben gerufene LLVM-Projekt bietet eine breite Infrastruktur rund um eine zentrale SSA-basierte Zwischensprache an. Eine ähnliche Programmdarstellung, die sich vor allem durch ihren graphbasierten Ansatz auszeichnet, wurde mit Firm an der Universität Karlsruhe entwickelt.

Diese Arbeit zeigt, dass eine Transformation der LLVM-Zwischensprache nach Firm es ermöglicht, die vorhandene Infrastruktur auch für Firm zu erschließen. Dadurch lassen sich verschiedene LLVM-Frontends und auch die Optimierungsphasen erfolgreich mit Firm kombinieren, wodurch C-, Fortran- und C++-Programme übersetzt werden können, ohne dass hierfür ein eigenes Firm-Frontend zum Einsatz kommen muss. Es ist anzunehmen, dass dies auch für weitere LLVM-Frontends gelingt.

Darüber hinaus ermöglicht die Transformation zwischen beiden Zwischensprachen einen direkten Vergleich der sprachneutralen Optimierungsphasen und architekturspezifischen Backends, wodurch neue Einsichten über die Effektivität der Optimierungen und die Vor- und Nachteile der jeweiligen Programmdarstellung gewonnen werden können. Eine Evaluierung anhand der CPU2000 Benchmark-Suite wurde im Rahmen der Arbeit ebenso durchgeführt.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Übersetzer-Zwischensprachen in SSA-Form gewinnen nicht nur in der Forschung zunehmend an Bedeutung, sondern werden vielfach auch industriell eingesetzt. Eine umfangreiche Infrastruktur auf dieser Basis bietet das an der University of Illinois at Urbana Champaign ins Leben gerufene LLVM-Projekt[1] [LLVa]. Bekannte Anwendungsbeispiele aus der Industrie sind Apples und Nvidias OpenCL[2]-Compiler oder Adobes „Pixel Bender“, welches als Programmiersprache für Filter in After Effects CS4 und Flash Player 10 zu Einsatz kommt. Darüber hinaus verwendet Apple LLVM, um verschiedene Teile der OpenGL-Pipeline zu beschleunigen und gegebenenfalls Shader per Software zu emulieren. Weitere Beispiele finden sich auf der LLVM-Webseite [LLVb].

Es ist daher offensichtlich, dass LLVM längst aus dem rein akademischen Stadium hinausgewachsen ist. Damit bietet es sich als Vergleichsobjekt für ähnliche Projekte, wie die an der Universität Karlsruhe konzipierte SSA-Zwischensprache Firm [Lib] an. Ausgehend von diesem Gedanken ist eine Transformation der LLVM-Zwischensprache nach Firm von Interesse, um die Vor- und Nachteile beider Konzepte zu vergleichen. Darüber hinaus kann dies auch als Brücke dienen, um die vorhandene Infrastruktur auch in Firm zu nutzen und bereits bestehende LLVM-Frontends mit den sprachneutralen Optimierungen und Backends von Firm zu kombinieren (siehe Abbildung 1.1). Da für Firm zur Zeit nur ein C-Frontend [CPa] und ein Java-Frontend zur Verfügung stehen, wäre dies eine sinnvolle Ergänzung, insbesondere wenn hierdurch auch eine Kombination mit zukünftigen LLVM-Compilern möglich würde.

Unmittelbar betrifft dies die C++- und Fortran-Frontends, die LLVM durch eine Anpassung des GCC-Compilers bereits mitbringt und zu denen Firm zur Zeit kein entsprechendes Gegenstück bietet. Die Enwicklung einer entsprechenden Transformationsphase im Rahmen dieser Arbeit hat gezeigt, dass eine erfolgreiche Nutzung dieser Frontends möglich und auch praktikabel ist. Darüber hinaus ist abzusehen, dass dies auch für weitere bereits vorhandene oder zukünftige Frontends gelingen dürfte. Als mögliche Beispiele seien Python, Ruby, D, PHP, Lua, Objective-C oder Ada genannt. Dabei ist allerdings anzumerken, dass sich viele dieser Frontends noch in der aktiven Entwicklung befinden. Informationen hierüber lassen sich der LLVM-Webseite entnehmen [Pro].

Schlussendlich ergibt sich bei der Implementierung einer solchen Transformation immer

[1] Low Level Virtual Machine

[2] Open Computing Language, siehe [Khr09]

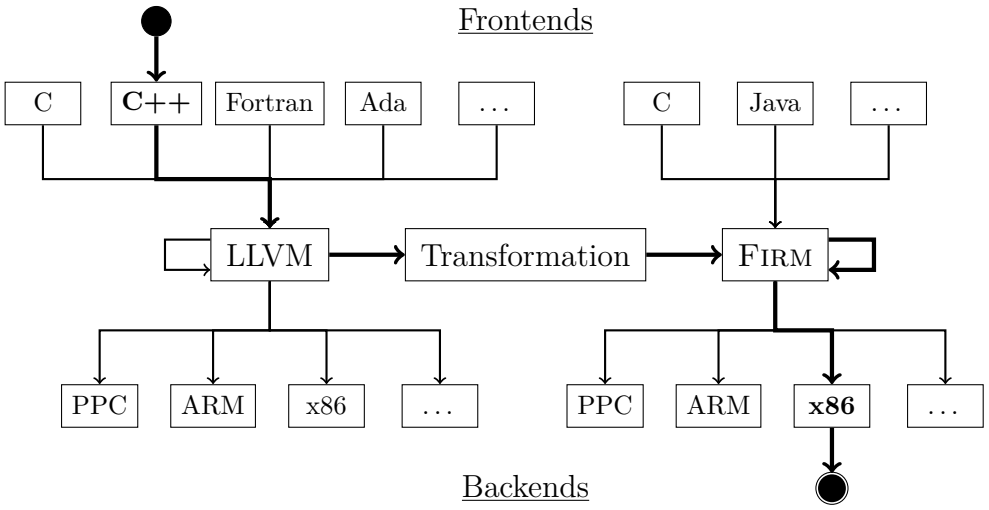


Abbildung 1.1: Transformation als Brücke zwischen LLVM und FIRM

auch ein direkter Vergleich der Eigenarten und Unterschiede beider Zwischensprachen, sowie der Effektivität vorhandener sprachneutraler Optimierungen und architekturspezifischer Backends. Eine Evaluierung der erzielten Ergebnisse und des Funktionsumfanges der Transformation wurde daher ebenso vorgenommen.

1.2 Aufbau der Arbeit

Kapitel 2 gibt zunächst einen kurzen Überblick über die theoretischen Hintergründe der SSA-Form und stellt den strukturellen Aufbau beider Zwischensprachen in Kurzform dar. Eine Betrachtung der für die Transformation relevanten und wesentlichen Unterschiede schließt das Kapitel ab. Ausgehend von diesen Betrachtungen wird in Kapitel 3 ein Konzept dargestellt, auf dessen Basis sich eine solche Transformation umsetzen ließe. Im Anschluss wird die Softwarearchitektur, welche das Konzept praktisch umsetzt, vorgestellt und ein kurzer Überblick über den Transformationsprozess gegeben. Kapitel 4 widmet sich den Details der Implementierung. Dabei wird beschrieben, wie diverse problematische Konstrukte nach FIRM übertragen werden und in welcher Form dies implementiert wurde. Gegebenenfalls werden weitere implementierungsspezifische Aspekte genauer vorgestellt. In Kapitel 5 erfolgt eine Evaluierung der erzielten Ergebnisse, sowohl in Bezug auf die Vollständigkeit der Transformation, als auch in Bezug auf die dabei erzielten Laufzeit-Ergebnisse, anhand der SPEC CPU2000 Benchmark-Suite [SPEa]. Im letzten Kapitel werden die Ergebnisse zusammengefasst und ein Ausblick darauf gegeben, welche weiteren Möglichkeiten und Chancen sich ergeben.

2 Grundlagen

Der folgende Abschnitt liefert zunächst einen Überblick über Steuerflussgraphen, die SSA-Form und die Umsetzung in den Zwischensprachen LLVM und Firm. Anschließend werden die wesentlichen Unterschiede beider Zwischensprachen dargestellt.

2.1 Steuerflussgraphen

LLVM und Firm verwenden Steuerflussgraphen für die Darstellung des Zwischencodes im Speicher. Dabei werden die Instruktionen zu sogenannten Grundblöcken gruppiert, welche die Knoten des Steuerflussgraphen darstellen. Hierbei gilt: sobald eine Instruktion im Grundblock ausgeführt wird, werden garantiert auch alle anderen Instruktionen dieses Blockes ausgeführt. Dies wird erreicht, indem Sprungbefehle stets nur am Ende eines Grundblockes stehen dürfen und dabei nur einen anderen Grundblock als Ziel haben, nicht jedoch eine spezifische Instruktion in diesem Block.

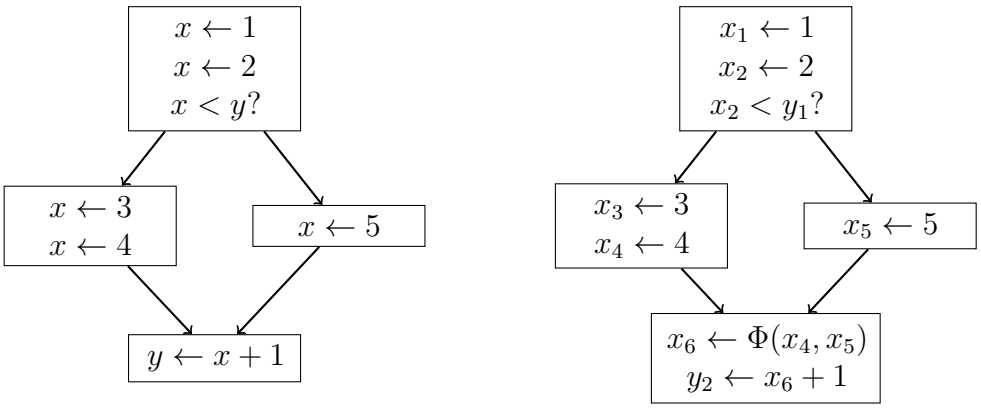
Damit stellen Grundblöcke die kleinste durch den Steuerfluss zu berücksichtigende Einheit dar. Üblicherweise wird für jede Funktion im Programm ein getrennter Steuerflussgraph verwendet, der über ausgezeichnete Start- und Endknoten betreten oder verlassen wird.

2.2 SSA-Form

Die SSA[1]-Form ist eine durch Rosen, Wegman und Zadeck [RWZ88] maßgeblich geprägte Programmdarstellung, bei der jeder Variablen nur durch genau eine Zuweisung ein Wert zugewiesen wird. Aufgrund dieser Eindeutigkeit bezeichnet man die Variablen meist auch einfach als „Werte“ in der SSA-Form. Dadurch wird eine Vereinfachung vieler Optimierungs- und Analysephasen erreicht, da bei jeder Verwendung eines Wertes die zugehörige Definition direkt und eindeutig bestimmt werden kann und keine umfangreiche Datenflussanalyse erforderlich ist. Es ist dabei zu beachten, dass eine Zuweisung zwar mehrfach durchlaufen werden kann, es sich statisch betrachtet jedoch stets um dieselbe Instruktion handelt, daher auch *Static* Single Assignment.

Die Transformation in SSA-Form wird erreicht, indem bei jeder Zuweisung ein neuer, bisher nicht verwendeter Variablenname vergeben wird, um den zugewiesenen Wert zu bezeichnen. Da der Variablenname nur hier zum Einsatz kommt, ist die Zuordnung einer

[1] Static single assignment



(a) Keine SSA-Form (b) In SSA-Form

Abbildung 2.1: Überführung in die SSA-Form

Definition stets eindeutig. Betrachtet man allerdings die Steuerflussgraphen, so kann es immer dort, wo der Steuerfluss in einem Block zusammenläuft (siehe Abbildung 2.1), zu einer statisch nicht eindeutigen Definition kommen. Dies liegt daran, dass in den verschiedenen Ursprungsblöcken auch verschiedene Definitionen derselben ursprünglichen Variablen vorliegen können. Die tatsächliche Definition hängt dann vom jeweiligen Steuerfluss ab.

Um dieses Problem zu lösen, wird eine spezielle Φ-Funktion eingeführt, die je nach vorliegendem Steuerfluss die korrekte Definition des Wertes auswählt. Es handelt sich dabei um keine klassische Instruktion, die in Maschinencode übersetzt werden kann, sondern lediglich um ein Hilfskonstrukt, mit dem das Backend die spätere Verknüpfung zwischen den verschiedenen Definitionen herstellen kann. Ein effizienter Algorithmus von Cytron et. al. zur Konstruktion der SSA-Form wird in [CFR+91] vorgestellt.

Obwohl es streng genommen nicht erforderlich ist, wird bei Betrachtung der SSA-Form meist die sogenannte Dominanz-Eigenschaft[2] [Hac06] vorausgesetzt. Diese trifft eine Aussage über die Dominanz-Beziehung zwischen den Definitionen von SSA-Werten und deren Verwendung. Es gilt:

Definition 1 (Dominanz-Relation) *Für zwei Instruktionen x und y im Steuerflussgraphen gilt: x dominiert y genau dann, wenn jeder vom Startblock ausgehende Pfad, der y erreicht, auch durch x führt.*

Definition 2 (Dominanz-Eigenschaft) *Die Definition D_x einer Variablen x dominiert alle Verwendungen $y = op(\ldots, x, \ldots)$ dieser Variablen, außer wenn $op = \Phi$. In diesem Fall dominiert D_x jedoch die jeweils vorangehende (Sprung-)Instruktion im zugehörigen Vorgänger-Block.*

[2] engl. „dominance property“

Ist diese Eigenschaft erfüllt, spricht man auch von der strikten SSA-Form. Sie stellt sicher, dass Variablen bei ihrer Verwendung stets definiert sind. Im Folgenden wird davon ausgegangen, dass die Dominanz-Eigenschaft erfüllt ist, da LLVM und Firm dies für die Korrektheit der Zwischendarstellung voraussetzen.

2.3 LLVM

Listing 1: Beispielhaftes LLVM-Modul

```
@a = private global i32 15                          ; Globale Variable @a
define i32 @func() nounwind {                           ; Funktion @func
entry:
  %0 = load i32* @a                                   ; Lade Variable @a
  %1 = icmp sle i32 %0, 10                           ; Vergleich %0 ≤ 10
  br i1 %1, label %exit, label %greater               ; Bedingter Sprung
greater:
  ; Es gilt %0 > 10
  %2 = add i32 %0, 10                                       ; Addiere 10
  br label %exit                                    ; Unbedingter Sprung
exit:
  ; Wähle %0 beim Sprung entry→exit und %2 bei greater→exit
  %3 = phi i32 [ %0, %entry ], [ %2, %greater ]
  ret i32 %3                                              ; Gebe %3 zurück
}
```

LLVM versteht sich selber nicht nur als Compiler-Zwischensprache, sondern als umfangreiche Infrastruktur zur Entwicklung von Compilern. Als solche stellt es – in Form von Programmen und C/C++-Bibliotheken – eine ganze Bandbreite von Funktionalitäten rund um eine zentrale Zwischensprache bereit und bietet eine umfassende Optimierungsstrategie für alle Phasen des Übersetzungsvorganges, sowie verschiedene Backends, diverse Analysephasen, inhärente Unterstützung für Garbage Collection und auch einen JIT-Compiler für verschiedene Zielplattformen.

Die Zwischensprache selbst ist dabei SSA-basiert und verwendet Befehlslisten, die sich in Form einer Assembler-Sprache oder auch in Binärdarstellung – als sogenannter Bitcode – ausgeben und bearbeiten lassen. Der strukturelle Aufbau einer auf diese Weise beschriebenen Übersetzungseinheit – in LLVM Modul genannt – gliedert sich dabei im Wesentlichen in eine Menge von globalen Variablen und Funktionen, sowie gegebenenfalls die Definition und Benennung diverser Typen. Die Typen selber unterteilen sich in primitive Typen, wie Integer und Floating-Point-Typen, sowie abgeleitete Typen, wie Zeiger, Arrays, Strukturen, Vektoren und Funktionstypen. Funktionen und globale

Variablen werden ebenso typisiert wie die Werte der lokalen Variablen und Ergebnisse von Instruktionen innerhalb der Funktionen.

Eine im Modul implementierte Funktion ist hierbei für die Darstellung als Steuerflussgraph in mehrere Grundblöcke unterteilt, die jeweils als strikt geordnete Befehlslisten umgesetzt sind. Dabei gibt es genau einen Startblock, der mit der Funktion betreten wird. Jeder Instruktion der Befehlsliste, die einen Rückgabewert produziert, wird dabei entsprechend der SSA-Form genau ein Name oder eine einfache Nummer zugeordnet, die innerhalb der Funktion eindeutig ist. Am Ende jedes Blocks steht eine terminierende Instruktion, die den weiteren Steuerfluss festlegt. Entsprechend den Φ-Funktionen der SSA-Form existiert in LLVM eine `phi`-Instruktion, mittels der abhängig vom Steuerfluss eine Selektion aus mehreren Variablen ermöglicht werden kann.

Listing 1 zeigt eine einfache Beispielfunktion in LLVM-Assembler. Eine umfangreiche Beschreibung der Zwischensprache findet sich unter [LA10], ein Überblick über LLVM ist der Master-Arbeit von Chris Lattner [Lat02] oder in aktuellerer Form [LA04] zu entnehmen.

2.4 Firm

Firm ist eine graphbasierte Zwischensprache, die ebenso wie LLVM eine SSA-Darstellung verwendet und am IPD Goos ins Leben gerufen wurde (siehe auch [TLB99]). Sie basiert unter anderem auf Ideen von Click und Paleczny [CP95] zur Zwischendarstellung von Programmen. Eine Implementation dieser Zwischensprache in C stellt die libFirm-Bibliothek [Lin02] dar, die ähnlich den LLVM-Bibliotheken Analyse- und Optimierungsphasen, Backends und diverse weitere Hilfsmittel rund um die Speicherrepräsentation von Firm bereitstellt.

Typ-Graph

Die Grundlage für ein Programm bildet in Firm der sogenannte Typ-Graph. Er enthält alle im Programm definierten Typen als Knoten und verbindet sie gegebenenfalls über Kanten, um Beziehungen zwischen den Typen zu modellieren. Dabei gibt es wie bei LLVM für sich allein stehende primitive Typen wie z. B. Integer und Floating-Point-Typen, sowie zusammengesetzte Typen die durch Komposition gebildet werden. Die Kompositionsbeziehung wird explizit durch Kanten hergestellt.

Um dies zu ermöglichen, gibt es sogenannte Entity-Knoten, welche als Basis für diese Typ-Komposition zum Einsatz kommen. Sie repräsentieren typisierte „Objekte“, die zur Laufzeit im Speicher abgelegt werden und damit adressiert werden können. Aus diesem Grunde erfassen sie zusätzliche Informationen über das Speicherlayout, wie z. B. die relative Adresse in einer Struktur, das Alignment oder einen Initialisierungswert – in Form eines weiteren Graphen – und verknüpfen sie durch eine Kante mit dem Typ der jeweiligen Daten.

Dabei gibt es in Firm verschiedene Struktur-Typen: gewöhnliche Strukturen fassen mehrere Entities zusammen (d. h. ihr Typ-Knoten ist auf dem Typ-Graphen mit mehreren Entity-Knoten verbunden), die sich jedoch weder überlappen noch einen Methodentyp besitzen dürfen. Eine Überlappung ist mit dem Union-Typ möglich (ähnlich dem Union-Typ in C/C++), während der Klassen-Typ auch Entities mit Methodentypen erlaubt, eine Überlappung jedoch ebenfalls verbietet. Der Array-Typ verwendet nur ein einziges Entity, welches alle möglichen Einträge repräsentiert.

Von besonderem Interesse ist dabei der sogenannte „Global-Type“. Dabei handelt es sich um einen speziellen Klassen-Typ, der in jedem Firm-Programm enthalten ist. Er enthält für alle globalen Variablen und Methoden entsprechende Entities, wobei für jede im Programm implementierte Methode, vergleichbar mit einem Initialisierungswert, ein Methodengraph im Entity hinterlegt ist.

Methoden-Graphen

Die Methoden-Graphen beinhalten die vollständige Zwischendarstellung einer Methode und bilden zunächst einen Steuerflussgraphen. Sie sind daher in mehrere Grundblöcke unterteilt, wobei explizite Start- und Endblöcke zu Einsatz kommen. Die bereits in Listing 1 vorgestellte LLVM-Funktion ist zum Vergleich in Abbildung 2.2 als Firm-Graph dargestellt (erstellt mit Hilfe des `ycomp`-Programmes [HGB+]).

Innerhalb der Blöcke stellen sich die Instruktionen in Form verschiedener Knoten dar, die durch Abhängigkeitskanten miteinander verbunden sind. Im Gegensatz zu dem von LLVM verwendeten klassischen Ansatz existiert in Firm keine Totalordnung zwischen den einzelnen Instruktionen. Stattdessen wird durch die Abhängigkeiten zwischen den Knoten lediglich eine Halbordnung erzwungen, so dass die Knoten davon abgesehen einfach lose im jeweiligen Block „schweben“. Man könnte auch sagen, dass die Darstellung in Firm alle möglichen Totalordnungen gleichzeitig repräsentiert und somit eine zusätzliche Abstraktionsebene bietet, während sich LLVM bereits hier auf eine strikte Befehlsanordnung festlegt, obwohl diese eigentlich erst im Backend erforderlich ist.

Instruktions-Knoten Die hierbei verwendeten Instruktions-Knoten sind mit den Instruktionen in LLVM vergleichbar. Beispielsweise ist ein `Add`-Knoten über Abhängigkeitskanten mit zwei weiteren Knoten verbunden, deren Ergebniswerte er addiert, während er seinerseits das Resultat zurückliefert. Dieses ist anders als bei LLVM nicht über das bereits dargestellte Typensystem typisiert. Stattdessen besitzt jede Kante einen sogenannten Modus, welcher im Wesentlichen primitive Typen bezeichnet, die in einem Register zwischengespeichert werden können. Beispiele hierfür sind **Iu** (32 Bit vorzeichenloser Integer), **Bs** (8 Bit vorzeichenbehafteter Integer), **P** (Zeiger) oder **D** (Double-Precision Floating-Point). Liefert ein Knoten mehrere Werte, so produziert er ein Tupel mit Modus **T**. Die einzelnen Werte können dann über einen sogenannten `Proj`-Knoten mit Hilfe des jeweiligen Index aus dem Tupel „herausprojiziert“ werden.

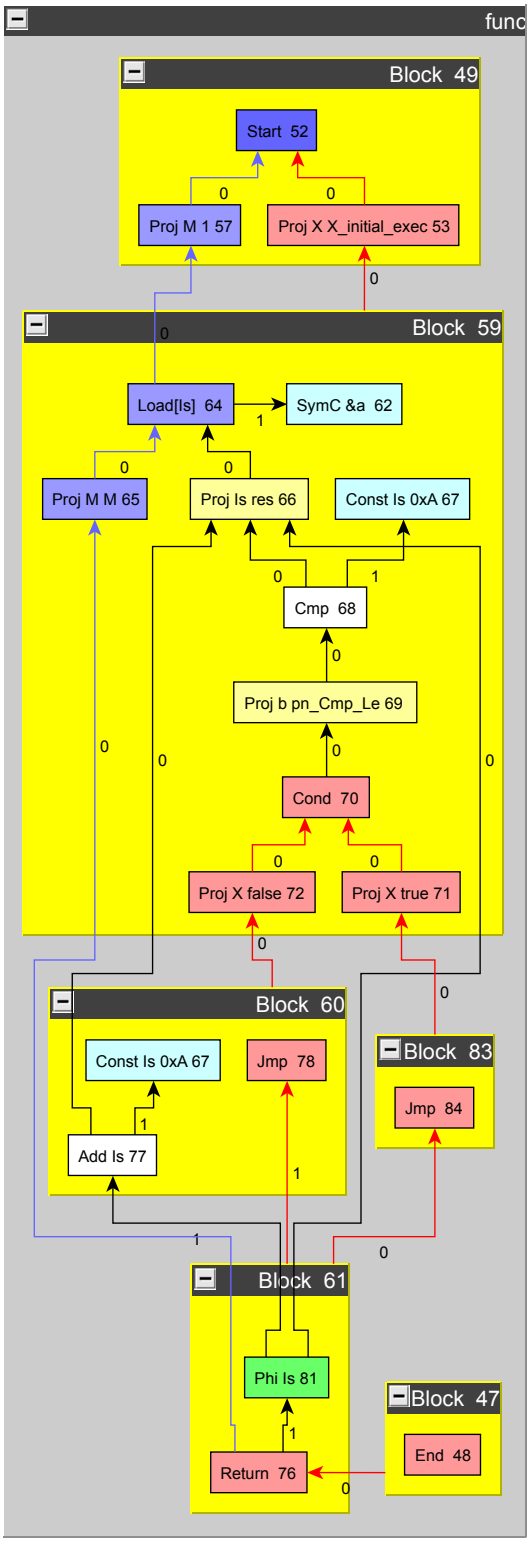


Abbildung 2.2: Beispielhafter Firm-Methodengraph

Zusätzliche Kanten Davon abgesehen gibt es einige Modi, denen eine spezielle Bedeutung zukommt. So wird z. B. der Steuerfluss zwischen den Blöcken durch Kanten mit Modus **X** festgelegt, die vom jeweiligen Zielblock zum entsprechenden Sprung-Knoten zeigen (da Firm-Graphen Abhängigkeiten beschreiben). Ein bedingter Sprung erzeugt dementsprechend mehrere „Resultate“ mit Modus **X**, also ein Tupel aus dem sich mittels `Proj`-Knoten die unterschiedlichen Sprungziele herausgreifen und mit den entsprechenden Zielblöcken verbinden lassen (vgl. `Cond`-Knoten 70 im Graphen).

Die von Firm über die Datenabhängigkeiten implizit festgelegte Halbordnung ist jedoch noch unzureichend. Beispielsweise kann sie keine Speicherzugriffe oder Funktionsaufrufe ordnen, wenn keine explizite Datenabhängigkeit zwischen ihnen besteht. Aus diesem Grunde besitzen Knoten, deren Reihenfolge aufgrund von Speicherzugriffen, Ein- und Ausgabe etc. eine Rolle spielt, eine zusätzliche Abhängigkeit vom Modus **M** (Memory) und produzieren ihrerseits wieder einen entsprechenden Wert, der im Prinzip den aktuellen Speicherzustand repräsentieren soll. Der erste Memory-Wert wird dabei vom `Start`-Knoten innerhalb des Startblocks produziert, welcher den Einsprungspunkt der Funktion darstellt. Von hier aus zieht sich die Speicher-Kante als roter Faden durch den Graphen, so dass durch die neuen Abhängigkeiten eine passende Ordnung erzwungen werden kann.

Phi-Knoten Durch die Verwendung der SSA-Form wird die Graphdarstellung erst praktikabel, da die jeweils verwendeten Werte (und damit die Ursprungsknoten der Kanten) hierdurch eindeutig werden. Jeder Knoten entspricht damit unmittelbar einem SSA-Wert. Die Φ-Funktionen werden durch entsprechende `Phi`-Knoten dargestellt, welche jeweils die eingehende Kante selektieren, deren Index mit dem Index der Sprungkante übereinstimmt, über welche der aktuelle Block erreicht wurde (vgl. Knoten 81 im Graphen). Die Memory-Werte vom Modus **M** sind ebenso SSA-Werte und müssen daher auch auf `Phi`-Knoten zurückgreifen.

2.5 Unterschiede

LLVM und Firm sind sich im strukturellen Aufbau relativ ähnlich, insbesondere aufgrund der gemeinsamen SSA-Form und der vergleichbaren konzeptionellen Ausrichtung. In Bezug auf eine Transformation von LLVM nach Firm sind dabei vor allem solche Fälle von Bedeutung, in denen der Funktionsumfang von LLVM den von Firm übersteigt, so dass die entsprechenden LLVM-Konstrukte entweder in Firm emuliert werden müssen oder eine Änderung in libFirm erforderlich machen.

Am augenfälligsten ist sicher zunächst die teilweise ungeordnete Graph-basierte Darstellung in Firm, welche den Befehlslisten in LLVM entgegensteht. Glücklicherweise impliziert aber die Anordnung der Instruktionen in LLVM automatisch auch eine Halbordnung, so dass es ausreicht, die Firm-Knoten entsprechend der von LLVM gegebenen Reihenfolge zu konstruieren. libFirm ist in der Lage, automatisch entsprechend der Konstruktionsreihenfolge Speicher-Kanten einzufügen, sollte dies erforderlich sein. An-

sonsten ist die Befehlsanordnung über die Datenabhängigkeiten implizit vorgegeben. Die Aufteilung der Funktionen in Blöcke, sowie die Verknüpfung der Blöcke mittels Sprunginstruktionen lässt sich weitestgehend analog von LLVM auf Firm übertragen.

Typsystem Als schwieriger stellt sich bei genauerer Betrachtung das Typ-System heraus. So bietet Firm zwar bei den zusammengesetzten Datenstrukturen mit mehrdimensionalen Arrays, Union-Typen und Klassen weitaus mehr Möglichkeiten als LLVM, dafür ist es allerdings notwendig, für eben jene Strukturen ein genaues Layout festzulegen, während LLVM dies weitestgehend dem Backend überlässt. Insbesondere stellt aber der Integer-Typ in LLVM ein Problem dar, da er eine beliebige Bit-Breite aufweisen darf, während sich Firm – praktisch gesehen – auf die üblichen Bit-Breiten von 8, 16, 32 und 64 Bit beschränkt. Entsprechende Typen und Modi lassen sich zwar in Firm definieren, werden allerdings zur Zeit nicht vom Backend unterstützt. Dies wirft gleich mehrere Probleme auf, wie z. B. die Implementierung der nötigen Modulo-2^n-Arithmetik in Firm, sollte der Integer von den üblichen Bit-Breiten abweichen oder die Umsetzung von Integer-Typen mit mehr als 64 Bit. Kritisch ist auch die Handhabung von booleschen Werten: diese werden in LLVM als 1-Bit-Integer darstellt, während Firm hier einen gesonderten Modus und Typ vorsieht. Dieser kann zwar für boolesche Operationen und als Bedingung verwendet werden, erlaubt jedoch keine arithmetischen Operationen wie in LLVM. Dadurch ist die Zuordnung eines Firm-Modus nicht immer eindeutig und vom jeweiligen Kontext abhängig.

Ein weiterer mit dem Typ-System verbundener Unterschied ist außerdem die Verwendung der Modi im Firm-Graphen, während LLVM hier weiterhin das vorhandene strenge Typ-System einsetzt. In den meisten Fällen stellt dies kein großes Problem dar, da die Modi in Firm generischer sind, als die entsprechenden Typen, jedoch betrifft dieser Umstand nur primitive Typen und Zeiger. Zusammengesetzte Strukturen können in LLVM durchaus als Werte lokaler Variablen auftreten, während – abgesehen vom Tupel – keine zusammengesetzten Firm-Modi existieren, die als Ersatz dienen könnten.

Instruktionen LLVM-Instruktionen lassen sich in vielen Fällen in ähnlicher Form in Firm wiederfinden, auch wenn die Instruktionen in Kombination mit dem flexiblen Integer-Typ häufig über die unmittelbar von Firm bereitgestellte Funktionalität hinausgehen. Unterschiede bestehen außerdem bei der Handhabung von lokalem Speicher auf dem Stack, der in LLVM stets über eine `alloca`-Instruktion reserviert wird, während Firm hier einen gesonderten Klassen-Typ verwendet, um den (statischen) Stack-Frame zu modellieren. Davon abgesehen verfolgt LLVM bei Typ-Konversionen einen anderen Ansatz als Firm und bietet eine Vielzahl von Konvertierungs-Instruktionen, denen in Firm ein einziger `Conv`-Knoten entgegensteht, um deren Funktionalität umzusetzen.

Abgesehen von den Instruktionen stellt LLVM zudem über sogenannte „Intrinsic-Funktionen“ einen erweiterbaren Mechanismus bereit, über den weitere relevante Mechanismen, wie z. B. die Handhabung von Funktionen mit variabler Parameteranzahl,

implementiert werden. Diese Intrinsics werden wie gewöhnliche Funktionen aufgerufen, erfordern allerdings eine gesonderte Behandlung, die in vielen Fällen mit einer eigenen Instruktion vergleichbar ist. Auf diesem Wege werden auch viele erweiterte Funktionalitäten, wie die Einbettung von Debug-Informationen, das Exception-Handling oder die Verwendung der Garbage Collection in LLVM integriert. Zwar gibt es in Firm einen vergleichbaren Mechanismus, allerdings werden die bei der Transformation berücksichtigten Intrinsics in Firm meist anders umgesetzt.

Weitere Unterschiede In Bezug auf den strukturellen Aufbau des Zwischencode-Programmes stellten insbesondere die umfangreichen Linker-Informationen, die LLVM bereitstellt, ein Problem dar, da es in libFirm aufgrund der bisher fehlenden C++-Unterstützung teilweise keine direkte Entsprechung gab und somit weitere Anpassungen notwendig waren. Dieser Umstand sollte jedoch mit den neueren libFirm-Versionen behoben sein. Darüber hinaus ist im Zusammenhang mit C++ auch das in LLVM implementierte Zero-Cost Exception-Handling (siehe [CCE+]) erwähnenswert, dem in libFirm leider zur Zeit kein direktes Äquivalent entgegensteht.

3 Entwurf

Dieses Kapitel gibt zunächst Aufschluss über die Konzeption des Transformationsvorganges und stellt die – im Rahmen der Arbeit entwickelte – Softwarearchitektur vor, welche dieses Konzept konkret umsetzt.

3.1 Konzeption

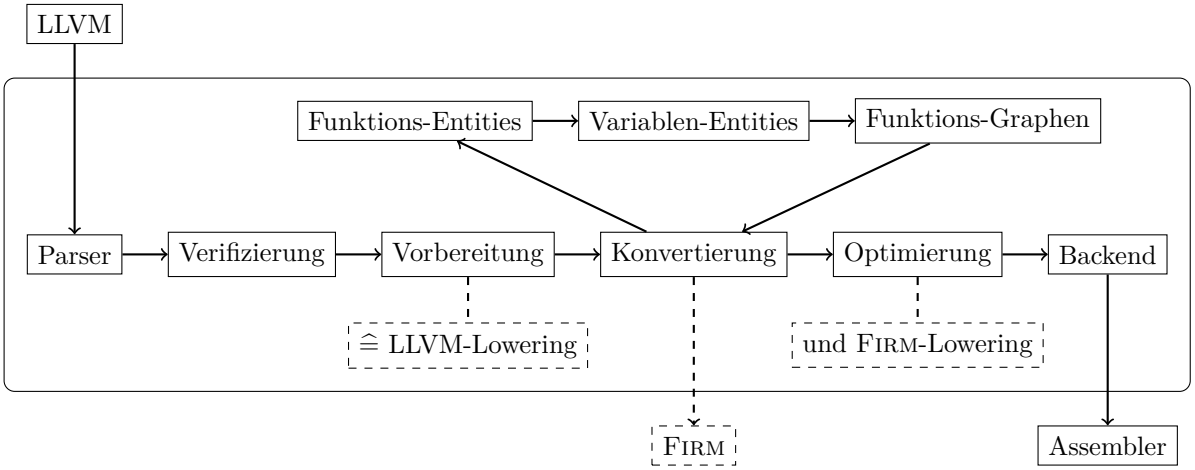


Abbildung 3.1: Pipeline-Architektur von LTF

Architektur Der Transformationsvorgang erfolgt über eine Pipeline-Architektur, ähnlich der klassischen im Compilerbau verwendeten Architektur. Der genaue Aufbau der Pipeline ist Abbildung 3.1 zu entnehmen. Die Parsing-Phase steht hier als Überbegriff für lexikalische, syntaktische und teilweise auch semantische Analyse. Dies liegt daran, dass bei der Implementation weitestgehend auf die entsprechenden LLVM-Bibliotheken zurückgegriffen wurde, welche den Parsing-Vorgang kapseln und direkt eine Speicherdarstellung des LLVM-Moduls zurückliefern.

Zu beachten ist weiterhin, dass die hier als Konvertierung bezeichnete Phase zwar mit der üblichen Transformationsphase vergleichbar ist, aber als Eingabe bereits eine Zwischendarstellung erhält, die vergleichsweise wenig abstrakt ist. Dennoch gibt es

einige Konstrukte, die sinnvollerweise vor der eigentlichen Konvertierung umgeformt oder vereinfacht werden sollten, um die anschließende Konvertierung nach Firm zu erleichern. Dies geschieht in der Vorbereitungsphase, welche auf dem LLVM-Zwischencode operiert und einige Konstrukte auf vergleichbare Operationen herunterbricht, die idealerweise eins zu eins in Firm übertragen werden können. Da als Ausgabe weiterhin LLVM-Zwischencode verwendet wird, ist dies natürlich nicht immer möglich.

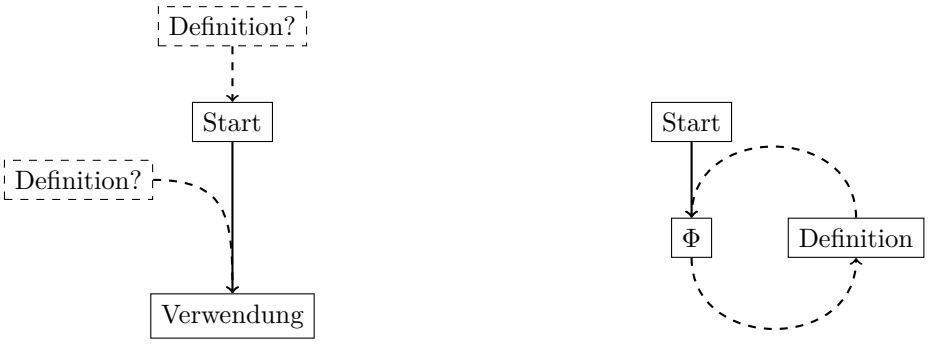
Der Konvertierungsvorgang selbst gliedert sich im Wesentlichen in drei Phasen: Zunächst werden Funktionsdeklarationen als Entities des „Global Type“ auf Firm übertragen, danach erfolgt eine ähnliche Übersetzung der globalen Variablen, so dass zunächst das Rahmenwerk in der jeweiligen Übersetzungseinheit geschaffen wird. Damit stehen in der nächsten Phase, nämlich der Übersetzung der Befehlslisten, in Firm-Graphen alle globalen Entities zur Verfügung, so dass bei Bedarf ein Zugriff auf sie möglich ist.

Anschließend an die Konvertierungsphase erfolgt die Optimierungsphase, in der die libFirm-Optimierungen auf den generierten Firm-Zwischencode angewendet werden und gegebenenfalls komplexe Firm-Konstrukte für das Backend heruntergebrochen werden. Eine strikte Trennung von beiden Prozessen erfolgt hier nicht, da Reihenfolge und Verkettung von Optimierungen und Lowering-Phasen als Teil der Optimierungsstrategie angesehen werden. Ein Austausch oder eine Anpassung dieser Strategie ist ebenfalls möglich. Dies gilt auch für die Backend-Phase, die im Moment jedoch im Wesentlichen auf die Funktionalität von libFirm zurückgreift.

Funktionsgraphen Bei der Übersetzung der Instruktionen selbst bietet es sich nun innerhalb der jeweiligen Blöcke an, diese in der von LLVM gegebenen Reihenfolge zu bearbeiten. Durch die strikte Befehlsanordnung von LLVM kann dann zugesichert werden, dass stets alle Abhängigkeiten konvertiert werden und damit zur Verfügung stehen, bevor die Instruktionen verarbeitet werden, welche jeweils darauf zugreifen. Fraglich ist allerdings, ob und inwiefern sich diese Zusicherung auf blockübergreifende Abhängigkeiten und insbesondere `phi`-Instruktionen übertragen lässt. Um dies zu erreichen, muss man natürlich auch die Blöcke in einer passenden Reihenfolge durchlaufen, die jedoch in LLVM zunächst willkürlich sein kann, da die Sprung-Instruktionen immer explizit den Zielblock benennen und die Anordnung der Blöcke somit keine Rolle spielt.

Glücklicherweise kann dies durch eine Neuordnung der Blöcke leicht behoben werden, wenn man die `phi`-Instruktionen zunächst aus den Betrachtungen ausschließt. Dabei ist es ausreichend, ausgehend vom Startblock alle Blöcke entlang der Steuerflusskanten zu durchlaufen, z. B. mittels Tiefensuche. Hierbei gilt: wird bei der Traversierung die Verwendung einer Variablen erkannt, deren Definition bisher nicht durchlaufen wurde, so ergeben sich zwei Möglichkeiten: entweder die Definition erfolgte vor dem bisher betrachteten Pfad oder aber sie erfolgte auf einem zweiten Pfad, der in den aktuellen Pfad mündet (siehe Abbildung 3.2 (a)).

Da LLVM allerdings keinen Sprung auf den Startblock erlaubt und der betrachtete Pfad dort beginnt, kann eine vorherige Definition ausgeschlossen werden, so dass nur noch



(a) Verwendung ohne Definition (b) Zyklische Abhängigkeit

Abbildung 3.2: Spezielle Steuerfluss-Situationen

eine Definition außerhalb des betrachteten Pfades in Frage kommt. Dies würde allerdings die Dominanz-Eigenschaft (siehe Kapitel 2.2) verletzen, da eine solche Definition die hier betrachtete Verwendung nicht dominieren würde. Somit kann diese Situation mit der strikten SSA-Form nie auftreten und es kann mittels Tiefensuche garantiert werden, dass bei jeder besuchten Instruktion – mit Ausnahme der `phi`-Instruktionen – stets auch alle für die Konstruktion erforderlichen Definitionen durchlaufen wurden. Dies ist auch anschaulich klar, da der jeweils verwendete Pfad ausgehend vom Startblock stets auch einen möglichen Programmablauf darstellt und in einem solchen Fall natürlich alle Variablen bei der Verwendung bekannt sein müssen.

Phi-Knoten Offen bleibt jedoch die Behandlung der `phi`-Instruktionen. Hier kann mit der gleichen Vorgehensweise nicht garantiert werden, dass auf Anhieb alle Abhängigkeiten erfüllt werden können, da die Dominanz-Eigenschaft hier nicht in gleicher Form anwendbar ist. Damit stellt sich das Problem, dass zwar für die Konstruktion auf einen Schlag alle Definitionen erforderlich sind, jedoch nicht notwendigerweise alle ankommenden Pfade erkundet wurden. Solch ein Fall tritt z. B. auf, wenn der weitere Pfad einen Zyklus bildet und wieder in den aktuellen Block mündet (Abbildung 3.2 (b)). Es bietet sich daher an, zunächst einen Platzhalter-Knoten zu erzeugen, der zwar keinerlei Abhängigkeiten benötigt, seinerseits aber als Abhängigkeit für andere Knoten verwendet werden kann. Sobald nun alle anderen Instruktionen konvertiert wurden, erhält man Zugriff auf die bisher fehlenden Abhängigkeiten. Somit ist es nun möglich, in einer zweiten Iteration sämtliche Platzhalter durch die jeweiligen `Phi`-Knoten zu ersetzen, denn nun stehen alle Abhängigkeiten als Knoten oder – bei einer Verkettung von `Phi`-Knoten – als Platzhalter zur Verfügung. Da nach der zweiten Iteration jedoch alle Platzhalter durch `Phi`-Knoten ersetzt wurden, wäre die Übersetzung aller Instruktionen hiermit vollständig.

Datenstrukturen Für den Zugriff auf die übersetzten Knoten bietet sich ein assoziativer Container an, welcher LLVM-Instruktionen und deren Firm-Äquivalent miteinander verknüpft. Dies könnte beispielsweise als Hashtabelle implementiert werden und ermöglicht es, die in der Speicherdarstellung von LLVM gegebenen Verknüpfungen und Verweise auf Firm-Knoten abzubilden. Dies ist auch für andere konstruierte Objekte sinnvoll, wie bei der Zuordnung zwischen den Typen beider Systeme oder zwischen den globalen Variablen in LLVM und den entsprechenden Entities in Firm.

Generell bietet sich bei der Übersetzung der Typen eine Konstruktion nach Bedarf an, wenn ein entsprechendes Firm-Konstrukt nicht in der Zuordnungstabelle zu finden ist. Im Gegensatz zu Variablen, Funktionen und Instruktionen kann man die Typen nicht auf einfache Weise vorab in Firm übertragen, da deren Verwendung sich auf das gesamte Programm verteilt und somit das Auffinden aller Typen eine Traversierung des gesamten Programmes erfordern würde oder aber den Aufbau einer entsprechenden Tabelle während des Einlesevorganges, was in der hier verwendeten Speicherdarstellung von LLVM aber nicht vorgesehen ist.

3.2 Softwarearchitektur

Der im Rahmen der Arbeit entwickelte Konverter namens LTF[1] ist grundsätzlich als wiederverwendbare Bibliothek konzipiert, welche den gesamten Transformationprozess von LLVM-Bitcode/-Assembler zum Assembler-Dialekt der Zielplattform via libFirm kapselt und es auch ermöglicht, nur bestimmte Teile der Pipeline zu verwenden, um z. B. direkt mit der Speicherdarstellung von libFirm weiterzuarbeiten oder ein LLVM-Modul direkt zu verwenden, anstatt es aus einer Datei einzulesen.

Zusätzlich zur Bibliothek wird mit `ltf-llc` ein Programm – ähnlich LLVMs `llc` – bereitgestellt, welches direkt die Zwischencodedateien von LLVM einlesen und per libFirm nach Assembler übersetzen kann. Eine Ausgabe des Firm-Zwischencodes ist ebenso möglich. Darüber hinaus existiert mit `ltf-gcc` ein Bash-Skript, welches mehrere Programme und insbesondere `ltf-llc` kombiniert, um ein GCC-ähnliches Interface anzubieten, welches als Compiler für C, C++ und Fortran verwendet werden kann. Allerdings ist eine Unterstützung von Exceptions in C++ derzeit nicht möglich.

Als Programmiersprache wurde aufgrund der Nähe zu LLVM C++ gewählt, so dass sowohl LLVM-, als auch libFirm-Bibliotheken problemlos verwendet werden können. Kontrolle und Steuerung erhält man über die `Converter`-Klasse, die als Fassade für die Bibliothek dient und direkt LLVM-Zwischencodedateien oder Module einliest und zu libFirm oder Assembler transformieren kann.

Parsing und Verifikation Ihrerseits verwendet die `Converter`-Klasse zunächst den Parser und die Verifikationsmechanismen der LLVM-Bibliotheken, um die ersten beiden

[1] Abkürzung für „LLVM to Firm“

Phasen der Pipeline umzusetzen und eine Speicherdarstellung von LLVM zu erhalten, mit welcher die weitere Verarbeitung erfolgen kann. Ein eigener Parser wäre zwar auch möglich, aber wenig sinnvoll, da die von LLVM erstellte Speicherdarstellung einfach navigierbar ist und primär lesend verwendet wird. Darüber hinaus erleichtert dies Anpassungen an zukünftige LLVM-Versionen.

Vorbereitungsphase Die Vorbereitungsphase bricht zunächst mehrere Instruktionen und Intrinsic-Funktionen herunter, die in libFirm direkt nicht vorhanden sind oder deren direkte Übersetzung bis jetzt nicht implementiert wurde. Als Beispiel seien die `frem`-Instruktion, die `memcpy`-Intrinsic oder die `sqrt`-Intrinsic genannt. Ein großer Teil des erforderlichen Lowerings lässt sich dabei direkt mittels der `IntrinsicsLowering`-Klasse von LLVM durchführen, wobei meist ein gewöhnlicher Funktionsaufruf die jeweilige Intrinsic oder Instruktion ersetzt, teilweise aber auch mehrere weitere Instruktionen erzeugt werden, z. B. im Falle von `bswap` oder `ctpop`. Darüber hinaus folgen noch Aufräumarbeiten, wie das Entfernen unbenutzter Funktionsdeklarationen, um unnötige Konvertierungen zu vermeiden.

Konvertierungsphase Die Konvertierung selber bildet den Schwerpunkt der Implementierung und erfolgt mittels mehrerer `Builder`-Objekte, die von einer gemeinsamen `Builder`-Klasse erben. Im Wesentlichen übersetzt ein `Builder` beim Aufruf der `retrieve`-Methode LLVM-Konstrukte durch die Konstruktion der jeweiligen Firm-Äquivalente und speichert die Zuordnung beider Speicherdarstellungen in einer lokalen Hashtabelle. Ist ein Wert bereits in der Hashtabelle verfügbar, so wird eine erneute Konstruktion unterbunden, so dass `retrieve` lediglich den entsprechenden Wert aus der Hashtabelle abruft und zurückgibt, was im Durchschnitt in konstanter Zeit erfolgen dürfte. Dabei kommt die Unordered-Map-Datenstruktur von Boost [MSJ] als Hashtabelle zum Einsatz, insbesondere, da diese mit C++0x auch Einzug in den C++-Standard halten wird. Die Zuordnungstabelle wird im Folgenden auch als Cache des `Builder`-Objektes bezeichnet.

Auf dieser Basis existieren sechs verschiedene `Builder`-Objekte, deren Interaktion und Aufbau in Abbildung 3.3 in gekürzter Form dargestellt ist. Dabei werden `Type-`, `Entity-` und `GraphBuilder` im globalen Kontext – d. h. über die Konstruktion der gesamten Übersetzungseinheit hinweg – verwendet, während `Block-`, `Node-` und `FrameBuilder` nur lokal für jeweils eine Funktion benutzt werden. Den Rahmen für den globalen Kontext jedes Programmes bildet die `Context`-Klasse, welche die global verwendeten `Builder` des jeweiligen Moduls zur Verfügung stellt und von der `Converter`-Klasse vor dem eigentlichen Konvertierungsvorgang initialisiert und an die weiteren `Builder`-Objekte weitergereicht wird.

Globale Entities Danach stößt die `Converter`-Klasse die Konstruktion sämtlicher globaler Variablen und Funktionsdeklarationen als Firm-Entities an, wobei über diese iteriert wird und jeweils eine Übergabe an den `EntityBuilder` stattfindet. Dabei werden

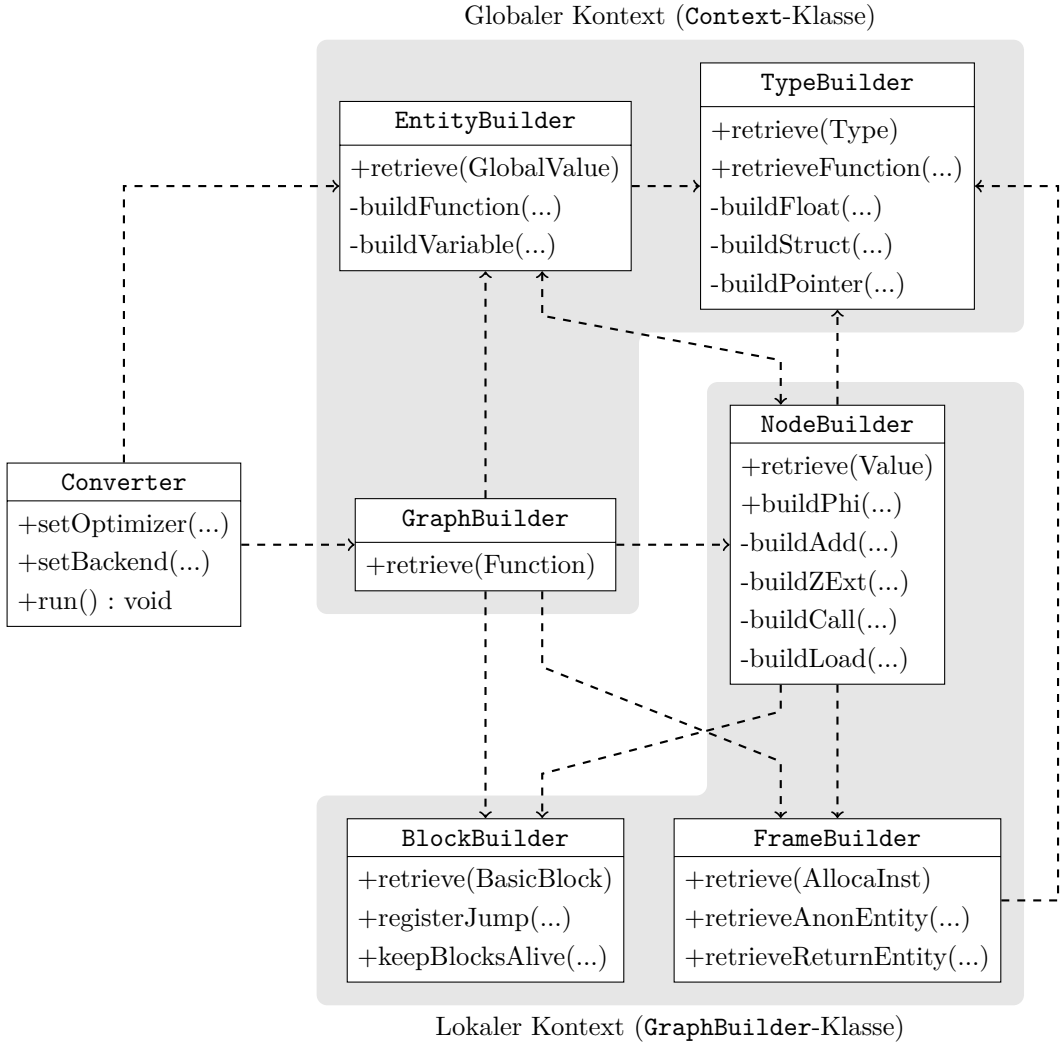


Abbildung 3.3: Builder-Objekte und -Ansteuerung in LTF

auch einige Sonderkonstruktionen, wie Konstruktor- und Destruktor-Funktionen abgefangen und umgesetzt. Hier wiederum kommt einerseits der `TypeBuilder` zum Einsatz, denn für die Konstruktion der jeweiligen Entities ist es notwendig, die entsprechenden Typen in Firm zu verwenden. Andererseits wird ein weiterer `NodeBuilder` verwendet, um Initialisierungs-Knoten für die globalen Variablen zu konstruieren.

Funktionsgraphen Sobald nun die Entities konstruiert wurden, wird der `GraphBuilder` vom `Converter` für jede Funktion mit lokaler Implementierung aufgerufen, um den Funktionsgraphen für das jeweilige Entity zu erzeugen und dort zu hinterlegen. Die Entities selbst kann der `GraphBuilder` ja nun vom `EntityBuilder` abrufen. Zur Konstruktion verwendet der `GraphBuilder` die lokalen `Builder`-Objekte, die nach der Konstruktion des Graphen – mitsamt ihres Cache – wieder zerstört werden (vgl. „lokaler Kontext“, Abbildung 3.3). Dabei wird zunächst mit Hilfe des `FrameBuilder` ein lokaler Frame-Typ konstruiert und über den `BlockBuilder` die Block-Struktur der Funktion aufgebaut. Zur Auflösung der Typen verwendet der `FrameBuilder` seinerseits den globalen `TypeBuilder`. Dann erfolgt die eigentliche Konstruktion des Graphen über den zuvor dargestellten zweistufigen Ablauf: zunächst werden die Instruktionen mit Hilfe des `NodeBuilder` übersetzt, wobei diesem alle lokalen und globalen `Builder` zur Auflösung von Verweisen auf Typen, Blöcke, Variablen etc. zur Verfügung stehen. Anschließend werden im zweiten Durchgang die `Phi`-Knoten konstruiert. Näheres dazu ist Kapitel 4.3 zu entnehmen.

Optimierungsphase Nach der Konstruktion aller Funktionsgraphen ist die eigentliche Konstruktionsphase abgeschlossen und die Optimierung des neu konstruierten Firm-Programmes kann beginnen. Hierzu verwendet die `Converter`-Klasse eine zuvor festgelegte Implementation der `Optimizer`-Klasse, welche eine bestimmte Optimierungsstrategie umsetzt und je nach Einsatzzweck durch eine andere Implementierung ersetzt werden kann. Eine an CParser [CPa] angelehnte – und weitestgehend statisch implementierte – Optimierungsstrategie wird durch die `CParserOptimizer`-Klasse umgesetzt und zur Zeit als Standard verwendet.

Backend Ähnlich der Optimierungsphase kann bei Verwendung der Bibliothek eine `Backend`-Klasse implementiert werden, welche das aktuelle Firm-Programm auf die Zielplattform umsetzt. Die Standardimplementierung `DefaultBackend` verwendet dabei das in libFirm implementierte Backend um den Assemblercode der Zielplattform zu erzeugen.

4 Implementierung

4.1 Typ-Übersetzung

Die Übersetzung der LLVM-Typen bildet die Basis für jegliche weitere Konstruktionen. Aus diesem Grunde werden die hierbei zu berücksichtigenden Eigenarten zuerst vorgestellt.

Dabei werden sämtliche LLVM-Typen durch den `TypeBuilder` nach Firm übertragen. Als Schlüssel für die entsprechende Hashtabelle wird der Zeiger auf den LLVM-Typ verwendet, da dieser in der Speicherdarstellung von LLVM für identische Typen garantiert stets identisch ist.

Primitive Typen

Die Übersetzung primitiver Typen ist weitestgehend trivial, indem ein primitiver Firm-Typ mit dem jeweils entsprechendem Modi erstellt wird. Im Falle von Integer-Typen mit Bit-Breiten, die das Firm-Backend nicht direkt unterstützt, wird auf den jeweils nächstgrößeren verfügbaren Typ zurückgegriffen oder eine Struktur erstellt. Details hierzu sind dem Abschnitt „Nicht-native Integer-Typen“ zu entnehmen.

Zu beachten ist weiterhin, dass Firm vorzeichenlose und -behaftete Integer-Typen und -Modi unterscheidet, während LLVM diese Unterscheidung auf unterschiedliche Instruktionen abwälzt. Beispielsweise verwendet LLVM zwei verschiedene Divisions-Instruktionen `udiv` und `sdiv`, je nachdem, ob eine vorzeichenlose oder -behaftete Division durchgeführt werden soll. Firm verwendet lediglich einen `Div`-Knoten und selektiert die zu verwendende Instruktion anhand der Modi. Aus diesem Grunde kann die Vorzeichenhaftigkeit nicht aus dem LLVM-Typ abgeleitet werden, weshalb LTF per Konvention stets vorzeichenbehaftete Typen verwendet. Gegebenenfalls ist dann eine explizite Konvertierung per `Conv`-Knoten erforderlich, um die jeweils gewünschte Befehlsauswahl zu erzwingen.

Von den erweiterten Floating-Point-Typen wird lediglich `x86_fp80` übersetzt, da die anderen erweiterten Floating-Point-Typen von Firm ohnehin nicht unterstützt werden, während das x86-Backend Modus **E** als 80 Bit Floating-Point-Zahl umsetzt.

Zusammengesetzte Typen

Die Übersetzung von Array-Typen ist relativ problemlos möglich, da LLVM nur eindimensionale Arrays verwendet. Der Element-Typ in Firm kann durch einen rekursiven Aufruf der `retrieve`-Methode aus dem Element-Typ in LLVM abgerufen werden. Das gleiche Verfahren findet auch bei anderen zusammengesetzten Typen Anwendung.

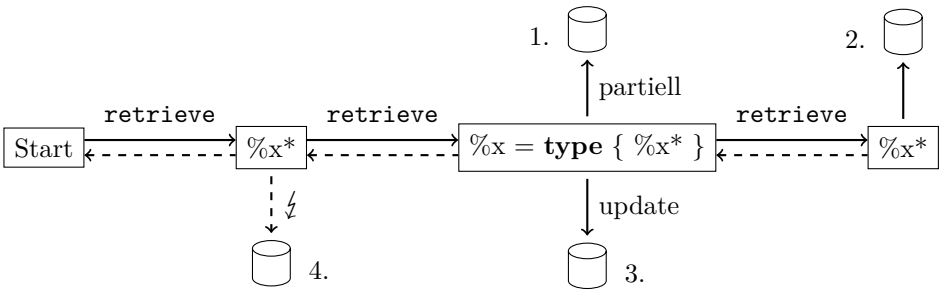


Abbildung 4.1: Konstruktion rekursiver Strukturen

Bei Strukturen muss in Firm zusätzlich ein Layout festgelegt werden, welches in LLVM direkt nicht zur Verfügung steht. Um das gegebenenfalls im Zwischencode angegebene Daten-Layout zu berücksichtigen, werden die `TargetData` und `StructLayout`-Klassen aus den LLVM-Bibliotheken verwendet, welche Offset und Alignment aller Felder in der Struktur zur Verfügung stellen. Für gepackte Strukturen wird ein Alignment von einem Byte verwendet.

Rekursive Datenstrukturen

Im Zusammenhang mit Zeigern lassen sich rekursive Datenstrukturen aufbauen, indem z. B. eine Struktur einen Zeiger auf sich selbst enthält. Dies ist beispielsweise bei der LLVM-Deklaration %x = **type** { %x* } der Fall. In diesem Zusammenhang benötigt man zur Konstruktion der Struktur bereits den Zeiger-Typ, dessen Konstruktion seinerseits den Struktur-Typ voraussetzt. Da das Layout der Struktur und die enthaltenen Entities bei der Konstruktion des Zeiger-Typs in libFirm noch nicht feststehen müssen, kann das Problem gelöst werden, indem die Struktur bereits vor der Konstruktion der enthaltenen Felder in den Cache aufgenommen wird. Der Abruf des Struktur-Typs während der (rekursiven) Konstruktion des Zeigers mittels `retrieve`-Methode gibt somit die bis zu diesem Punkt nur partiell konstruierte Struktur zurück, so dass zunächst der Zeiger konstruiert und zur partiellen Struktur hinzugefügt werden kann. Dieses Vorgehen findet bei allen zusammengesetzten Typen Anwendung, bei denen libFirm eine partielle Konstruktion ermöglicht.

Ein weiteres Problem ist zu beachten, wenn die Konstruktion nicht von der rekursiven Struktur selbst ausgeht, sondern von einem weiteren – außen stehenden – Zeiger auf die Struktur (siehe Abbildung 4.1). Sollte sich die Struktur noch nicht im Cache des `TypeBuilder` befinden, wird zunächst ihre Konstruktion angestoßen und aufgrund des rekursiven Aufbaus dabei genau jener Zeiger-Typ per `retrieve` erneut abgerufen, mit dem die Konstruktion ursprünglich begonnen hatte. Da Zeiger in libFirm nicht partiell konstruiert werden können, befindet sich noch kein Eintrag im Cache, so dass ein neuer Zeiger-Typ erstellt wird, was nun aufgrund des im Cache vorhandenen partiellen Struktur-

Typs möglich ist. Das Resultat ist, dass der ursprüngliche Zeiger-Typ alleine durch die Konstruktion des jeweiligen Basis-Typen bereits rekursiv erstellt wurde und nun erneut erstellt würde. Aus diesem Grund muss vor der eigentlichen Konstruktion des Zeiger-Typen immer erst geprüft werden, ob jener nicht durch die rekursive Konstruktion bereits im Cache abgelegt wurde.

In diesem Zusammenhang sollte noch erwähnt werden, dass der `opaque`-Datentyp von LLVM in Firm durch eine leere Struktur abgebildet werden kann. Er findet dann Verwendung, wenn auf eine Datenstruktur verwiesen wird, die außerhalb der aktuellen Übersetzungseinheit definiert wurde (in C z. B. durch eine „Forward Declaration“ möglich). Dies ist ohnehin nur für Zeiger von praktischer Relevanz, da das Layout der Datenstruktur bei der Verwendung unbekannt ist.

Funktions-Typen

Die Übersetzung der Funktions-Typen ruft einige Probleme hervor, da der Firm-Typ nicht eindeutig aus dem LLVM-Typ abgeleitet werden kann. Beispielsweise besteht in LLVM die Möglichkeit, Parameter mit zusätzlichen Attributen zu versehen, welche den resultierenden Firm-Typen beeinflussen können. Allerdings sind diese Attribute nicht dem LLVM-Typ zugeordnet, sondern der jeweiligen Funktions-Deklaration und gegebenenfalls den `call`-Instruktionen, welche die Funktion aufrufen. Ein Beispiel hierfür wäre das `byval`-Attribut, welches – angewendet auf einen Zeiger-Parameter – erzwingt, dass der zugrundeliegende Wert kopiert werden soll und ein Zeiger auf die Kopie übergeben wird. Dies kann in Firm umgesetzt werden, indem anstatt des Zeiger-Typen direkt der Typ des zugrundeliegenden Wertes – beispielsweise einer Struktur – zum Einsatz kommt.

Dieser Umstand ist auch relevant bei der Übersetzung von Funktions-Typen mit variabler Parameter-Anzahl. Bei der Deklaration wird dabei zunächst ein Funktions-Typ erstellt, der lediglich statische Parameter enthält, aber gesondert gekennzeichnet wird. Der eigentliche Funktions-Typ kann dabei erst bei einem Aufruf mittels `call`-Instruktion – anhand der verwendeten Parameter – festgelegt werden. Da LLVM hier stets denselben Typ verwendet, Firm jedoch die zusätzlichen Parameter explizit im Typ benennen muss, kann hinter einem Funktions-Typ in LLVM wieder eine ganze Reihe von potentiellen Firm-Typen stehen.

Im Endeffekt bedeutet dies, dass zusätzlich zum LLVM-Typen noch weitere Daten hinzugezogen werden müssen, um einen entsprechenden Firm-Typen zu erstellen. Um dies umzusetzen, bietet der `TypeBuilder` eine zusätzliche `retrieveFunction`-Methode, welcher entsprechende Informationen, beispielsweise von Seiten der `call`-Instruktion, übergeben werden können. Um weiterhin die Funktions-Typen im Cache bereithalten zu können, muss der Schlüssel um diese zusätzlichen Daten erweitert werden. Dies geschieht, indem die erforderlichen Informationen zu einem sogenannten „Meta-Schlüssel“ zusammengefasst werden und an den eigentlichen Schlüssel angehängt werden. Dies ist dank der Erweiterungsmechanismen der Boost.Unordered-Bibliothek vergleichsweise einfach zu bewerkstelligen.

Nicht-native Integer-Typen

Ein weiteres Problem stellen die zusätzlichen Integer-Typen dar, die nicht direkt in Firm umgesetzt werden können. Diese treten vor allem im Zusammenhang mit Bit-Feldern in C auf, können durchaus aber auch bei der Initialisierung kleiner Strukturtypen Anwendung finden, indem diese mit einem konstanten Integer passender Breite überschrieben werden. Unterschieden wird in LTF zunächst zwischen Integer-Typen, die kleiner als 64 Bit sind und somit in einem der von Firm verwendeten Integer-Typen eingebettet werden können und solchen, die diese Grenze überschreiten.

Modulo-Typen

Nicht-native Integer mit einer Breite von $n < 64$ Bit müssen eine Arithmetik modulo 2^n umsetzen und werden daher im Folgenden als Modulo-Typen bezeichnet. Um sie in Firm abzubilden, wird zunächst der nächstgrößte Firm-Typ mit $m > n$ Bit Größe verwendet. Per Konvention wird dabei angenommen, dass die oberen (nicht verwendeten) $m - n$ Bits undefiniert sind und damit beliebige Werte enthalten können. Dies hat den Vorteil, dass diese nicht ständig neu gesetzt werden müssen, um einer Definition zu genügen. Außerdem lassen sich so viele arithmetische Operationen ohne Anpassungen auf Modulo-Typen erweitern. Beispielsweise ist die Addition automatisch korrekt auf Modulo-Typen übertragbar, da sich der Übertrag stets nur auf die jeweils höheren Bits auswirkt, so dass die undefinierten oberen Bits keine Auswirkung auf die unteren Bits haben. Mathematisch betrachtet stellen die oberen Bits ein Vielfaches von 2^n dar, so dass sich bei Addition zweier Zahlen a und b der Breite n mit undefinierten Bits Folgendes ergibt:

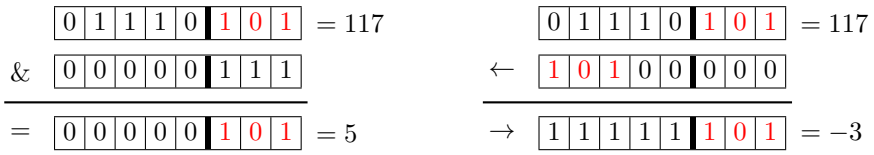
$$(a + 2^n r) + (b + 2^n s) = a + b + 2^n(r + s) \equiv a + b \pmod{2^n} \tag{4.1}$$

Dieselbe Argumentation lässt sich z. B. auch auf die Multiplikation übertragen:

$$(a + 2^n r) \cdot (b + 2^n s) = ab + 2^n(as + br) + 2^{2n} rs \equiv ab \pmod{2^n} \tag{4.2}$$

Natürlich tritt früher oder später ein Fall auf, in dem die oberen Bits tatsächlich definiert werden müssen. Dies ist beispielsweise bei einer Division der Fall oder wenn der Wert in einem Vergleich verwendet werden soll oder zu einem anderen Typ konvertiert werden muss. Wenn eine solche Situation auftritt, wird der Wert zunächst durch ein Setzen der oberen Bits normalisiert, wobei zu unterscheiden ist, ob der Typ vorzeichenlos oder -behaftet interpretiert werden soll.

Diese Normalisierung wird im Konvertierungs-Framework von LTF umgesetzt und maskiert den Wert im Falle einer vorzeichenlosen Erweiterung („Zero-Normalization“) durch einen `And`-Knoten, so dass die jeweils undefinierten oberen Bits auf Null gesetzt werden (siehe Abbildung 4.2). Bei der vorzeichenbehafteten Erweiterung („Sign-Normalization“) muss das Vorzeichenbit, also in diesem Fall das n-te Bit auf alle oberen Bits übertragen werden. Dies wird durch eine Abfolge von Linksshift und arithmetischen Rechtsshift



(a) Zero-Normalization (b) Sign-Normalization

Abbildung 4.2: Normalisierung von Modulo-Typen

umgesetzt. Dabei schiebt der Linksshift das Vorzeichenbit soweit nach oben, bis es zum Vorzeichenbit des tatsächlichen Integer-Typs wird, wobei die undefinierten Bits vollständig aus dem Integer hinausgeschoben werden. Der arithmetische Rechtsshift um die gleiche Distanz stellt den Wert wieder her und überträgt das Vorzeichenbit auf alle oberen Bits, wodurch diese nun eindeutig definiert sind.

Unter Beachtung dieser Umstände ist es möglich, die entsprechenden LLVM-Typen in Firm zu emulieren. Zu beachten ist insbesondere, dass der **i1**-Typ in LLVM, welcher dem booleschen Wert in Firm entspricht auch davon betroffen ist und somit auch für arithmetische Operationen zur Verfügung steht. Dies stellt aber nur eine Möglichkeit dar, **i1** in Firm darzustellen, da Firm boolesche Werte für gewöhnlich als Modus **b** darstellt. Genaueres zur Behandlung boolescher Werte lässt sich dem Abschnitt 4.4 über das Konvertierungs-Framework entnehmen.

BigInt-Typen

Integer mit mehr als 64 Bit werden in LTF als „BigInt-Typen“ bezeichnet. Da Firm nativ mit solchen Integer-Typen nicht umgehen kann und auch keinen Modus zur Verfügung stellt, der dies bewerkstelligen könnte, müssen diese in kleinere Typen heruntergebrochen werden. Aus diesem Grunde erstellt der `TypeBuilder` keinen primitiven Typ mehr, sondern eine Struktur mit mehreren 32-Bit-Integern, welche in der Lage ist, die entsprechenden Datenwerte zu fassen.

Abgesehen vom Laden und Speichern, sowie der Umsetzung von BigInt-Konstanten, die ein genaues Vielfaches von 32 Bit darstellen, unterstützt LTF zur Zeit keine BigInt-Typen. Dies reicht allerdings meist aus, da die LLVM-Frontends für gewöhnlich nur auf BigInt-Typen zurückgreifen, um z. B. Strukturen durch das Überschreiben mit einer BigInt-Konstante zu initialisieren. Die Handhabung der BigInt-Typen im Funktionsgraphen selber entspricht ansonsten der Handhabung von Aggregat-Typen, weshalb in diesem Zusammenhang auf den Abschnitt „Zusammengesetzte Typen“ in Kapitel 4.5 verwiesen sei.

4.2 Globale Variablen und Funktionen

Die Konstruktion der erforderlichen Entities im Global-Type des Firm-Programmes findet im `EntityBuilder` statt. Erforderliche Typen werden dabei gegebenenfalls vom `TypeBuilder` abgerufen. Zur Konstruktion von Initialisierungswerten wird außerdem ein `NodeBuilder` hinzugezogen.

Linkage-Typen

LLVM unterstützt 14 verschiedene Linkage-Typen, die den jeweiligen Variablen bzw. Funktionen zugeordnet werden können und steuern, welche Anweisungen dem Linker über die ausgegebenen Assembler-Direktiven für das jeweilige Symbol übergeben werden. Da libFirm bisher nur auf C und Java beschränkt war, mussten zunächst einige Anpassungen durchgeführt werden, damit das Backend die entsprechenden Direktiven erzeugen kann. Dies betrifft vor allem die Linkage-Typen `weak` und `linkonce`.

Dabei wird der Linker dazu veranlasst, die als `weak` oder `linkonce` markierten Symbole zu überschreiben, wenn er gleichnamige Symbole findet, die nicht mit `weak` bzw. `linkonce` gekennzeichnet wurden. Beispielsweise erlauben es einige C-Compiler, Symbole explizit als `weak` zu markieren, damit man sie in einer anderen Übersetzungseinheit überschreiben kann. Bei `linkonce` kommt derselbe Mechanismus zum Einsatz, allerdings darf der Linker nicht verwendete Symbole verwerfen. Dies wird beim Instanziieren von Templates und Inline-Funktionen in C++ verwendet, die über eine Header-Datei in mehreren Quellcodedateien eingebunden werden und damit auch in mehreren Übersetzungseinheiten Maschinencode ausgeben. Der Linker kann dann die überflüssigen Definitionen verwerfen.

Genau genommen darf ein Inlining aber erst dann stattfinden, wenn die Symbole vom Frontend mit `weak_odr`- bzw. `linkonce_odr` markiert wurden. Die Benennung orientiert sich dabei an der One-Definition-Rule aus dem C++-Standard, durch die sichergestellt wird, dass alle Definitionen eines Symbols im Programm identisch sind. Würde diese Prämisse nicht gelten, dann könnte der Compiler den falschen Code inlinen, denn der tatsächlich aufzurufende Programmcode kann erst beim Linken abgeleitet werden, wenn alle Symbole bekannt sind. Die aktuelle Implementation von `weak` in libFirm erlaubt jedoch ohnehin noch kein Inlining.

Initialisierungswerte

In LLVM können Variablen und Konstanten mit beliebigen konstanten Ausdrücken initialisiert werden. Dabei kommen dieselben LLVM-Konstrukte zum Einsatz, wie bei der Darstellung konstanter Ausdrücke in den Befehlslisten. Auch Firm verwendet in beiden Fällen Firm-Knoten, so dass es sich anbietet, die `NodeBuilder`-Klasse auch in diesem Kontext zu verwenden.

Dies wird umgesetzt, indem der `EntityBuilder` eine eigene Instanz der `NodeBuilder`-Klasse verwendet. Diese verwendet allerdings keinen Funktions-Graphen als Basis, sondern

den speziell dafür vorgesehenen „Const-Graphen“. Der wesentliche Unterschied besteht darin, dass bei der Konstruktion auf dem Const-Graphen der Funktionsumfang eingeschränkt wird, so dass keine Steuerfluss-Konstrukte oder Speicherzugriffe möglich sind. Ansonsten erhält der `NodeBuilder` – wie auch bei der Konstruktion der Funktions-Graphen – einen LLVM-Wert und erzeugt daraus entsprechende Knoten, die als Initialisierer bei den entsprechenden Entities hinterlegt werden können.

Eine gesonderte Behandlung muss allerdings für die zusammengesetzten Strukturen erfolgen, da Firm-Knoten keine Strukturen oder Arrays produzieren können, sondern nur solche Werte, zu denen ein Modus existiert. Es ist aber leicht möglich, diesen Fall abzufangen und für jedes Feld einzeln einen Initialisierer anzugeben. Dieses Vorgehen muss im Falle von verschachtelten Strukturen gegebenenfalls rekursiv angewendet werden.

Durch die Initialisierer kann es aber auch zu einem weiteren Spezialfall kommen, denn es ist möglich, eine andere globale Variable zu referenzieren. Aus diesem Grunde ist der `EntityBuilder` grundsätzlich auch in der Lage, Variablen bei Bedarf zu konvertieren. Wenn also eine noch nicht konstruierte Variable rekursiv angefordert wird, so wird diese direkt konvertiert und später übersprungen. Ebenso wie bei den rekursiven Typen kann es hierbei zu Zyklen kommen, beispielsweise bei zwei Zeigern, die sich gegenseitig referenzieren. Auch hier lässt sich dabei dieselbe Lösung verwenden: das Entity wird bereits vor der Konstruktion des Initialisierungswertes als partiell konstruiertes Entity im Cache abgelegt, so dass eine weitere Rekursion verhindert wird.

Intrinsic-Variablen

LLVM bildet einige Konstrukte mittels spezieller „Intrinsic-Variablen“ ab. Beispielsweise werden Konstruktur- und Destruktor-Funktionen umgesetzt, indem Zeiger auf die jeweiligen Funktionen in den speziellen Arrays `@llvm.global_ctor` und `@llvm.global_dtor` abgelegt werden. Firm verfolgt hier einen etwas anderen Ansatz, bei dem den Entities ein Segment zugeordnet wird und für Konstruktoren und Destruktoren zwei reservierte Segmente zur Verfügung stehen, in denen sich Funktionszeiger ablegen lassen. Entsprechend muss eine gesonderte Behandlung erfolgen, sobald eine Variable mit entsprechendem Namen übersetzt wird.

Praktisch bedeutet dies, dass die Zeiger dabei aus dem LLVM-Array extrahiert werden und für jeden Eintrag ein Entity im jeweiligen Segment erzeugt wird und mit einem Zeiger auf die jeweilige Funktion initialisiert wird. Dabei ist zu beachten, dass die Funktionsdeklarationen vor den Variablen übersetzt werden, so dass die entsprechenden Zeiger an dieser Stelle konstruiert werden können.

4.3 Funktions-Aufbau

Der Aufbau eines Funktionsgraphen ist im `GraphBuilder` in mehrere Stufen unterteilt, die im Folgenden kurz dargestellt werden.

Konstruktion der Blöcke

In einem ersten Schritt werden die von LLVM vorgesehenen Blöcke, wie in Kapitel 3.1 dargelegt, in Konstruktionsreihenfolge gebracht, indem sie ausgehend vom Startblock mittels Tiefensuche erkundet werden. Dies hat den zusätzlichen Nebeneffekt, dass unerreichbare Blöcke bereits hier erkannt und ausgeschlossen werden. Unmittelbar danach werden für die (übrigen) Blöcke mit Hilfe des `BlockBuilder` entsprechende Firm-Blöcke konstruiert, damit später auf diese zurückgegriffen werden kann. Zu beachten ist dabei, dass die Steuerflussabhängigkeiten erst später aufgebaut werden können, da zu diesem Zeitpunkt noch keine Steuerflussknoten existieren.

Konstruktion des Frame-Typs

Im Anschluss daran wird der Frame-Typ der Funktion konstruiert. Zwar unterstützt Firm mit dem `Alloc`-Knoten dasselbe Konstrukt wie LLVM mit den `alloca`-Instruktionen, aber die Verwendung des Frame-Typs erlaubt weiterreichende Optimierungen. Da LLVM statische Allokationen üblicherweise auf den Startblock verschiebt und dort gruppiert, ist es in den meisten Fällen ausreichend, nur solche Allokationen für den Frame-Typ zu berücksichtigen. Davon abgesehen sind Allokationen im Startblock in jedem Fall statisch, denn der Startblock darf per Konvention in LLVM nie als Sprungziel verwendet werden und wird in jedem Fall ausgeführt.

Praktisch bedeutet dies, dass über den Startblock iteriert wird und für jede gefundene `alloca`-Instruktion der `FrameBuilder` aufgerufen wird, welcher ein entsprechendes Entity auf dem Frame-Typ einrichtet. Beim Zugriff auf eine `alloca`-Instruktion kann später dann der `NodeBuilder` das entsprechende Entity vom `FrameBuilder` erfragen und – soweit vorhanden – direkt darauf zugreifen. In allen Fällen, in denen dies nicht möglich ist (z. B. weil die Allokation nicht im Startblock steht), wird auf den `Alloca`-Knoten zurückgegriffen.

Konstruktion des Graphen

Die eigentliche Konstruktion des Funktionsgraphen wird vom `NodeBuilder` durchgeführt. Dabei werden im `GraphBuilder` die Blöcke in der zuvor festgelegten Reihenfolge durchlaufen und alle darin vorkommenden Instruktionen linear an den `NodeBuilder` übergeben. Dieser hat Zugriff auf alle lokalen und globalen `Builder` und kann somit die bei der Konstruktion anfallenden Verweise auflösen.

Gemäß dem in Kapitel 3.1 vorgestellten Verfahren werden dabei zunächst anstatt der `Phi`-Knoten Platzhalter-Knoten verwendet. Daraufhin erfolgt eine zweite Iteration aller `phi`-Instruktionen in derselben Reihenfolge, wobei diesmal die spezielle `buildPhi`-Methode des `NodeBuilder` verwendet wird. Diese ersetzt die vorhandenen Platzhalter-Knoten durch die eigentlich erforderlichen `Phi`-Knoten mit den entsprechenden Abhängigkeitskanten. Näheres dazu ist Kapitel 4.5 zu entnehmen.

Finalisierung der Blöcke

Die verwendeten Blöcke werden in libFirm als sogenannte „Immature Blocks“ erstellt. Dies hat zum einen den Vorteil, dass die Vorgänger-Knoten bei der Konstruktion des Blockes nicht bekannt sein müssen und zum anderen ermöglicht es das automatische Erstellen der Speicherkanten, inklusive hierfür erforderlicher `Phi`-Knoten. Um dies zu gewährleisten, verwaltet libFirm intern weitere Datenstrukturen zu jedem Block, die erst dann freigegeben werden können, wenn die Konstruktion des Blockes abgeschlossen ist und sämtliche Vorgänger-Knoten (d. h. Sprünge auf den Block) feststehen.

Damit eine solche Finalisierung der Blöcke frühestmöglich geschehen kann, verwaltet der `BlockBuilder` einen Referenzzähler für jeden Block, der auf $1 + n$ initialisiert wird, wobei n die aus LLVM bekannte Anzahl der Vorgänger-Knoten bezeichnet. Sobald der `NodeBuilder` einen Sprung auf einen Knoten fertig stellt, benachrichtigt er den `BlockBuilder`, welcher daraufhin den Referenzzähler verringert. Ebenso verringert der `GraphBuilder` den Zähler eines Blockes, wenn alle Knoten im Block konstruiert wurden. Sobald der Referenzzähler 0 erreicht, kann der Block finalisiert werden.

Behandlung von Endlosschleifen

Da eine Funktion aus einer Endlosschleife heraus nie verlassen wird, kann es vorkommen, dass die dabei involvierten Blöcke keine Verbindung zum Endknoten des Graphen besitzen. Da die Firm-Graphen aber Abhängigkeitskanten verwenden, führt dies dazu, dass die jeweiligen Blöcke vom `End`-Knoten auch nicht erreicht werden können und von der Steuerfluss-Optimierung als „toter Code“ entfernt werden. Um dies zu verhindern, ist es möglich, spezielle „Keep-Alive“-Kanten in den Graphen einzubauen, welche künstliche Verbindungen zwischen `End`-Knoten und den jeweiligen Blöcken herstellen.

Ein entsprechendes Vorgehen wird in LTF vom `BlockBuilder` implementiert und findet unmittelbar vor dem Abschluss der Graph-Konstruktion Anwendung. Dabei werden zunächst aus der Menge der bereits bekannten Blöcke alle jene entfernt, die beim Ablaufen des Graphen vom `End`-Knoten aus gefunden werden. Die Menge M der verbleibenden Blöcke soll nun am Leben erhalten werden, indem Keep-Alive-Kanten zu einer hinreichenden Menge von Wurzel-Blöcken aufgebaut werden, von denen alle weiteren Blöcke erreichbar sind. Um diese zu isolieren, werden die verbleibenden Blöcke einzeln betrachtet und jeweils alle transitiv von dort erreichbaren Blöcke aus M entfernt. Am Ende verbleibt eine minimale Menge von Wurzel-Blöcken, die sich nicht gegenseitig erreichen können. (Im Falle eines Zyklus verbleibt der erste betrachtete Block des Zyklus, da die anderen Blöcke von ihm aus erreicht wurden.)

Um die verbleibenden Blöcke am Leben zu erhalten, muss nun jeweils eine Keep-Alive-Kante zur letzten Instruktion erzeugt werden, die einen Speicher-Wert (Modus **M**) erzeugt und damit Nebeneffekte hervorruft. Zu diesem Zweck benachrichtigt der `NodeBuilder` den `BlockBuilder` beim Konstruieren der terminierenden Instruktion am Ende jedes Blockes (der Sprung muss vom `BlockBuilder` ohnehin erfasst werden). Hierbei wird

dann der entsprechende Speicher-Wert hinterlegt, damit er später verwendet werden kann, um die Knoten im Block am Leben zu erhalten.

4.4 Konvertierungs-Framework

Bevor der `NodeBuilder` und die Konvertierung der Funktionslisten betrachtet werden, soll ein Blick darauf geworfen werden, wie LTF mit den Modi im Funktions-Graphen umgeht und wie gegebenenfalls Typ-Konvertierungen umgesetzt werden.

Ansatz

Zwar sind die verwendeten LLVM-Typen dank der Verifikationsphase stets kompatibel zueinander, da einem LLVM-Typen aber oftmals mehrere potentielle Firm-Modi entgegenstehen, ist die eindeutige Festlegung des jeweiligen Modus vom Kontext abhängig und kann zu Konflikten führen. Beispielsweise kann der LLVM-Typ **i1** in Firm als boolescher Wert mit Modus **b** oder als Integer mit Modus **Bs** oder **Bu** (je nachdem, ob die Interpretation des Vorzeichens erwünscht ist) verwendet werden.

Aus diesem Grunde gilt generell folgende Konvention: das Ergebnis eines Firm-Knoten, welcher vom `NodeBuilder` konstruiert wurde, kann einen beliebigen Modus aufweisen, der zu dem jeweils produzierten LLVM-Typ kompatibel ist. Beispielsweise erzeugt das Laden eines **i1**-Wertes in LTF einen Wert vom Modus **Bs**, während ein Vergleich einen Wert vom Modus **b** erzeugt. Die Konvention vermeidet die ansonsten notwendigen ständigen Konvertierungen zur Erhaltung des jeweiligen Modus. Wäre **i1** z. B. per Konvention Modus **b**, so müsste nach jeder arithmetischen Operation erneut eine Konvertierung nach Modus **b** erfolgen, was insbesondere bei Verkettung mehrerer arithmetischer Operationen unnötigen Aufwand bedeutet. Dies gilt in gleichem Maße für Modus **Bs**, nur dass das Problem dann bei den logischen Operatoren auftritt. Indem man die Konvertierung so bis zum Zeitpunkt der tatsächlichen Verwendung hinauszögert, vermeidet man das Problem weitestgehend.

Damit dies ohne weitere Umstände möglich ist, benötigt man eine generalisierte Typ-Konvertierung, welche die dabei notwendigen Spezialfälle zu berücksichtigen weiß. Dies kann durch den `Conv`-Knoten von Firm nicht immer gewährleistet werden. Beispielsweise erfordert die Konvertierung nicht-nativer Integer-Typen häufig auch eine Normalisierung der undefinierten Bits. Diese entstehende Lücke füllt das Konvertierungs-Framework, welches entsprechende Typ-Konvertierungen in Form von Firm-Knoten aufbauen kann.

Dabei wird ein Bottom-Up-Ansatz verfolgt, der ausgehend von speziellen Konvertierungen generalisierte Konvertierungen bereitstellt und sich ansonsten an den in LLVM verwendeten Aufbau mit mehreren Konvertierungs-Funktionen anlehnt (siehe [LA10]), da dies einerseits die Behandlung etwaiger Sonderfälle vereinfacht und andererseits genau definiert ist (und mittels Assertions sichergestellt werden kann), welche Eingabetypen jeweils zu erwarten sind.

Bool-Cast

Elementare Konvertierungen von oder zu Modus **b** werden unter dem Oberbegriff „Bool-Cast“ zusammengefasst. Dabei wird üblicherweise ein `Mux`-Knoten konstruiert, wenn die Konvertierung vom Modus **b** ausgeht. Dieser wählt – ähnlich dem ternären `?:`-Operator in C/C++ – ausgehend vom Eingabewert einen von zwei möglichen Rückgabewerten aus. Zu beachten ist dabei, dass die vorzeichenbehafteten Konvertierungen das gesetzte Bit als -1 interpretieren. Somit erzeugt beispielsweise die vorzeichenbehaftete Erweiterung mittels `bc::SExt` direkt die normalisierten Werte -1 oder 0, je nachdem, ob der ursprüngliche Wert `true` oder `false` ist.

Wird dagegen nach Modus **b** konvertiert, findet meist ein Vergleich statt. Das Reduzieren eines Integers mittels `bc::trunc` maskiert beispielsweise das unterste Bit und vergleicht es mit 0, während die vorzeichenbehaftete Konvertierung von Floating-Point-Typen durch `bc::FPToSI` im Wesentlichen prüft, ob der Wert -1 ist (dies reicht aus, da Werte außerhalb des Intervalls $[-1, 0]$ ohnehin undefiniert sind).

Firm-Cast

Unter dem Begriff „Firm-Cast“ werden alle Konvertierungen zusammengefasst, die Firm unmittelbar unterstützt. Zu beachten ist, dass einige der LLVM-Konvertierungen dennoch mit mehreren `Conv`-Knoten in Firm umgesetzt werden müssen. Beispielsweise müssen die vorzeichenlosen bzw. -behafteten Erweiterungen `fc::ZExt` und `fc::SExt` zunächst sicherstellen, dass der Eingabe-Wert einen entsprechenden Modus aufweist, so dass Firm die korrekte Konvertierung erzeugt. Dies gilt ebenso für die Konvertierungen `fc::UIToFP` und `fc::SIToFP` zu den Floating-Point-Typen (und deren Umkehrung), da das Vorzeichenbit hierbei entweder ignoriert oder interpretiert werden muss.

Modulo-Cast

Dies beinhaltet vor allem die Normalisierung der undefinierten Bits bei nicht-nativen Integer-Typen (siehe Kapitel 4.1, Abschnitt „Modulo-Typen“) in Form der beiden Funktionen `mc::ZNorm` und `mc::SNorm`. Dabei konstruiert `mc::ZNorm` eine Normalisierung der undefinierten Bits auf 0, während `mc::SNorm` das Vorzeichenbit zum Normalisieren verwendet.

Any-Cast

Die Konvertierungen aus den letzten Abschnitten werden nun in Form von generalisierten Konvertierungen als „Any-Cast“ zusammengefasst. Dies bedeutet, dass `ac::SExt` beispielsweise problemlos sowohl boolesche Werte im Modus **b** als auch Integer-Typen beliebiger Breite (inklusive nicht-nativer Typen) erweitern kann. Dies erfordert je nach Ursprungs-Modus einen Bool-Cast, einen Firm-Cast oder auch eine Kombination von

Modulo- und Firm-Cast, denn eine vorzeichenbehaftete Erweiterung erfordert z. B. zunächst das Normalisieren der undefinierten Bits auf das Vorzeichenbit.

Auf diese Weise werden alle LLVM-Konvertierungen als Any-Cast für sämtliche kompatiblen Modi zur Verfügung gestellt. Darüber hinaus existieren einige zusätzliche Konvertierungen, beispielsweise um einen beliebigen Modus, der zu **i1** kompatibel ist, in den gewünschten Modus zu konvertieren oder einen Integer in den jeweiligen vorzeichenlosen oder -behafteten Modus zu überführen.

4.5 Instruktions-Übersetzung

Der `NodeBuilder` bildet das Kernstück von LTF und überträgt LLVM-Instruktionen auf äquivalente Firm-Knoten. Dazu stehen ihm sämtliche globale Variablen und Funktionen, sowie die Blockstruktur des Graphen und dessen Frame-Typ über die diversen `Builder` zur Verfügung.

Konvertierungen

Aufgrund des bereits vorgestellten Konvertierungs-Frameworkes stehen für die von LLVM angebotenen Konvertierungen bereits passende Konstruktions-Funktionen zur Verfügung, die direkt zur Umsetzung der Instruktionen verwendet werden können. Eine Ausnahme stellt die `bitcast`-Instruktion dar, die in vergleichbarer Form in Firm nicht existiert. Die hierbei stattfindende Reinterpretation der Bitfolge lässt sich allerdings nachbilden, indem der Wert im Speicher zwischengespeichert und mit dem Zielmodus von der entsprechenden Adresse zurückgeladen wird. Der erforderliche Speicherplatz wird mittels `Alloc`-Knoten reserviert und nach dem Bitcast per `Free`-Knoten freigegeben.

Zu beachten ist dabei, dass die Änderung eines Zeiger-Typen in LLVM ebenfalls mittels Bitcast umgesetzt wird, während Firm im Graphen hier überhaupt nicht mehr zwischen verschiedenen Zeiger-Typen unterscheidet, sondern stets Modus **P** verwendet. In diesem Fall wird daher der Bitcast in Firm verworfen und durch den ursprünglichen Wert ersetzt.

Da für den Bitcast Speicherzugriffe verwendet werden, kann er zur Zeit – mit Ausnahme von Zeigern – nicht zur Übersetzung konstanter Ausdrücke auf dem Const-Graph verwendet werden. Die Initialisierung einer Variablen mittels Bitcast ist somit noch nicht möglich, wenn hierfür ein Speicherzugriff notwendig ist. Die in Firm über Strukturen nachgebildeten BigInt-Typen (siehe Kapitel 4.1, Abschnitt „BigInt-Typen“) sind ebenfalls vom Bitcast ausgeschlossen, so dass z. B. kein Bitcast von `x86_fp80` auf `i80` möglich ist.

Konstante Ausdrücke

Konstante Ausdrücke lassen sich nicht gänzlich in das bestehende Schema einordnen, da sie – im Gegensatz zu Instruktionen – nicht als gesonderte Einheiten in den Befehlslisten auftreten, sondern direkt als Parameter verwendet werden können. Dennoch werden

sie auf getrennte Firm-Knoten abgebildet. Dies bedeutet, dass sie beim Abruf vom `NodeBuilder` bei Bedarf (rekursiv) konstruiert werden müssen.

Zu beachten ist weiterhin, dass verschiedene Vorkommen desselben konstanten Ausdruckes in der Speicherdarstellung von LLVM in der Regel auf denselben Wert zeigen. Würde man die durch den konstanten Ausdruck konstruierten Knoten im lokalen Block erzeugen, so könnte dies bei erneuter Verwendung in einem anderen Block die SSA-Eigenschaften verletzen. Aus diesem Grunde werden konstante Ausdrücke generell nur im ersten Block des jeweiligen Graphen erzeugt (welcher dem LLVM-Startblock entspricht), was zwar der üblichen Konstruktionsordnung widerspricht, aber gemeinhin zulässig ist, da hier keine Nebeneffekte auftreten können und somit keine Speicherkanten verwendet werden müssen. Da es nur diesen einen eindeutigen Startblock gibt, dominieren die konstruierten Knoten automatisch alle möglichen Verwendungen – unabhängig vom Block – und die SSA-Eigenschaft bleibt erhalten.

Speicherzugriffe

Speicherzugriffe sind weitestgehend analog von LLVM nach Firm übertragbar, indem verwendete Zeiger und Typen lokal erfragt, bzw. vom `TypeBuilder` abgerufen werden und mit deren Hilfe entsprechende `Load`- bzw. `Store`-Knoten in Firm konstruiert werden. Zu beachten ist dabei, dass zum Laden und Speichern derselbe Modus verwendet wird, was bei der teilweise mehrdeutigen Zuordnung von LLVM-Typ nach Firm-Modus eine bestimmte Konvention erfordert. Aus diesem Grunde ist jedem LLVM-Typ in LTF ein sogenannter Basis-Modus zugeordnet, der hier verwendet wird und durch Hilfsfunktionen vom LLVM-Typ zu erfragen ist. Ebenso bietet das Konvertierungs-Framework entsprechende `[Type]ToBase`-Funktionen an.

Darüber hinaus muss im Falle zusammengesetzter Strukturen eine Serialisierung bzw. Deserialisierung der gesamten Datenstruktur erfolgen, welche später in Abschnitt „Zusammengesetzte Typen“ genauer beschrieben wird.

Adressberechnung

Listing 2: Adressberechnung in LLVM

```
; Entspricht &((ptr+1)->b[2]), mit struct Test { int a; int b[10]; }
%0 = getelementptr %struct.Test* %ptr, i32 1, i32 1, i32 2
```

In LLVM wird die Adressberechnung durch die `getelementptr`-Instruktion umgesetzt (siehe Listing 2). Dabei kann man ausgehend von einem Zeiger rekursiv per Indizes durch Datenstrukturen und Arrays navigieren. Bei Zeiger-Typen setzt der Index eine Zeiger-Arithmetik, ähnlich der in C/C++ um. Details zur `getelementptr`-Instruktion sind der Sprachreferenz [LA10] und [Speb] zu entnehmen.

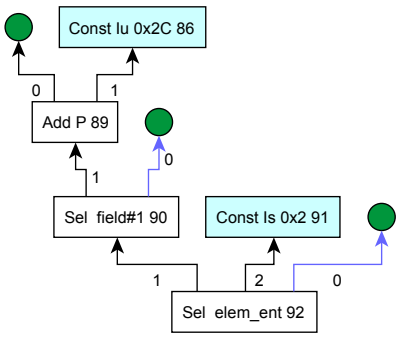


Abbildung 4.3: Adressberechnung in Firm

Um dasselbe Verhalten in Firm umzusetzen, ist eine Kombination von `Sel`-Knoten (für Arrays und Strukturen) und expliziter Pointer-Arithmetik (in Form von `Add`- und `Mul`-Knoten mit Modus **P**) notwendig. Zum Vergleich ist die Selektion aus Listing 2 in Abbildung 4.3 als Firm-Graph dargestellt (wobei der `Mul`-Knoten bereits durch die Konstantenfaltung entfernt wurde).

Zum Aufbau der Firm-Knoten werden dabei rekursiv die Indizes durchlaufen und die jeweils betrachteten Typen bei der Auswahl der Selektionsknoten herangezogen. Bei Array- und Struktur-Typen kann der Firm-Typ aus dem LLVM-Typ abgeleitet und dann zusammen mit dem jeweiligen Index das gewünschte Entity abgerufen werden. Hiermit lässt sich nun der entsprechende `Sel`-Knoten konstruieren. Bei Zeiger-Typen wird mit Hilfe von `Add`- und `Mul`-Knoten jeweils ein Vielfaches der Typ-Größe zum Zeiger addiert. Der so erhaltene neue Zeiger dient als Basis für die Adressberechnung mit dem nächsten Index. Als Typ zur Auswahl der nächsten Selektion wird dann entweder der Typ des selektierten Entities oder der Basis-Typ des Zeigers verwendet.

Sobald alle Indizes umgesetzt wurden, wird die letzte Adresse als Ergebnis der Instruktion zurückgeliefert. Da es zum Zeitpunkt dieser Arbeit in libFirm noch zu Problemen im Zusammenhang mit LLVM-Bitcasts und den `Sel`-Knoten kommen kann, besteht die Möglichkeit, diese bei Arrays und Strukturen durch explizite Zeiger-Arithmetik zu ersetzen, wobei dies weniger Optimierungsmöglichkeiten bietet.

Funktionsaufrufe

Bei Funktionsaufrufen muss zunächst zwischen direkten Funktionsaufrufen und solchen über Funktions-Zeiger unterschieden werden. Dabei ändert sich aber lediglich die Erzeugung der Zeiger in Firm. Während bei direkten Funktionsaufrufen das Entity vom `EntityBuilder` abgerufen wird, ist der Zeiger bei indirekten Aufrufen wie jeder andere Operand lokal im `NodeBuilder` abrufbar.

Ausgehend vom LLVM-Typ der Funktion bzw. des Zeigers kann nun – unter Berücksichtigung etwaiger hierfür relevanter Attribute und zusätzlicher Parameter – ein

entsprechender Firm-Typ für den jeweiligen Aufruf vom `TypeBuilder` abgerufen werden (vgl. Kapitel 4.1, Abschnitt „Funktions-Typen“). Ansonsten ist zu beachten, dass Parameter- und Rückgabewerte stets in ihrem Basis-Modus übergeben werden, also gegebenenfalls noch eine Konvertierung erfordern (siehe Abschnitt „Speicherzugriffe“). Dies ist einerseits nötig, damit der Modus mit dem Parameter-Typ kompatibel ist und andererseits, damit die aufgerufene Funktion den Modus des Parameters ausgehend vom LLVM-Typ eindeutig zuordnen kann. Bei zusammengesetzten Typen sind gegebenenfalls weitere Vorkehrungen notwendig, wobei Genaueres darüber dem Abschnitt „Zusammengesetzte Typen“ zu entnehmen ist.

Ebenfalls zu beachten ist, dass hier auch Aufrufe von Intrinsic-Funktionen berücksichtigt werden müssen, sofern sie nicht schon in der Vorbereitungsphase entfernt wurden. Dies betrifft vor allem die Handhabung variabler Argumentlisten.

Phi-Instruktionen

Wie bereits zuvor dargelegt, werden `phi`-Instruktionen in zwei Schritten übersetzt, wobei zunächst ein Platzhalter-Knoten zum Einsatz kommt. Dabei handelt es sich um einen `Phi`-Knoten ohne Vorgänger mit dem jeweiligen Modus. Es ist anzumerken, dass der jeweils dem LLVM-Typ zugeordnete Basis-Modus verwendet wird, um Mehrdeutigkeiten zu vermeiden (siehe Abschnitt „Speicherzugriffe“). Zudem muss der Referenzzähler des Blockes beim `BlockBuilder` erhöht werden, um eine Finalisierung des Blockes vor Fertigstellung der Konstruktion zu vermeiden. Nachdem der `Phi`-Knoten konstruiert wurde, wird der Zähler wieder reduziert. Näheres dazu ist dem Kapitel 4.3 zu entnehmen.

Als nicht immer trivial gestaltet sich die Konstruktion des tatsächlichen `Phi`-Knotens im Anschluss an die Konstruktion der restlichen Instruktionen. Zu diesem Zweck werden zunächst die – nun zur Verfügung stehenden – Firm-Knoten für alle Eingabe-Werte gesammelt und der zum jeweiligen LLVM-Block gehörige Firm-Block vom `NodeBuilder` abgerufen. Diese Paare werden dann zum Abruf in einer Hashtabelle abgelegt. Danach werden die Vorgänger-Blöcke des aktuellen Blocks in der von Firm gegebenen Reihenfolge iteriert und jeweils der zugehörige Wert aus der Hashtabelle abgerufen. Dieses Vorgehen stellt sicher, dass die Reihenfolge der Eingabe-Werte am `Phi`-Knoten zu jener der Vorgänger-Blöcke passend ist und damit eine korrekte Zuordnung sichergestellt wird.

Nun kann es vorkommen, dass zusätzlich zu den in LLVM genannten Blöcken weitere Firm-Blöcke erzeugt werden müssen, z. B. bei der in Abschnitt „Switch-Instruktion“ beschriebenen Behandlung der `switch`-Instruktionen. Dies ist in begrenztem Maße möglich, muss jedoch bei der Zuordnung der Abhängigkeiten hier berücksichtigt werden, denn ansonsten lässt sich möglicherweise dem in LLVM genannten Block kein entsprechender Firm-Block als Vorgänger zuordnen. Um dies dennoch zu ermöglichen, kann ein neuer Block als „Phi-Weiterleitung“ für den ursprünglichen LLVM-Block beim `BlockBuilder` registriert werden. Damit ist es nun möglich, die Abhängigkeit von dem LLVM-Block mit Hilfe des neuen Blocks aufzulösen. Zu beachten ist allerdings, dass die Weiterleitungs-Relation nicht transitiv ist.

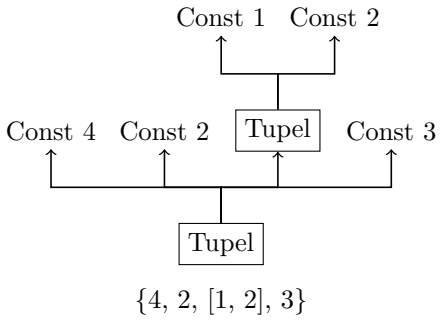


Abbildung 4.4: Zusammengesetzte Typen als Tupel

Zusammengesetzte Typen

Wie schon bei den BigInt-Typen erwähnt, können Firm-Knoten keine zusammengesetzten Typen handhaben, da keine entsprechenden Modi zur Verfügung stehen. Es ist auf einfache Weise auch nicht möglich, diese Strukturen im Speicher nachzubilden, da die Werte den SSA-Eigenschaften genügen und somit eine ausführliche Lebenszeit-Analyse und eine entsprechende Speicherzuteilung notwendig wäre.

Es ist allerdings zu beobachten, dass sich die direkte Verwendung zusammengesetzter Strukturen (ohne Zeiger und Speicherzugriffe) primär auf sehr kleine Strukturen konzentriert. Beispielsweise werden die zwei Komponenten einer komplexen Zahl vom Fortran-Frontend als ein solcher Struktur-Typ behandelt. Aus diesem Grunde stellt die Verwendung des Tupel-Modus in Firm (siehe Abbildung 4.4) die einfachste – wenn auch sicher nicht optimalste – Lösung dar. Sie hat den Vorteil, dass die Problematik der Speicherzuteilung auf den Registerzuteiler von Firm abgewälzt wird, der die verschiedenen Felder der Struktur später als individuelle Werte behandelt und diese gegebenenfalls auch über den Speicher auslagert.

Für die Umsetzung sind im Wesentlichen drei Aspekte relevant: 1. Beim Speichern und Laden zusammengesetzter Werte muss das jeweilige Tupel in eine Datenstruktur serialisiert oder aus ihr heraus deserialisiert werden. 2. Bei Funktionsparametern und -rückgaben ist ein entsprechendes Vorgehen notwendig, da Tupel nicht direkt übergeben werden können. 3. Für den Zugriff auf das Struktur-Tupel müssen die `extractvalue`- und `insertvalue`-Instruktionen bereitgestellt werden. Die ersten beiden Punkte lassen sich auch auf BigInt-Typen beziehen, die ebenso als Tupel umgesetzt werden.

Speichern und Laden Zur Serialisierung des Tupels an einer gegebenen Speicheradresse werden der Reihe nach für jeden Wert des Tupels entsprechende `Store`-Knoten konstruiert. Zur Berechnung der jeweiligen Zieladressen wird dabei auf die bestehende `getelementptr`-Infrastruktur zurückgegriffen (siehe Abschnitt „Adressberechnung“). Notwendige Typ-Informationen lassen sich vom entsprechenden LLVM-Typ ableiten. Besitzt eines der

Felder seinerseits einen zusammengesetzten Datentyp, so wird die Serialisierung rekursiv für das jeweilige Feld aufgerufen. In Firm wird dies durch ein weiteres Tupel in dem bereits betrachteten Tupel abgebildet.

Die Deserialisierung verläuft weitestgehend analog, indem für jedes Feld ein `Load`-Knoten erzeugt wird und die Ergebnisse aller `Load`-Knoten am Ende zu einem Tupel zusammengefasst werden. Sollte eine rekursive Deserialisierung erfolgen, so wird das dabei erzeugte Tupel in das bestehende Tupel eingebettet.

Funktionsaufrufe Da Tupel nicht direkt übergeben werden können, bieten sich im Wesentlichen zwei Möglichkeiten an: entweder man zerlegt das Tupel in mehrere Parameter bzw. Rückgabewerte oder man legt es im Speicher ab und übergibt damit eine Kopie der Struktur, die in der aufgerufenen Methode gegebenenfalls wieder deserialisiert wird. Die erste Variante krankt dabei an zwei wesentlichen Punkten: sie erfordert zum einen eine Änderung der Funktionssignatur, was durch externe Aufrufe nicht berücksichtigt wird und zum anderen ermöglicht das libFirm-Backend nicht beliebig viele Rückgabewerte. Damit verbleibt die zwar ineffiziente, dafür aber leicht umsetzbare Übergabe durch den Speicher.

Zu diesem Zweck wird mittels `Alloca`-Knoten ein entsprechender Speicherbereich auf dem Stack reserviert, wenn bei einem Funktionsaufruf entsprechende Parameter erkannt werden. Es erfolgt dann die übliche Serialisierung des Tupels in den Speicher und die Übergabe der dabei verwendeten Adresse an die Funktion, welche die Struktur dann gegebenenfalls wieder deserialisiert. Nach dem Funktionsaufruf wird der Speicher mittels `Free`-Knoten wieder freigegeben. Die Rückgabe zusammengesetzter Datentypen erfolgt auf ähnliche Weise, wobei zunächst in der aufgerufenen Funktion mit Hilfe des `FrameBuilder` ein Entity auf dem Frame-Typen reserviert wird, an dessen Adresse das Rückgabe-Tupel serialisiert wird. Das libFirm-Lowering behandelt diese Konstruktion entsprechend und ermöglicht dem Aufrufer den Zugriff auf die zurückgegebene Speicheradresse, von der das Tupel dann gegebenenfalls wieder deserialisiert werden kann.

Instruktionen Die `extractvalue`-Instruktion selektiert mittels Index-Liste einen Wert in der Struktur. Dies lässt sich einfach in Firm umsetzen, indem rekursiv für jeden Index ein `Proj`-Knoten konstruiert wird, der den jeweils nächsten Wert aus dem Tupel selektiert. Dies reicht aus, da man im Falle verschachtelter Strukturen jeweils wieder ein Tupel erhält.

Ein wenig schwieriger gestaltet sich die `insertvalue`-Instruktion, die den Wert, auf den die Index-Liste zeigt, durch einen neuen Wert ersetzt. Zu beachten ist, dass keine wirkliche Ersetzung stattfindet, sondern stattdessen ein neues Tupel erzeugt wird, in welchen der neue Wert eingesetzt wurde. Um dies zu ermöglichen, wird rekursiv ein neues Tupel aufgebaut, wobei alle unveränderten Einträge per `Proj`-Knoten aus dem alten Tupel extrahiert und ohne Veränderung in das neue Tupel übernommen werden. Bei dem durch den jeweiligen Index bezeichneten Eintrag erfolgt nun entweder eine Rekursion,

um gegebenenfalls ein verschachteltes Tupel zu konstruieren oder die Einsetzung des vorgegebenen Wertes, wenn das Ende der Index-Liste erreicht wurde.

Da die Tupel in Firm nur als Hilfskonstrukte dienen, tauchen sie im Firm-Graphen nicht als zusammengefasste Objekte auf. Stattdessen werden alle Einträge in der Struktur als individuelle Werte gehandhabt. Dies hat den Nebeneffekt, dass beim Laden und Speichern an derselben Adresse solche `Load`- und `Store`-Knoten durch die Optimierungen entfernt werden können, bei denen sich tatsächlich keine Werte ändern.

Switch-Instruktion

Mit Hilfe des `Cond`-Knoten und einem Integer als Eingabewert lässt sich die `switch`-Instruktion von LLVM emulieren. Dabei wird jeder mögliche Sprung mit einem `Proj`-Knoten aus dem `Cond`-Knoten herausprojiziert, wobei der dabei verwendete Index mit dem jeweiligen Case-Wert übereinstimmt. Im Gegensatz zu LLVM erlaubt Firm aber nur 32-Bit-Integer als Selektionswert. Für Modulo-Typen kleiner 32 Bit ließe sich dies zwar durch eine Normalisierung lösen, jedoch kann ein 64-Bit-Switch auf diese Weise nicht realisiert werden, ebenso wie dies mit BigInt-Typen nicht möglich wäre. Da solche `switch`-Anweisungen durchaus auch vom Frontend erzeugt werden, müssen sie gegebenenfalls durch ein Steuerfluss-Konstrukt mit mehreren Blöcken ersetzt werden.

Die naive Implementation durch eine Abfolge von `if`- und `else`-Blöcken ließe sich zwar einfach implementieren, nutzt aber nicht die vorhandene Ordnung der Case-Werte aus. Stattdessen wird eine sogenannte If-Kaskade aufgebaut, die eine Art binäre Suche zwischen den verfügbaren Case-Werten durchführt (Abbildung 4.5). Eine ähnliche Konstruktion wird bereits durch libFirm selber vorgenommen, wenn die `switch`-Konstruktion eine zu große Sprungtabelle erfordern würde, kann jedoch in diesem Fall nicht genutzt werden, da schon die Konstruktion des entsprechenden `Cond`-Knoten scheitert. Aus diesem Grund wäre gegebenenfalls auch eine Anpassung in libFirm denkbar, um hier auch 64-Bit-Integer zuzulassen. In Bezug auf LTF wäre diese Lösung aber auch nicht auf die BigInt-Typen erweiterbar (wobei dieser Fall vermutlich eher hypothetischer Natur ist).

Die Umsetzung der If-Kaskade erfolgt, indem zunächst alle Case-Werte gesammelt und sortiert werden. Danach wird rekursiv der Suchraum halbiert, indem jeweils ein Vergleich mit dem in der Mitte des Suchraumes stehenden Case-Wertes erfolgt. In den beiden Folge-Blöcken erfolgt rekursiv eine weitere Aufteilung, bis letztentlich der zutreffende Case-Wert und damit das Sprungziel feststeht. Zu beachten ist dabei, dass zur Sortierung der Case-Werte derselbe Vergleichsoperator verwendet werden muss wie bei der Konstruktion der Vergleiche in Firm. Insbesondere spielt es eine Rolle, ob vorzeichenlos oder -behaftet verglichen wird. Davon abgesehen muss für jeden Block, der später zu einem der Sprungziele führt, eine Phi-Weiterleitung beim `BlockBuilder` eingerichtet werden, damit dieser im Falle eines `Phi`-Knotens die bis dato unbekannten neuen Blöcke dem ursprünglichen Block mit der `switch`-Instruktion zuordnen kann (vgl. Abschnitt „Phi-Instruktionen“).

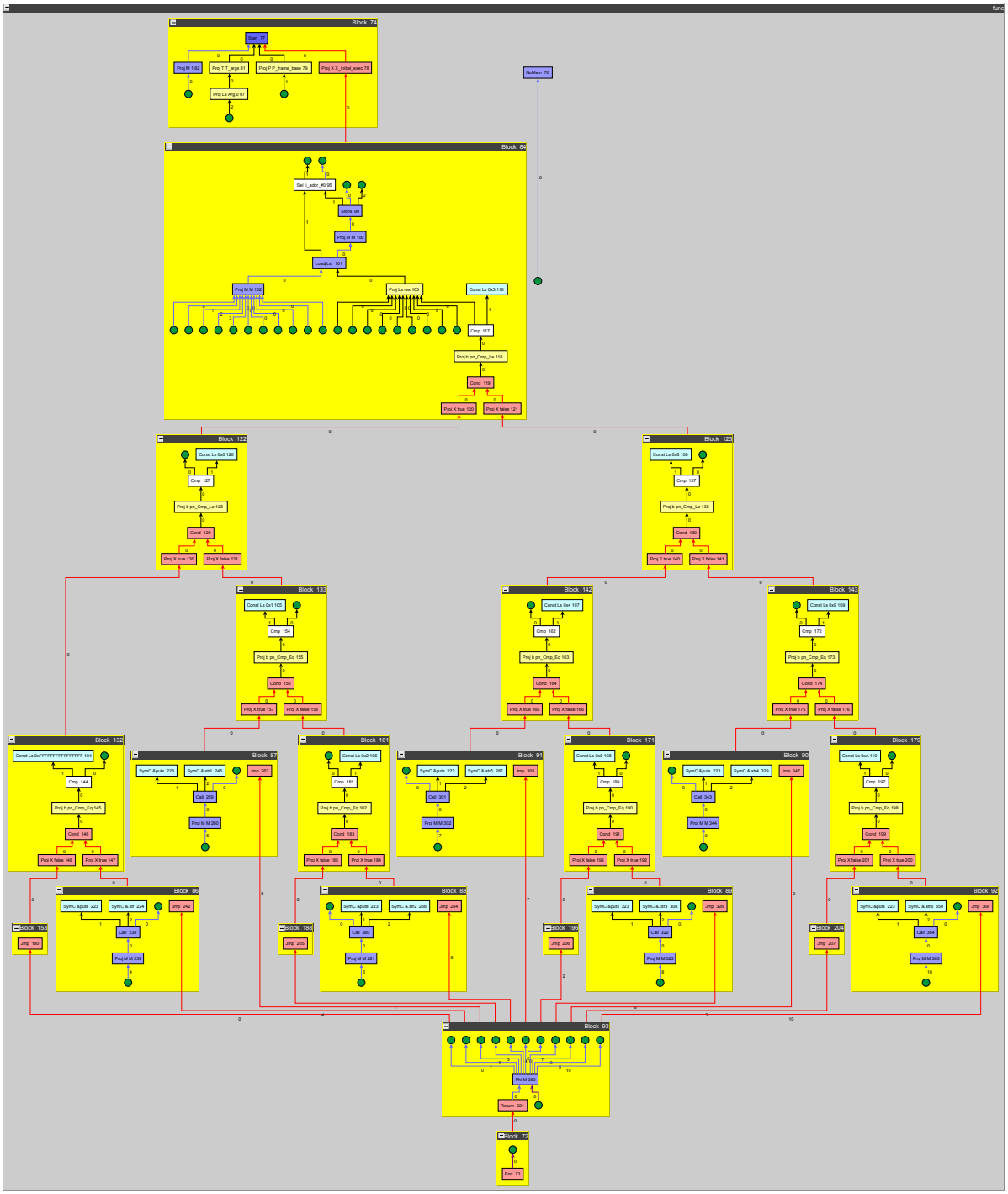


Abbildung 4.5: Beispiel einer If-Kaskade

Select-Instruktion

Die `select`-Instruktion ist mit dem ternären `?:`-Operator in C/C++ vergleichbar. Sie liefert als Rückgabewert einen der zwei angegebenen Operanden, wobei die Auswahl anhand eines dritten booleschen Operanden erfolgt. Firm stellt mit dem `Mux`-Knoten dieselbe Funktionalität bereit.

Häufig kann das Backend dies mit einer einzigen entsprechenden Assembler-Instruktion umsetzen, z. B. mit der Conditional-Move-Instruktion (`cmov`) bei x86. Je nach verwendeten Modi, Backend und Zielplattform kann es aber sein, dass entsprechende Instruktionen nicht zur Verfügung stehen, so dass der Knoten durch ein entsprechendes Steuerfluss-Konstrukt ersetzt werden muss. Dies findet in einer Lowering-Phase statt, die im Rahmen der Arbeit in libFirm ergänzt wurde. Sie ersetzt den `Mux`-Knoten durch einen bedingten Sprung in Kombination mit einem `Phi`-Knoten, welcher den jeweiligen Wert in Abhängigkeit vom getätigten Sprung auswählt. Im Wesentlichen entspricht dies einer zum ternären Operator äquivalenten Konstruktion mittels `if` und `else`.

Variable Argumentlisten

Funktionen mit einer variablen Anzahl an übergebenen Argumenten werden in LLVM mit Hilfe verschiedener Intrinsics, sowie der `va_arg`-Instruktion umgesetzt. Dabei wird laut Dokumentation der `llvm.va_start`-Intrinsic ein Zeiger übergeben, wobei die Datenstruktur an der Zieladresse „abhängig von der Zielplattform“ initialisiert wird. Die `va_arg`-Instruktion kann dann zusammen mit diesem Zeiger verwendet werden, um ein Argument mit dem gegebenen Typ von der Argumentliste abzurufen. Unklar bleibt dabei, wie der Zeiger konstruiert werden soll, wenn die Datenstruktur auf den dieser zeigt plattformabhängig ist.

Betrachtet man die Ausgabe der LLVM-Frontends, so sind zwei Dinge zu beobachten: einerseits dient als „Datenstruktur“ ein einzelner Zeiger, dessen Zeiger wiederum an `llvm.va_start` übergeben wird und andererseits wird bei den getesteten Frontends die `va_arg`-Instruktion nie benutzt, sondern ein Zugriff mittels `getelementptr`-Instruktion umgesetzt, wobei stillschweigend angenommen wird, dass `llvm.va_start` den übergebenen Zeiger auf die Adresse des ersten variablen (nicht benannten) Parameters initialisiert. Nach jedem Zugriff wird weiterhin der Zeiger aktualisiert, damit er auf das jeweils nächste Argument zeigt.

Um also eine mit dem erzeugten Zwischencode kompatible Implementierung zu schreiben, benötigt man im Wesentlichen eine Implementierung von `llvm.va_start`, welche die Adresse des ersten variablen Parameters an die gegebene Zieladresse schreibt. Alles Weitere wurde bereits im Frontend heruntergebrochen. Eine entsprechende Umsetzung in Firm ist weitestgehend trivial und besteht im Wesentlichen darin, dass das Entity des letzten Funktionsparameters selektiert und die nächste Adresse auf dem Stack berechnet wird, die dem Alignment genügt. Der resultierende Wert wird mit einem `Store`-Knoten an die Zieladresse geschrieben. Als ebenso trivial entpuppt sich die Implementierung der

`llvm.va_copy`-Intrinsic, die lediglich den Zeiger von gegebener Quell- zur Zieladresse kopieren muss, um den Zustand der Argument-Iteration zu übernehmen.

Zusätzlich dazu wurde eine entsprechende Implementation der `va_arg`-Instruktion geschrieben, um die Kompatibilität mit potentiellen weiteren Frontends oder bei zukünftigen Änderungen zu erhöhen. Dabei wird dieselbe Konstruktion verwendet, wie sie vom Frontend beim Laden eines Argumentes genutzt wird, d. h. an der aktuellen Adresse wird der Wert des Parameters ausgelesen und abhängig von der Größe des angeforderten Typs mit einem `Add`-Knoten die Adresse für den Zugriff auf den nächsten Parameter berechnet und mittels `Store`-Knoten zurückgeschrieben.

4.6 Optimierungs-Framework

Der Optimierungsablauf in LTF ist weitestgehend an jenem von CParser angelehnt. Die Optimierungsstrategie lässt sich bei Benutzung der LTF-Bibliothek austauschen, indem eine eigene Implementierung der `Optimizer`-Klasse geschrieben wird. Prinzipiell stehen der abgeleiteten Klasse die von libFirm angebotenen Optimierungs-Pässe als Instanzen der `Pass`-Klasse zur Verfügung, welche sich mit der `getPass`-Methode abrufen lassen und eine Konfiguration und Ausführung der jeweiligen Optimierungsphasen ermöglichen.

Dabei lassen sich an einigen der Pässe individuelle Parameter festlegen, die über entsprechende Getter- und Setter-Methoden realisiert werden. Dies betrifft beispielsweise Grenzwerte für die Inlining-Heuristik oder Callback-Methoden, die von den einzelnen Pässen verwendet werden. Darüber hinaus lässt sich jeder Pass getrennt aktivieren oder deaktivieren. Die Optimierungsstrategie selber wird vom `Optimizer` in der `run`-Methode umgesetzt, wobei dieser die jeweiligen Pässe in entsprechender Reihenfolge über deren `run`-Methode aufrufen kann. Dies betrifft nicht nur Optimierungen, sondern auch Lowering-Pässe, da deren Platzierung im Zusammenhang mit den Optimierungs-Pässen Teil der Optimierungsstrategie sein kann. Abgesehen hiervon können Implementierungen der `Optimizer`-Klasse verschiedene Callback-Methoden überschreiben, die es beispielsweise ermöglichen, bestimmte transparente Optimierungen in libFirm vor der Konstruktionsphase oder vor dem Backend zu konfigurieren.

Zur einfachen Steuerung kann jede `Optimizer`-Implementierung die `setOptLevel`-Methode implementieren, mit welcher das Optimierungslevel – üblicherweise ausgehend vom Kommandozeilenparameter – gesetzt wird. Hierbei besteht insbesondere die Gelegenheit, individuelle Pässe an- und auszuschalten oder deren Parameter entsprechend dem Optimierungslevel zu setzen. Zusätzliche Parameter können natürlich gegebenenfalls durch weitere Methoden zur Verfügung gestellt werden. Standardmäßig wird in LTF die `CParserOptimizer`-Klasse verwendet, welche die Optimierungsstrategie und -level des CParser-Compilers in LTF umsetzt.

5 Evaluierung

5.1 Vollständigkeit

LTF ist inzwischen in der Lage, umfangreiche Teile des LLVM-Zwischencodes zu übersetzen. Verbleibende Schwächen existieren – wie teilweise bereits erwähnt – bei der Unterstützung von Exceptions und den damit verbundenen Intrinsics, der Übersetzung einiger Funktions- und Parameterattribute, Vektor-Typen und der Unterstützung diverser erweiterter Funktionen mittels LLVM-Intrinsics (z. B. atomare Speicherzugriffe, Debugging-Informationen, Garbage-Collection-Support, ...). Teilweise sind zur Laufzeit leider noch Fehler zu beobachten, deren Ursache nicht abschließend geklärt ist.

Welche Benchmarks der SPEC-CPU2000-Suite zur Zeit mit LTF lauffähig sind, kann Tabelle 5.1 entnommen werden. Dabei gilt ein Testlauf nur dann als fehlerfrei, wenn das Kompilat den jeweiligen `ref`-Datensatz des Benchmarks ohne fehlerhafte Ausgabe vollständig verarbeitet. Dies ist erforderlich, um einen zulässigen Gesamtdurchlauf gemäß den Regeln der SPEC-Suite durchzuführen.

Die meisten verbleibenden Fehler lassen sich auf noch nicht behobene Fehler in LTF oder libFirm zurückführen. Teilweise sind dies Floating-Point-Abweichungen, wie im Falle von `252.eon`, welcher zwar auf mehreren Optimierungsstufen lauffähig ist, dabei aber leicht abweichende Ergebnisse ausgibt. Nur durch Deaktivieren der Optimierungen lässt sich das korrekte Ergebnis erzwingen. Ein ähnliches Problem könnte auch bei `178.galgel` auftreten, welcher mit Optimierungen nicht mehr terminiert. Davon abgesehen treten vor allem mit LTF-Optimierungslevel `-O3` noch vereinzelte fehlerhafte Kompilate oder Fehler zur Kompilierzeit auf, die noch nicht abschließend geklärt oder behoben sind. Dies betrifft auch `186.crafty`, welcher auf den verschiedenen Optimierungslevels noch an demselben Fehler scheitert.

Die nicht umgesetzten LLVM-Funktionalitäten zeigen sich hier nur für die Fehler bei `176.gcc` verantwortlich. Hierbei fehlt die Umsetzung der `llvm.stacksave`- und `llvm.stackrestore`-Intrinsics, welche hier vom Frontend erzeugt werden. Dies liegt daran, dass libFirm zur Zeit kein entsprechendes Konstrukt bereitstellt und eine Umsetzung mittels `Free`-Knoten problematisch wäre.

Weiterhin sei angemerkt, dass für `186.crafty` einige System-Header ausgetauscht wurden, um die Makros `FD_ZERO`, `FD_SET` und `FD_ISSET` neu zu definieren. Dies liegt an der Umsetzung mittels Inline-Assembler, welche von LTF noch nicht übersetzt werden kann. Bei späteren Vergleichen zwischen verschiedenen Compilern wurde dies ebenso umgesetzt, um vergleichbare Ergebnisse zu gewährleisten.

	Optimierungslevel															
LLVM	3				2				1				0			
libFirm	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
164.gzip	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
175.vpr	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
176.gcc	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
181.mcf	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
186.crafty	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
197.parser	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
252.eon	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
253.perlbmk	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
254.gap	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
255.vortex	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
256.bzip2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
300.twolf	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
168.wupwise	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
171.swim	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
172.mgrid	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
173.applu	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
177.mesa	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
178.galgel	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
179.art	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
183.equake	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
187.facerec	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
188.ammp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
189.lucas	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
191.fma3d	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
200.sixtrack	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
301.apsi	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓

Tabelle 5.1: Gültige Testläufe (`ref`-Datensatz)

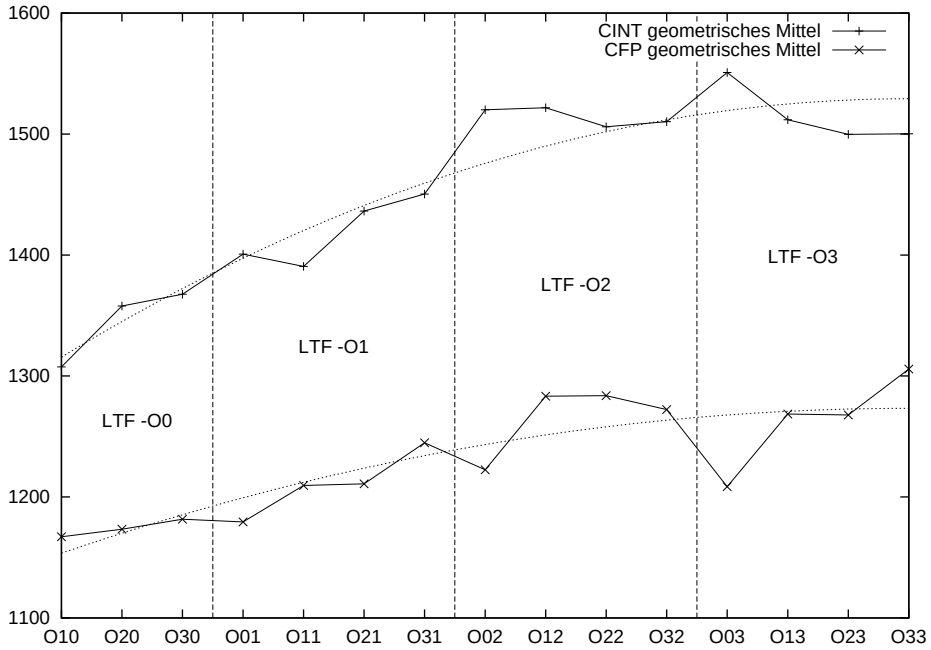


Abbildung 5.1: Verschiedene Optimierungslevel

5.2 Leistungsbewertung

Zum Erfassen der Messwerte kamen mehrere identisch ausgestattete Systeme mit 2,6 GHz Intel Pentium Dual-Core-E5300 Prozessor, sowie 4 GB Hauptspeicher zu Einsatz. Als Betriebssystem diente dabei Ubuntu Linux 9.10 mit Kernel-Version 2.6.31. Um dennoch potentielle Unterschiede zwischen den Systemen zu berücksichtigen, wurden die verschiedenen Optimierungslevel eines Benchmarks stets auf demselben Rechner getestet, so dass diese in jedem Fall vergleichbar sind. Damit wurden für alle Benchmarks und Optimierungsstufen Messungen durchgeführt, um zu untersuchen, wie sich LLVM- und LTF-Optimierungslevel gegenseitig beeinflussen. Die detaillierten Einzelergebnisse sind Tabelle 7.1 aus dem Anhang zu entnehmen. Dabei wurde jeweils der SPECint_base2000-Messwert erfasst. Als Zielarchitektur für libFirm wurde `core2` gewählt, um eine möglichst optimale Ausgabe zu erzeugen.

Das angegebene Optimierungslevel fasst das in LLVM angegebene Optimierungslevel und jenes von LTF zusammen. Dabei steht die erste Ziffer – gemäß der Reihenfolge der Anwendung – für das Optimierungslevel in LLVM, welches an das Frontend (`llvm-gcc`, `llvm-gfortran`, ...) übergeben wird, während die zweite Ziffer das LTF-Optimierungslevel steuert.

Um einen besseren Überblick zu erhalten, wurde außerdem für jedes Optimierungslevel das geometrische Mittel über die Integer- und Floating-Point-Benchmarks berechnet.

Um den nicht funktionierenden Testläufen Rechnung zu tragen, wurde die Auswahl auf jene Benchmarks beschränkt, die in jedem Optimierungslevel fehlerfrei laufen. Die dabei erhaltenen Ergebnisse sind in Abbildung 5.1 grafisch dargestellt.

Dabei ist für die ersten beiden Segmente, d. h. für die LTF-Optimierungslevel `-O0` und `-O1` noch ein positiver Effekt der LLVM-Optimierungen zu bemerken. Dieser Effekt wird bei den höheren LTF-Optimierungsleveln zunehmend unklarer. Bei den Integer-Benchmarks erzielen zusätzliche LLVM-Optimierungen sogar tendenziell schlechtere Ergebnisse, so dass das beste Ergebnis sogar ohne jegliche LLVM-Optimierungen bei `-O03` erzielt wird.

Die Floating-Point-Benchmarks dagegen zeigen ein widersprüchliches Bild. Auffällig ist im Vergleich zu den Integer-Benchmarks ein deutlicher Abfall des Ergebnisses, wenn die LLVM-Optimierungen deaktiviert sind. Dies könnte möglicherweise mit der fehlenden Unterstützung der – bei Fortran häufig verwendeten – zusammengesetzten Typen in Firm zusammenhängen, wodurch die Optimierungen mit diesem Konzept nicht direkt umgehen können. Darüber hinaus ist der beim Lowering erzeugte Zwischencode vergleichsweise ineffizient und speicherlastig. Betrachtet man die restlichen LLVM-Optimierungslevel, so ist nur im letzten Segment eine Verbesserung zu bemerken, die allerdings der Beobachtung bei den Integer-Benchmarks entgegensteht.

Compiler-Vergleich

Für den Vergleich mit anderen Compilern wurden aufgrund der unterschiedlichen Tendenzen für Integer- und Floating-Point-Benchmarks zwei verschiedene Optimierungslevel gewählt. Nach Abbildung 5.1 bieten sich `-O03` für die Integer-Benchmarks und `-O33` für die Floating-Point-Benchmarks an. Für `252.eon` und `178.galgel` musste aufgrund der auftretenden Fehler auf `-O00` zurückgegriffen werden. Mit diesen Einstellungen wurde auf einem der Rechner ein vollständiger Durchlauf der SPEC-CPU2000-Suite durchgeführt, dessen Ergebnis im Anhang auf den Seiten 50 und 51 angefügt ist.

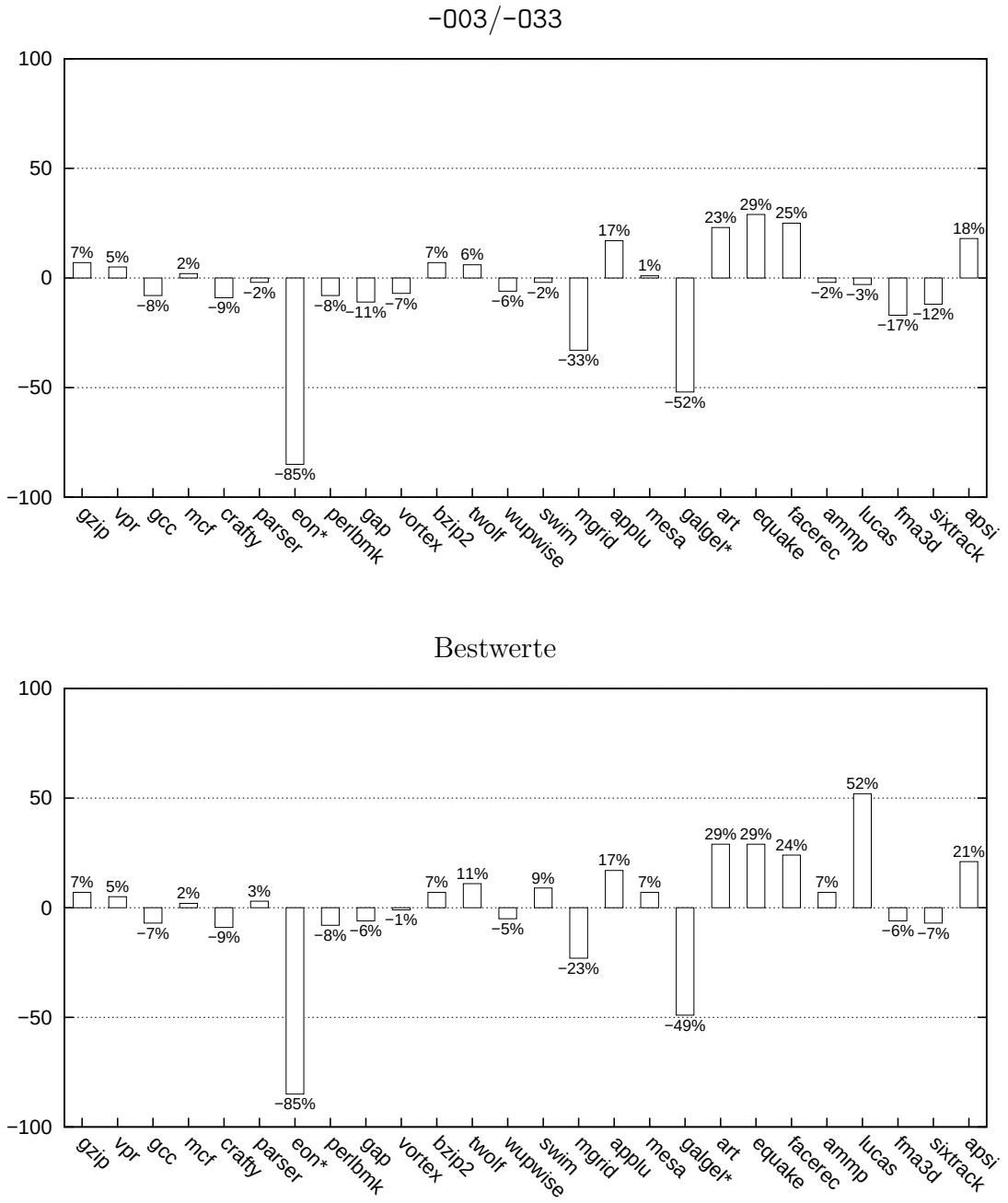
Davon abgesehen wurden mit vergleichbaren Einstellungen (`-O3`, `-march=core2` und `-fomit-frame-pointer`) die Ergebnisse der anderen Compiler ermittelt. Aus Gründen der Vergleichbarkeit wurden die SSE-Erweiterungen bei LLVM deaktiviert (`-mno-sse`), da CParser und LTF kein SSE verwenden und auch GCC ohne explizite Anforderung darauf verzichtet. Die dabei erzielten Resultate sind in Tabelle 5.2 aufgeführt. Zusätzlich dazu wurden die jeweils besten Einzelergebnisse von LTF mit aufgeführt (aus Tabelle 7.1), da je nach Kombination der Optimierungslevel noch bessere Resultate möglich sind.

Ein grafischer Vergleich der LTF-Ergebnisse mit `llvm-gcc` ist Abbildung 5.2 zu entnehmen. Auffällig sind dabei die schlechten Ergebnisse von `252.eon` und `178.galgel`, aufgrund der fehlenden Optimierungen. Schließt man diese aus den Überlegungen aus, so erhält man – bezogen auf das geometrische Mittel der verbleibenden Benchmark-Ergebnisse – etwa $-1,7\%$ bei den Integer-Benchmarks und $+6,6\%$ bei den Floating-Point-Benchmarks. Bei selektiver Auswahl der jeweils besten Einzelergebnisse erhält man $+0,2\%$ und $+16\%$.

	Clang `-O3`	CParser `-O3`	GCC `-O3`	LLVM-GCC `-O3`	LTF-GCC `-O03/-O33`	LTF-GCC Max
164.gzip	1245	1359	1309	1292	1385	1386
175.vpr	1496	1465	1568	1476	1554	1557
176.gcc	2279	2218	2613	2293	2107	2136
181.mcf	1749	1837	1803	1800	1843	1835
186.crafty	1987	1885	1908	1977	1806	1801
197.parser	1347	1446	1595	1467	1440	1509
252.eon	-	-	1420[2]	1640	250[1]	251[1]
253.perlbmk	1109	2148	2505	2029	1864	1873
254.gap	2025	1807	2205	2059	1840	1929
255.vortex	1866	1731	1956[3]	1848	1727	1835
256.bzip2	1481	1585	1533	1474	1576	1579
300.twolf	2575	2743	2688	2564	2728	2837
168.wupwise	-	-	1719	1510	1421	1436
171.swim	-	-	2523	2061	2027	2241
172.mgrid	-	-	1135	977	658	753
173.applu	-	-	1568	1186	1384	1391
177.mesa	1286	1132	1662	1354	1364	1447
178.galgel	-	-	1703[4]	1882	900[1]	952[1]
179.art	1251	2528	1870	1424	1748	1830
183.equake	1661	2124	2447	1774	2291	2286
187.facerec	-	-	1807	1208	1515	1503
188.ammp	1122	607	1253	1172	1144	1256
189.lucas	-	-	2038	1304	1263	1976
191.fma3d	-	-	1482	1397	1158	1317
200.sixtrack	-	-	713	691	608	644
301.apsi	-	-	1805	1152	1356	1394
Spec-Int	-	-	1873	1791	1508	1535
Spec-FP	-	-	1623	1318	1265	1374

[1] Optimierungslevel `-O00`
[2] Optimierungslevel `-O1`
[3] Ohne `-fomit-frame-pointer`
[4] Mit `-ffloat-store`

Tabelle 5.2: Compiler-Vergleich



Bestwerte

Abbildung 5.2: Resultate im Vergleich zu LLVM

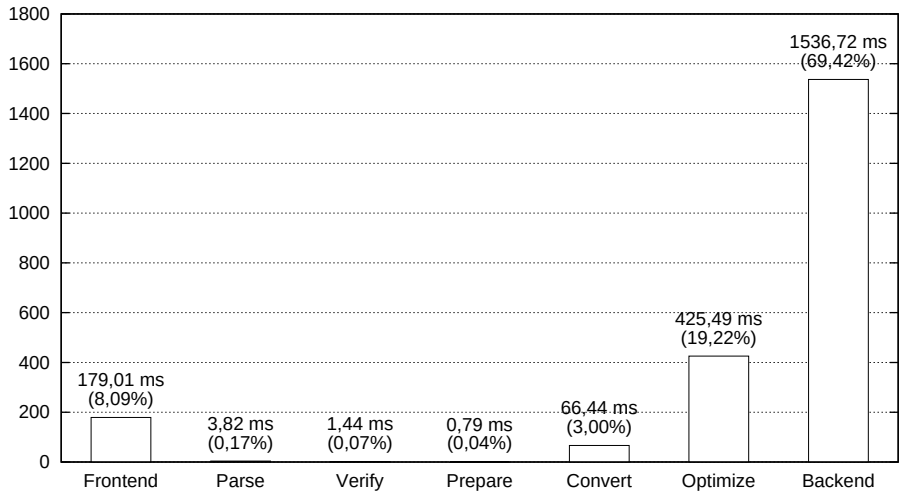


Abbildung 5.3: Laufzeit der Pipeline-Stufen

5.3 Laufzeit

Um die Laufzeit der eigentlichen Konvertierung zu erfassen, wurde die in den einzelnen Stufen der Pipeline verbrachte Zeit beim Kompilieren der kompletten SPEC2000-Suite auf dem Testsystem erfasst und jeweils das arithmetische Mittel der resultierenden Zeiten ermittelt. Dabei ist die Ausführung von `llvm-gcc`, um den LLVM-Zwischencode zu generieren, als Frontend-Phase zusätzlich angegeben. Das Ergebnis ist in Abbildung 5.3 dargestellt. Im Schnitt kostet die Konvertierung von LLVM nach Firm – bezogen auf die SPEC2000-Benchmarks – also nur etwa 3% der Compiler-Laufzeit. Dies ist aufgrund der weitestgehend linearen Transformation und der häufigen 1:1-Entsprechung zwischen LLVM-Instruktion und Firm-Knoten zu erwarten gewesen. Entsprechendes gilt für die Vorbereitungsphase, die weitestgehend linear die Befehlslisten durchläuft und gegebenenfalls Ersetzungen durchführt. Der Schwerpunkt liegt damit weiterhin bei den Optimierungen und dem Backend.

6 Fazit

6.1 Zusammenfassung

Erfreulicherweise ist LTF bereits in der Lage, auch viele komplexe Programme zu übersetzen. Zwar existieren noch einige Fehler, aber dass sich die vollständige SPEC CPU2000 Suite – wenn auch mit Einschränkungen in Bezug auf die Optimierungslevel – auf diese Weise übersetzen lässt, ist sicher als Erfolg zu werten. Dies gilt ebenso für die Unterstützung von Fortran und C++, die – zumindest im Falle von Fortran – ursprünglich nicht ins Auge gefasst war, sich aber dann mit vergleichsweise wenig Aufwand noch umsetzen ließ.

In Hinblick auf die Leistungsfähigkeit der Optimierungen lassen sich teilweise deutliche Steigerungen gegenüber LLVM beobachten (z. B. `189.lucas` bei geschickter Auswahl des Optimierungslevels), genauso häufig aber auch das Gegenteil (z. B. `172.mgrid`). Betrachtet man die Mittelwerte, ist zwar – wenn die Optimierungen anwendbar sind – eine gewisse Steigerung zu beobachten, diese hängt aber auch von der Auswahl der Benchmarks ab. Zusammenfassend ist das Resultat jedoch durchaus mit LLVM vergleichbar und leidet durch die Konvertierung zu Firm nicht merklich.

6.2 Ausblick

An zusätzlichen Funktionen bieten sich bei einer weiteren Entwicklung insbesondere die Unterstützung von Exceptions und damit des vollständigen Funktionsumfanges von C++ an. Allerdings dürfte dies aufgrund der nötigen Anpassungen in libFirm, um das Zero-Cost-Exception-Handling umzusetzen, mit einigem Aufwand verbunden sein. Einfacher könnte sich dagegen die Umsetzung der ebenso noch fehlenden Unterstützung von Inline-Assembler-Code gestalten, da ein entsprechender Mechanismus in libFirm bereits vorgesehen ist. Weiterhin interessant wäre auch noch die Umsetzung von Vektor-Instruktionen auf die entsprechenden Funktionalitäten in libFirm, die hier aufgrund des zusätzlichen Aufwandes, bei zunächst geringem Nutzen ausgespart wurde. Eine Umsetzung der Floating-Point-Instruktionen auf SSE-Instruktionen könnte eventuell die Floating-Point-Ungenauigkeiten bei `252.eon` und (möglicherweise) `178.galgel` vermeiden und einen merklichen Leistungsschub bringen. Dies betrifft jedoch eher libFirm, dessen SSE-Umsetzung zur Zeit noch Probleme aufweist.

Eine Übertragung der Debug-Symbole könnte für den praktischen Einsatz interessant sein und gegebenenfalls das Aufspüren noch vorhandener Laufzeit-Fehler erleichtern.

Auch wäre es interessant zu sehen, ob eine Performance-Verbesserung zu beobachten ist, wenn die aktuelle, vergleichsweise ineffiziente Behandlung zusammengesetzter Typen im Graphen intelligenter gestaltet werden kann (Kapitel 4.1, Abschnitt „Zusammengesetzte Typen“). Auf lange Sicht sind aber auch noch Änderungen in libFirm geplant, um unter anderem das Typ-System auch für die Graphen zu verwenden, womit sich auch das Problem der zusammengesetzten Typen lösen dürfte.

Beim Thema Vollständigkeit bestehen vor allem bei der Unterstützung der BigInt-Typen noch große Lücken. Auch die `bitcast`-Instruktion wird noch nicht vollständig unterstützt. Dies betrifft beispielsweise die Möglichkeit, `bitcast` in konstanten Ausdrücken und Initialisierern zu verwenden, was aufgrund des Lade/Speicher-Mechanismus in Firm auf dem Const-Graphen nicht möglich ist (Kapitel 4.5, Abschnitt „Konvertierungen“). Denkbar wäre auch eine Erweiterung von Firm um einen `Bitcast`-Knoten, der mit der LLVM-Instruktion vergleichbar ist. Bei Versuchen mit dem Clang-Frontend von LLVM [Cla] traten außerdem auch einige Bit-Casts auf, die nicht direkt unterstützt werden. Beispielsweise ist eine direkte Konvertierung des `x86_fp80`-Datentyps nach `i80` zur Zeit nicht möglich, könnte jedoch mit vertretbarem Aufwand noch nachgerüstet werden.

In Bezug auf Skriptsprachen könnte insbesondere eine ähnliche Integration von Garbage-Collection-Informationen, wie es in LLVM der Fall ist, interessant sein. Da hier allerdings aufgrund der dynamischen Sprachstruktur häufig der JIT-Compiler von LLVM fest in das Laufzeitsystem integriert wird und eine statische Übersetzung nicht immer möglich ist, bleibt ohnehin fraglich, ob sich Firm in diesem Zusammenhang überhaupt einsetzen ließe, ohne das gesamte Laufzeitsystem anzupassen.

Davon abgesehen wäre es noch interessant, diverse weitere aktuelle und zukünftige Frontends auf die Kompatibilität im Zusammenhang mit LTF und libFirm zu testen. Gegebenenfalls eröffnen sich hier weitere Möglichkeiten, die sich ohne großen Aufwand umsetzen ließen.

7 Anhang

	Optimierungslevel																	
LLVM	3				2				1				0					
libFirm	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0		
164.gzip	1310	1329	1281	1139	1308	1336	1281	1134	1341	1352	1275	1134	1386	1301	1240	672		
175.vpr	1436	1466	1454	1345	1439	1465	1446	1352	1467	1502	1369	1266	1557	1548	1444	917		
176.gcc	×	×	×	×	×	×	×	×	×	×	×	×	2136	2136	2132	1234		
181.mcf	1800	1815	1819	1781	1822	1824	1818	1787	1801	1809	1736	1701	1835	1822	1721	1377		
186.crafty	×	×	×	×	×	×	×	×	×	×	×	×	1796	1801	1799	1172		
197.parser	1509	1503	1382	1352	1494	1481	1331	1309	1495	1482	1249	1212	1435	1418	1191	755		
252.eon	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	251		
253.perlbmk	1586	×	1592	1831	1613	×	1581	1848	1623	1574	1573	1821	1873	1860	1849	1082		
254.gap	×	1810	1884	1620	×	1810	1865	1629	×	1831	1879	1601	1852	1850	1929	826		
255.vortex	×	1835	1724	1675	×	1750	1650	1584	×	1745	1642	1574	1711	1709	1596	1120		
256.bzip2	1487	1478	1371	1297	1481	1465	1364	1287	1491	1499	1374	1291	1579	1560	1469	778		
300.twolf	×	2794	2695	2557	×	2741	2691	2575	×	2763	2720	2527	2837	2767	2637	1448		
168.wupwise	1432	1434	1103	891	1378	1383	1106	1069	1431	1436	1094	1079	546	673	963	209		
171.swim	2098	2088	2083	2052	2120	2085	2082	2051	2184	2099	2081	2046	2241	2240	2201	1410		
172.mgrid	657	656	641	632	661	662	638	631	659	660	640	631	753	746	714	234		
173.applu	1387	1335	1034	966	1391	1338	1010	966	1336	1316	1032	991	1295	1224	941	410		
177.mesa	1383	1281	1297	1299	1175	1447	1196	1196	1261	1417	1244	1182	1322	1304	1147	899		
178.galgel	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	952		
179.art	1772	1751	1830	1651	1820	1776	1788	1570	1731	1803	1715	1570	1353	1434	1414	1746		
183.equake	2286	2035	2122	2111	2098	2110	2131	1956	2101	2013	2133	1946	2028	2113	2150	1112		
187.facerec	1503	1468	1408	1272	1460	1458	1409	1256	1446	1422	1345	1228	1190	1150	1121	582		
188.ammp	1132	1146	955	972	1141	1145	871	979	1137	1125	866	927	1248	1246	1256	695		
189.lucas	1271	1290	1674	1629	1234	1223	1584	1666	1309	1280	1634	1677	1976	1868	1387	1180		
191.fma3d	1153	975	1093	1274	1067	1114	976	1277	1114	1116	1060	1226	×	1317	1049	807		
200.sixtrack	567	544	644	616	549	552	624	577	504	564	622	575	632	626	633	309		
301.apsi	1394	1370	1287	1094	1374	1304	1271	1273	1284	1285	1295	1252	×	1280	1277	526		

Tabelle 7.1: SPEC CPU2000 Benchmark-Ergebnisse (`ref`-Datensatz)

CINT2000 Result

--

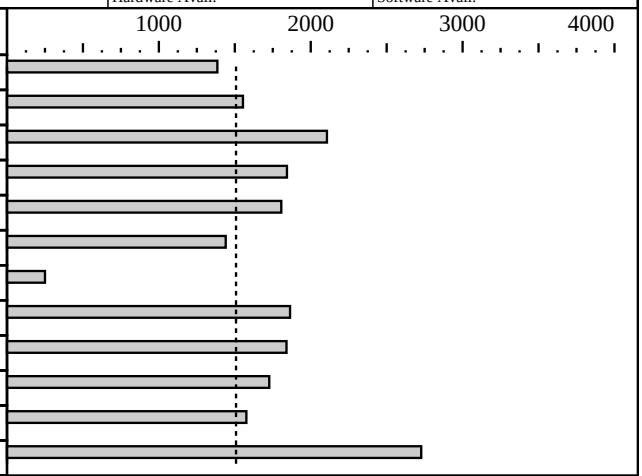
--

SPECint2000 = --

SPECint_base2000 = 1508

SPEC license #: -- | Tested by: -- | Test date: -- | Hardware Avail: -- | Software Avail: --

Benchmark	Reference Time	Base Runtime	Base Ratio	Runtime	Ratio
164.gzip	1400	101	1385		
175.vpr	1400	90.1	1554		
176.gcc	1100	52.2	2107		
181.mcf	1800	97.6	1843		
186.crafty	1000	55.4	1806		
197.parser	1800	125	1440		
252.eon	1300	520	250		
253.perlbmk	1800	96.6	1864		
254.gap	1100	59.8	1840		
255.vortex	1900	110	1727		
256.bzip2	1500	95.2	1576		
300.twolf	3000	110	2728		



Hardware

CPU:	Intel Pentium Dual-Core E5300
CPU MHz:	2.6 GHz
FPU:	--
CPU(s) enabled:	--
CPU(s) orderable:	--
Parallel:	--
Primary Cache:	--
Secondary Cache:	--
L3 Cache:	--
Other Cache:	--
Memory:	4 GiB
Disk Subsystem:	--
Other Hardware:	--

Software

Operating System:	Ubuntu Linux, Kernel 2.6.31-16-generic-pae 53-Ubuntu SMP
Compiler:	LTF-GCC r169 (LLVM 2.6, libFirm 27182) (ltf-gcc)
File System:	--
System State:	--

Notes/Tuning Information

```
Portability Flags:
  186.crafty:  -DLINUX_i386
  253.perlbmk: -DSPEC_CPU2000_NEED_BOOL -DSPEC_CPU2000_LINUX_I386
  252.eon:     -DHAS_ERRLIST -fpermissive -fno-exceptions
  178.galgel:  -ffixed-form
254.gap:    -DSYS_HAS_SIGNAL_PROTO -DSYS_HAS_MALLOC_PROTO -DSYS_HAS_CALLOC_PROTO -DSYS_IS_USG -DSYS_HAS_IOCTL_PROTO -DSYS_HAS_TIME_PROTO --fno-call-conv-opt

Optimization Settings:
  All:         -march=core2
  CFP2000:    Using -O33.
  CINT2000:   Using -O03.
  178.galgel: Using -O00, to avoid invalid output (unknown reason).
  252.eon:    Using -O00, to avoid invalid output (fp inaccuracy).

Other Changes:
  186.crafty: Replaced FD_ZERO, FD_SET, FD_ISSET in select.h, to avoid inline assembler.

Machine: i44pc45
```

Standard Performance Evaluation Corporation
info@spec.org
http://www.spec.org/

CFP2000 Result

--

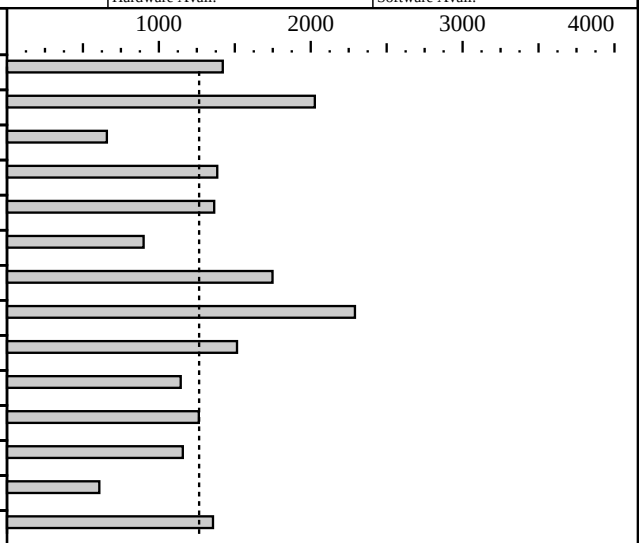
--

SPECfp2000 = --

SPECfp_base2000 = 1265

SPEC license #: -- | Tested by: -- | Test date: -- | Hardware Avail: -- | Software Avail: --

Benchmark	Reference Time	Base Runtime	Base Ratio	Runtime	Ratio
168.wupwise	1600	113	1421		
171.swim	3100	153	2027		
172.mgrid	1800	274	658		
173.applu	2100	152	1384		
177.mesa	1400	103	1364		
178.galgel	2900	322	900		
179.art	2600	149	1748		
183.equake	1300	56.7	2291		
187.facerec	1900	125	1515		
188.ammp	2200	192	1144		
189.lucas	2000	158	1263		
191.fma3d	2100	181	1158		
200.sixtrack	1100	181	608		
301.apsi	2600	192	1356		



Hardware

CPU:	Intel Pentium Dual-Core E5300
CPU MHz:	2.6 GHz
FPU:	--
CPU(s) enabled:	--
CPU(s) orderable:	--
Parallel:	--
Primary Cache:	--
Secondary Cache:	--
L3 Cache:	--
Other Cache:	--
Memory:	4 GiB
Disk Subsystem:	--
Other Hardware:	--

Software

Operating System:	Ubuntu Linux, Kernel 2.6.31-16-generic-pae 53-Ubuntu SMP
Compiler:	LTF-GCC r169 (LLVM 2.6, libFirm 27182) (ltf-gcc)
File System:	--
System State:	--

Notes/Tuning Information

```
Portability Flags:
  186.crafty:  -DLINUX_i386
  253.perlbmk: -DSPEC_CPU2000_NEED_BOOL -DSPEC_CPU2000_LINUX_I386
  252.eon:     -DHAS_ERRLIST -fpermissive -fno-exceptions
  178.galgel:  -ffixed-form
 254.gap:     -DSYS_HAS_SIGNAL_PROTO -DSYS_HAS_MALLOC_PROTO -DSYS_HAS_CALLOC_PROTO -DSYS_IS_USG -DSYS_HAS_IOCTL_PROTO -DSYS_HAS_TIME_PROTO --fno-call-conv-opt

Optimization Settings:
  All:         -march=core2
  CFP2000:     Using -O33.
  CINT2000:    Using -O03.
  178.galgel: Using -O00, to avoid invalid output (unknown reason).
  252.eon:     Using -O00, to avoid invalid output (fp inaccuracy).

Other Changes:
  186.crafty: Replaced FD_ZERO, FD_SET, FD_ISSET in select.h, to avoid inline assembler.
```

Standard Performance Evaluation Corporation
info@spec.org
http://www.spec.org/

Literaturverzeichnis

[CCE+] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI: *Itanium C++ ABI: Exception Handling*. `http://www.codesourcery.com/public/cxx-abi/abi.html`.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck: *Efficiently computing static single assignment form and the control dependence graph*. ACM Transactions on Programming Languages and Systems, 13:451–490, März 1991.

[Cla] *Clang-Compiler: A C Language Family Frontend for LLVM*. `http://clang.llvm.org`.

[CP95] Cliff Click and Michael Paleczny: *A Simple Graph-Based Intermediate Representation*. In *Intermediate Representations Workshop*, pages 35–49, 1995.

[CPa] *CParser-Compiler*. `http://sourceforge.net/projects/cparser`.

[Hac06] Sebastian Hack: *Register Allocation for Programs in SSA Form*. Dissertation, Universität Karlsruhe, Oktober 2006. `http://digbib.ubka.uni-karlsruhe.de/volltexte/302599`.

[HGB+] Sebastian Hack, Rubino Geiß, Michael Beck, Moritz Kroll, and Philipp Leiß: *yComp-Visualisierungs-Programm*. `http://www.info.uni-karlsruhe.de/software/ycomp`.

[Khr09] Khronos OpenCL Working Group: *The OpenCL Specification*, Oktober 2009. `http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf`.

[LA04] Chris Lattner and Vikram Adve: *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, März 2004. University of Illinois at Urbana-Champaign.

[LA10] Chris Lattner and Vikram Adve: *LLVM Assembly Language Reference Manual*, Februar 2010. `http://llvm.org/releases/2.6/docs/LangRef.html`.

[Lat02] Chris Lattner: *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, Dezember 2002. `http://llvm.cs.uiuc.edu`.

[Lib] *LibFirm Website*. `http://www.libfirm.org`.

[Lin02] Götz Lindenmaier: *lib*Firm*: A Library for Compiler Optimization Research Implementing Firm*. Technical Report 2002-5, Universität Karlsruhe, September 2002. http://digbib.ubka.uni-karlsruhe.de/volltexte/5002002.

[LLVa] *The LLVM Compiler Infrastructure*. http://llvm.org.

[LLVb] *LLVM Users*. http://llvm.org/Users.html.

[MSJ] Jeremy Maitin-Shepard and Daniel James: *Boost.Unordered C++-Bibliothek*. http://www.boost.org/doc/libs/1_42_0/doc/html/unordered.html.

[Pro] *Projects built with LLVM*. http://llvm.org/ProjectsWithLLVM.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck: *Global value numbers and redundant computations*. In *"POPL 1988"*, pages 12–27, 1988.

[SPEa] *SPEC CPU2000 Benchmark Suite*. http://www.spec.org/cpu2000.

[Speb] Reid Spencer: *The Often Misunderstood GEP Instruction*. http://llvm.org/docs/GetElementPtr.html.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler: *Documentation of the Intermediate Representation Firm*. Technical Report 1999-44, Universität Karlsruhe, Dezember 1999. http://digbib.ubka.uni-karlsruhe.de/volltexte/302599.