Bachelorarbeit

Ein Low Level Virtual Machine-Backend für Objective Caml

Universität Siegen

vorgelegt von

Gary Colin Benner

Prüfer

PD Dr. Kurt Sieber

Dipl.-Inform. Benedikt Meurer

Vorwort

Das Thema für diese Arbeit habe ich ausgewählt, da ich mich schon seit langem für Compiler, besonders deren Backends interessiere. Wesentlich für mein Interesse an LLVM war wohl David Tereis Arbeit über die Entwicklung eines LLVM-Backends für den Glasgow Haskell Compiler.

Ein neues Backend für den OCaml-Compiler zu schreiben war zudem naheliegend, da ich mich auch sehr für High-Level-Sprachen und funktionale Programmierung interessiere, auch wenn ich mich zuvor mehr mit Haskell beschäftigt habe.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Seit Grace Hopper (die den Begriff „compiler“ prägte) vor 60 Jahren (1951/52) den ersten Compiler für A-0 (Arithmatic Language, version 0), einen frühen Vorgänger von Cobol, schrieb, haben sich Compiler zu einem der zentralen Themen der Informatik entwickelt. Während sie anfangs ein Nischendasein fristeten und man Programme meist in Assembler-Sprachen schrieb, sind Compiler heutzutage nicht mehr wegzudenken. Das Lexen und das Parsen von Programmen ist schon lange gut verstanden und wird heute meist von Lexern und Parsern erledigt, die aus Grammatiken generiert werden.

Anders sieht es in der Regel bei der anschließenden Code-Erzeugung aus: Bei den meisten Compilern erfordert die Code-Erzeugung viel Handarbeit. Tools, die wie die Lexer- und Parser-Generatoren für viele verschiedene Programmiersprachen eingesetzt werden können, sind im Zusammenhang mit der Code-Erzeugung nicht besonders weit verbreitet. Ein solches Tool, das zusätzlich zur Code-Erzeugung auch sprach- und architekturunabhängige Optimierungen anbietet, ist die Low Level Virtual Machine (LLVM), die in dieser Arbeit eine wesentliche Rolle spielt.

Im Rahmen dieser Arbeit werde ich auf die Erzeugung von Maschinencode aus in Objective Caml (OCaml) geschriebenen Programmen unter Verwendung von LLVM eingehen. Dazu habe ich als Alternative zu dem bisherigen Backend des OCaml Native-Code-Compilers `ocamlopt` den Prototypen eines neuen Backends auf Basis von LLVM entwickelt. Dabei möchte ich untersuchen, wie gut sich LLVM für die Implementierung eines OCaml-Backends eignet und wie sich die Verwendung des LLVM-Frameworks auf die Komplexität des Backends und die Laufzeit des Compilers auswirkt.

1.1 Zielsprachen

Während die direkte Erzeugung von Assembler-Code durch dessen (fast) 1:1-Beziehung zum Maschinencode dem Compiler volle Kontrolle über die erzeugte Binärdatei erlaubt, ist es auch ein großer Aufwand den Assembler-Code zu generieren. Insbesondere gut optimierten Assembler-Code auszugeben war lange Zeit für Compiler nicht üblich und ist auch heute noch mit einem erheblichen Aufwand verbunden. Zudem müssen derartige Optimierungen für jede Architektur, für die der Compiler Binärprogramme erzeugen können soll, implementiert werden. Dadurch gibt es auch heute noch Compiler, die entweder nur wenige Architekturen unterstützen oder aber nur wenig optimieren.

Dies macht die Verwendung einer High-Level-Sprache, für die es einen portablen, gut optimierenden Compiler gibt, als Zielsprache des Compilers interessant. Man hat dadurch wenig Arbeit mit der Übersetzung von Programmen für verschiedene Architekturen und erhält viele Optimierungen ohne zusätzlichen Aufwand.

Eine andere Variante, die sich gerade in den letzten Jahren immer größerer Beliebtheit erfreut, ist die Verwendung von virtuellen Maschinen. Hierbei wird Maschinencode für einfach in Software realisierbare virtuelle Maschinen erzeugt, die diesen zur Laufzeit entweder interpretieren oder mit einem Just-in-time-Compiler in nativen Maschinencode übersetzen, wodurch man portable Binärdateien erhält. Dies sorgt zwar für eine erhöhte Portabilität und eine einmal entwickelte virtuelle Maschine kann zum Ausführen mehrerer Sprachen verwendet werden, man muss aber mit einem erhöhten Overhead zur Laufzeit leben.

Weiterhin ist es möglich, Code in einer High-Level-Assembler-Sprache zu generieren. Ganz ähnlich zur Verwendung einer virtuellen Maschine wird hier Code erzeugt, der nicht für die Ausführung auf realer Hardware gedacht ist. Anders als bei einer virtuellen Maschine ist es jedoch hier üblich, aus der High-Level-Assembler-Sprache mit Hilfe eines Ahead-of-time-Compilers nativen Code zu erzeugen, statt dies zur Laufzeit mittels eines JIT-Compilers zu tun.

Durch die Assembler-ähnliche Darstellung mit verhältnismäßig niedrigem Abstraktionslevel behält man dabei relativ viel Kontrolle über den erzeugten nativen Code, muss sich aber nicht gleich auf eine Architektur festlegen und kann bei Verwendung eines entsprechenden Compilerframeworks, wie etwa des hier verwendeten LLVM-Frameworks, vom Compiler unabhängig entwickelte Optimierungen und hardwarespezifische Backends verwenden.

2 Grundlagen

Zunächst ist ein Compiler ein Programm, das Programme in einer Quellsprache einliest und daraus ein Programm in einer Zielsprache ausgibt. Dabei lässt sich ein Compiler einteilen in ein Frontend, das für das Einlesen und Analysieren des Codes zuständig ist und ein Backend, das dann aus dem, was das Frontend berechnet hat, den zu generierenden Code ausgibt.[1]

Das Frontend führt dabei verschiedene Analysen aus: Die sogenannte *lexikalische Analyse*, bei der der Quelltext in eine Sequenz von Tokens zerlegt wird, die *syntaktische Analyse*, bei der aus dieser Tokensequenz ein Syntaxbaum des Programms erstellt wird und die *semantische Analyse*, bei der beispielsweise eine Typüberprüfung durchgeführt wird. Wenn all diese Analysephasen das Programm als gültig erkannt haben, wird der erstellte Syntaxbaum an das Backend weitergereicht.

Dort werden verschiedene Transformationen durchgeführt, an deren Ende der Zielcode ausgegeben wird. Dabei wird das Programm meist durch verschiedene Zwischenrepräsentationen dargestellt. Unterscheiden kann man hier wiederum verschiedene Phasen, zum einen die Zwischencode-Erzeugung, die maschinenunabhängigen Optimierungen, die Codeerzeugung und die maschinenabhängigen Optimierungen.

[1] Dies und das Folgende nach Aho (2007), S. 4 f.

2.1 OCaml

Caml ist eine seit 1985 vom „Institut national de recherche en informatique et en automatique“ (INRIA) entwickelte funktionale Programmiersprache.[2] Es gibt zwei wichtige Implementierungen von Caml: *Caml Light* und den Nachfolger *Objective Caml* (OCaml). OCaml ist die Hauptimplementierung und erweitert Camls Kernsprache um ein Objektsystem und Module.

OCaml bietet den High-Performance Native-Code-Compiler `ocamlopt` für neun Architekturen: IA32, PowerPC, AMD64, Alpha, Sparc, Mips, IA64, HPPA und StrongArm, sowie den sehr schnell arbeitenden Bytecode-Compiler `ocamlc`.[3]

Bisher besitzt dieser Native-Code-Compiler ein Backend, das plattformspezifischen Assembler-Code erzeugt. Zusätzlich habe ich im Rahmen dieser Bachlorarbeit ein neues Backend entwickelt, das LLVM zur Optimierung und Codeerzeugung nutzt.

2.1.1 Darstellung von Daten

Ganze Zahlen werden in OCaml im einfachsten Fall als sogenannte *tagged integers* dargestellt. Eine ganze Zahl, die ein Maschinenwort ausfüllt, unterscheidet sich von einem gleich langen Zeiger dadurch, dass ihr letztes Bit (das niedrigstwertige) stets 1 ist. Das kann bei Zeigern nämlich durch das Alignment der Daten auf Vielfache von 4 Byte oder 8 Byte nicht vorkommen, bei ihnen sind also wenigstens die letzten beiden Bits 0.

Durch diese Unterscheidung kann der Garbage Collector erkennen, ob es sich bei einem Wert um einen Zeiger handelt oder nicht. Auch für die Implementierung polymorpher Funktionen, die sowohl Zahlen als auch komplexere Daten als Argumente erhalten, ist dies sehr nützlich.

Komplexere Daten werden grundsätzlich auf dem Heap gespeichert und dargestellt als ein Header, der ein Maschinenwort lang ist, gefolgt von den eigentlichen Daten, wie in

[2] Dies und das Folgende von der Caml Homepage `http://caml.inria.fr/`
[3] OCaml Homepage `http://caml.inria.fr/ocaml/index.en.html`

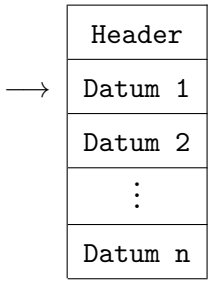

Abbildung 2.1: Darstellung von Daten auf dem Heap

Abbildung 2.1 dargestellt. Man bezeichnet diese Daten auch als *boxed*, da sie mit dem Header „verpackt“ werden.

Der Zeiger auf diese Daten zeigt dabei stets auf das erste Datenelement, da häufiger auf die Daten als auf den Header zugegriffen werden muss. Bei den Daten kann es sich um ganze Zahlen, Zeiger auf andere Daten, Gleitkommazahlen, Zeichenketten oder Kombinationen davon handeln.

Opfert man nicht das letzte Bit von Zahlen zur Unterscheidung von Zeigern müsste man auch Zahlen und nullstellige Konstruktoren boxed darstellen, was mit einem erhöhten Overhead einherginge. Zudem käme es zu einer Erhöhung des Speicherverbrauchs, da für jede ganze Zahl und jeden nullstelligen Konstruktor doppelt so viel Platz nötig wäre. Allerdings ist dafür die Integer-Arithmetik etwas aufwendiger und es geht die Möglichkeit verloren, Zahlen mit der Länge eines Maschinenwortes darzustellen. Möchte man so große Zahlen dennoch darstellen, so muss man diese als boxed darstellen.

2.1.2 Optimierungen

Für symbolischen Code und Integerarithmetik erzeugt das bisherige Native-Code-Backend sehr guten Maschinencode, aber nur dann, wenn der Quelltext gut geschrieben ist. Zwar kann der Compiler die primitiven Operationen in sehr effizienten Assembler-Code übersetzen, optimiert aber nur wenig. Durchgeführt werden etwa Constant Folding, also das Auswerten (einfacher) konstanter Ausdrücke zur Compile-Zeit, und einfache

Optimierungen von Additionen und Subtraktionen.

Beispielsweise können die Funktionen

```
let foo () = 123 * 2 - 200 + 7
let bar a b = a + b - a
```

zu

```
let foo () = 53
let bar a b = b
```

transformiert werden, darüber hinaus wird jedoch nicht viel optimiert.

Schon wenn etwa Multiplikation von Variablen mit Konstanten auftauchen, ist der Compiler nicht mehr in der Lage diese sinnvoll zu optimieren. So kann er schon

```
let foo n = 0 * n
```

nicht in die konstante Null-Funktion transformieren. Stattdessen wird Maschinencode generiert, der die Multiplikation mit der 0 ausführt.

Dies führt zwar dazu, dass erfahrene OCaml-Programmierer mit Wissen über den generierten Maschinencode zwar hochperformanten Code Schreiben können, doch es widerspricht ein Stück weit dem Konzept einer High-Level-Programmiersprache, dass man beim Programmieren Wissen über die zugrundeliegenden hardwarenahen Konzepte haben muss. Selbst Programmiersprachen mit einer niedrigeren Abstraktionsebene nehmen dem Programmierer manche dieser Optimierungen ab, etwa moderne C- oder Fortran-Compiler.

2.1.3 Aufbau des OCaml-Compilers

Das in Abbildung 2.2 gezeigte Schema veranschaulicht die verwendeten Darstellungen des Codes und die zwischendurch durchgeführten Transformationen.

Der OCaml-Compiler besteht aus einem Frontend mit Präprozessor, Lexer und Parser, die einen zunächst ungetypten abstrakten Syntaxbaum (engl.: „abstract syntax tree“,

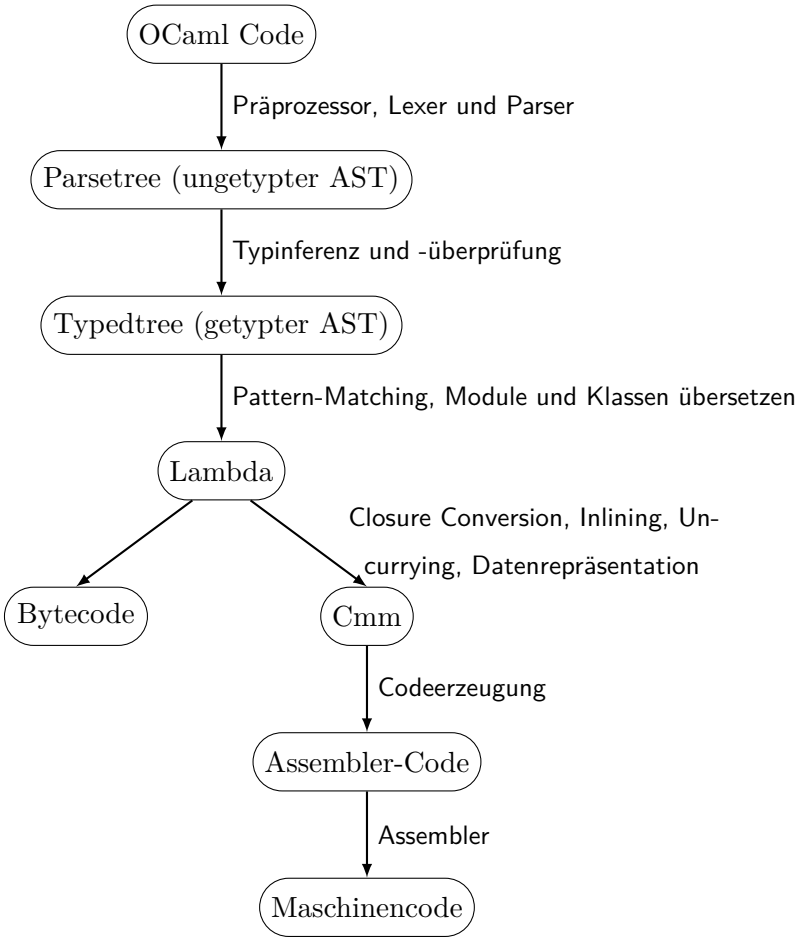

Abbildung 2.2: Aufbau des OCaml-Compilers [2]

AST) erzeugen, der mit Hilfe von Typinferenz in einen getypten AST überführt wird, wobei auch die Wohlgetyptheit des Programms überprüft wird.

Von diesem AST ausgehend übersetzt das Backend Module, Klassen und Pattern Matching in einfachere Konstrukte (etwa switch und if-then-else) und erzeugt daraus die Lambda-Darstellung. Dabei werden Programme auf einem noch recht hohen Level mit vielen komplexen Konstrukten (beispielsweise für for-Schleifen) repräsentiert.

An dieser Stelle geht es für den Bytecode- und den Native-Code-Compiler unterschiedlich weiter. Beim Bytecode-Compiler wird im Wesentlichen der Lambda-Code in einer kompakten Form, die als Bytecode dient, gespeichert, während beim Native-Code-Compiler

noch einige weitere Transformationen durchgeführt werden müssen.

Als erstes werden Closures und Currying aus dem Lambda-Code in die einfacheren Konstrukte von Cmm, einer Variante von C--, übersetzt, das Inlining von Funktionen durchgeführt – sofern dies sinnvoll erscheint – und Literale aus dem Code extrahiert und explizit als globale Daten modelliert. Aus diesem Code wird dann über weitere Zwischendarstellungen, bei deren Erzeugung schon architekturabhängige Entscheidungen einfließen, in Assembler-Code für die Zielarchitektur übersetzt. Dieser wird auf die übliche Weise mit einem Assembler in Maschinencode umgewandelt.

2.1.4 Cmm-Konstrukte

Im Folgenden werde ich näher auf Cmm eingehen, da das neue Backend nach der Erzeugung des Cmm-Codes ansetzt und damit Cmm in LLVM-IR[4] übersetzen muss.

Cmm beinhaltet ein sehr einfaches Typsystem, das nur wenige Typen unterscheidet. Lokale Variablen und alle Zwischenergebnisse sind entweder eine Adresse, eine ganze Zahl oder eine Gleitkommazahl. Nur zum Laden und Speichern stehen mehr Typen zur Verfügung, die sich aber von den eben genannten Typen nur dadurch unterscheiden, dass sie andere Längen haben können. Dies wird genutzt, um beispielsweise das Laden eines einzelnen Bytes oder einer Gleitkommazahl mit einfacher Präzision zu ermöglichen. Sofort nach dem Laden wird aber der Wert in ein Maschinenwort beziehungsweise eine Gleitkommazahl mit doppelter Präzision umgewandelt. Die entsprechende Umwandlung erfolgt beim Speichern in umgekehrter Richtung.

Anders als C-- wird Cmm nicht als eine Variante von C mit C-ähnlicher Syntax dargestellt, sondern mit einer sehr viel einfacheren, S-Expression-basierten, Lisp-ähnlichen Syntax, bei der der sequentielle Aspekt des Programms, den man bei C-- leicht sehen kann, in den Hintergrund tritt.

[4] LLVMs Assembler-Sprache (Low Level Virtual Machine intermediate representation)

Arithmetische Operatoren

Es gibt die üblichen arithmetischen Operatoren `+`, `-`, `*`, `/` für ganze und Gleitkommazahlen, Addition und Subtraktion von Zeigern, Negation und Betrag von Gleitkommazahlen und einen Modulo-Operator für ganze Zahlen.

Zudem sind die üblichen bitweisen Operatoren für eine Verschiebung nach links, eine vorzeichenbehaftete und eine vorzeichenlose Verschiebung nach rechts, sowie die booleschen Operatoren UND, ODER und XOR als bitweise Operatoren vorhanden.

Vergleichsoperatoren

Zum Vergleich zweier Werte gibt es die üblichen Operatoren: `==`, `!=`, `<`, `<=`, `>` und `>=`. Diese sind in Varianten für ganze Zahlen (vorzeichenbehaftet), Zeiger (vorzeichenlos) und Gleitkommazahlen vorhanden. Die Vergleichsoperatoren liefern entweder wahr (in Form einer beliebigen von Null verschiedenen ganzen Zahl) oder falsch (Null) zurück.

Kontrollfluss

Es gibt einen `if`-Ausdruck, der als Bedingung eine beliebige ganze Zahl erwartet, die als wahr interpretiert wird, wenn sie von Null verschieden ist.

Auch ein `switch`-Ausdruck, den man als Verallgemeinerung des `if`-Ausdrucks auf mehr Fälle betrachten kann, ist vorhanden. Dieser wird nur zur Darstellung des Pattern Matchings verwendet.

Zur Darstellung von Schliefen gibt es das `loop`-Konstrukt, das sein Argument in einer Endlosschleife ausführt, wenn nicht der Kontrollfluss die Schliefe mit dem `catch`/`exit`-Konstrukt verlässt.

Dieses besteht aus einem `catch`-Ausdruck, der es erlaubt aus seinem Argument mit Hilfe eines `exit`-Ausdrucks zu einem dazugehörigen, im `catch`-Ausdruck enthaltenen Handler

zu springen. Im Wesentlichen handelt es sich hierbei um eine garantiert lokale Variante einer goto-Anweisung mit einem zugehörigen Label.

Exception Handling

Exceptions können mit `raise` geworfen und mit Hilfe eines `try`-Ausdrucks gefangen werden. `try`/`raise` sind quasi eine nicht-lokale Variante von `catch`/`exit`, das heißt `raise` darf, anders als `catch` innerhalb einer im Inneren von `try` aufgerufenen Funktion ausgeführt werden. Weiterhin darf `raise` auch außerhalb des Gültigkeitsbereichs eines `catch`-Ausdrucks verwendet werden. In dem Fall wird die Exception nicht gefangen und das Programm beendet.

Sonstiges

Weiterhin gibt es Operationen, um ganze Zahlen in Gleitkommazahlen umzuwandeln und umgekehrt, sowie Operationen zum Lesen und Schreiben von Daten. Mit `let` können lokale Variablen erzeugt werden, mit `assign` kann solchen Variablen ein neuer Wert zugewiesen werden.

Zudem erfordert es die Baumstruktur des Cmm-Codes, einen Ausdruck `seq` für die Hintereinanderausführung zweier Ausdrücke zu haben.

Zur Allokation und Initialisierung von Speicher auf dem Heap dient `alloc`, mit `apply` lassen sich OCaml-Funktionen, mit `extcall` Funktionen mit C Calling Convention aufrufen.

`checkbound` dient zur Überprüfung, ob ein Zugriff auf ein Arrayelement oder ein einzelnes Zeichen einer Zeichenkette tatsächlich innerhalb der Grenzen dieser Datenstruktur erfolgt, andernfalls wird eine Exception geworfen.

2.2 Low Level Virtual Machine

Das Low Level Virtual Machine-Projekt ist ein Open Source-Projekt, das ein optimierendes Compiler-Framework entwickelt. Die Arbeit an LLVM begann 2000 im Rahmen der Master-Arbeit von Chris Arthur Lattner, der heute einer der Hauptentwickler von LLVM ist.[5]

LLVM stellt einen quell- und zielunabhängigen Optimierer zur Verfügung, zusammen mit der auch als LLVM-IR bezeichneten Zwischensprache, auf der dieser operiert. LLVM-IR-Programme können abgesehen von der assemblerähnlichen textuellen Darstellung auch als eine Art Bytecode dargestellt werden, der sich aufgrund der sehr regelmäßigeren Struktur sehr viel schneller parsen lässt; ein Just-in-time-Compiler, der LLVM-IR ausführt und einen Ahead-of-time-Compiler, der LLVM-IR in nativen Assembler-Code übersetzt.

Für die Codeerzeugung ist LLVM aus mehreren Gründen ein attraktives Ziel. Zum einen erlaubt die sehr freie, BSD-Lizenz-ähnliche, Illinois Open Source License die kostenlose Verwendung, Weiterentwicklung und Verbreitung von LLVM ohne nennenswerte Einschränkungen, zum anderen ist LLVM ein moderner Code-Generator mit Unterstützung für elf Architekturen. Die zur Verfügung stehenden Optimierungen sorgen dafür, dass dabei recht guter Code generiert wird. Dabei ist LLVM (mit gewissen Lücken) gut dokumentiert und wird aktiv weiterentwickelt. Interessant ist zudem die gute Unterstützung von Vektoroperationen und deren Umsetzung auf Hardware-Vektoreinheiten.[6]

Zur Zeit unterstützt LLVM elf verschiedene Architekturen (ARM, CellSPU, Hexagon, MBlaze, MSP430, Mips, PTX, PowerPC, Sparc, X86 und XCore).[7] LLVM wird hauptsächlich von Apple finanziert und in verschiedenen Bereichen eingesetzt, so unterstützt Apples Entwicklungsumgebung Xcode ab Version 4.2 nicht mehr GCC, sondern nur noch die LLVM-basierten Compiler Clang und LLVM-GCC, eine Variante von GCC, die für

[5] vgl. LLVM-Homepage `http://www.llvm.org` (15. 3. 2012). Dieser Abschnitt basiert auf dem Abschnitt über LLVM in Terei 2009, S. 18ff

[6] vgl. Terei 2009, S. 17f

[7] vgl. `http://llvm.org/docs/CodeGenerator.html#targetimpls` (15. 3. 2012)

die Optimierung und Codeerzeugung LLVM verwendet wird.[8] Man kann also durchaus sagen, dass LLVM sich im Produktiveinsatz befindet.

2.2.1 LLVM-IR

Sowohl als Eingabesprache, als auch als interne Zwischenrepräsentation wird LLVM-IR eingesetzt (sowohl in der menschenlesbaren, als auch einer binären Form). Die Sprache muss dabei zum einen ein ausreichend niedriges Level haben, um sinnvoll für die Codeerzeugung von Compilern verschiedener Sprachen genutzt werden zu können und diesem Compiler ein gewisses Maß an Kontrolle über den genierten Maschinencode bieten, zum anderen muss das Level hoch genug sein, um sinnvolle Aussagen über das Programm treffen und auf diesen basierend Optimierungen durchführen zu können.

Von üblichen (Low-Level-)Assembler-Sprachen unterschiedet sich LLVM-IR deshalb in mehreren Punkten.

Stack Anders als bei Assembler-Sprachen üblich, verwaltet LLVM den Stack selbst und ermöglicht dem Nutzer nur sehr eingeschränkten Zugriff. So ist es zwar möglich, Variablen auf dem Stack anzulegen, Funktionen aufzurufen und Exceptions zu behandeln, wobei jeweils der Stack verändert wird. Auf die genauen Änderungen hat der Nutzer aber keinen Einfluss. Man kann weder entscheiden, in welcher Reihenfolge die Daten auf den Stack gelegt werden, noch an welcher Stelle im Code dies genau passiert.

Typen Anders als bei Low-Level-Assemblern üblich ist LLVM (statisch) typisiert. Dies erlaubt einerseits – etwa durch ganzzahlige Typen verschiedener Länge – die weitgehende Plattformunabhängigkeit von LLVM, andererseits aber auch bessere Analyse- und damit auch Optimierungsmöglichkeiten. Durch das Typsystem können Informationen aus der Quellsprache erhalten bleiben, die bei einem Low-Level-Assembler verloren gingen, die jedoch für gewisse Optimierungen wichtig sind.

[8] vgl. `http://llvm.org/Users.html` (15. 3. 2012)

LLVM verfügt über ein – nicht nur für Assembler-Sprachen – ungewöhnlich detailliertes Typsystem. Als Basistypen umfasst es den leeren Typen `void`, ganzzahlige Typen `i1`, `i2`, `i3`, ..., `i64`, ... mit beliebiger Länge sowie verschiedene Gleitkommaformate, unter anderem die üblichen IEEE 754 single precision (`float`) und double precision (`double`) Formate.

Aus diesen Basistypen können Zeiger (etwa `i64*` als Zeiger auf `i64`), Strukturen (`{i64, double}` als Paar aus `i64` und `double`), Arrays (`[42 x i64]` als Array aus 42 aufeinanderfolgendenen Elementen vom Typ `i64`) und Funktionen (`i64(double, float)` für eine Funktion, die aus einem `double` und einem `float` einen `i64` macht) als abgeleitete Typen definiert werden.

Durch die Verwendung des Typsystems ist es, wenn im Backend nicht mehr alle Typinformationen vorliegen, nötig, das LLVM-Programm mit expliziten Typecasts zu übersäen, die aber später bei der Codeerzeugung ignoriert werden.

Funktionen Einer der offensichtlichsten Unterschiede ist vermutlich die Tatsche, dass sämtlicher Code in LLVM in Funktionen organisiert ist, die wiederum aus Basisblöcken aufgebaut sind. Sprünge können nur innerhalb einer Funktion an den Anfang eines Basisblocks gehen. Dies macht es für LLVM sehr viel einfacher, Aussagen über den Code zu treffen, etwa bei welchem Code es sich um toten Code handelt.

SSA LLVM verwendet die sogenannte *single static assignment*-Form (SSA). Hierbei werden statt realen Hardware-Registern virtuelle Register verwendet, von denen jedes nur einmal geschrieben werden kann. Möchte man einen geänderten Wert, etwa das Ergebnis einer Addition zweier Register speichern, so wird dafür ein neues Register angelegt. Um dennoch sinnvoll mit Registern arbeiten zu können, stehen unendlich viele virtuelle Register als Abstraktion der Hardwareregister zur Verfügung.

Nun kann es nötig sein, in einem Register Ergebnisse aus zwei verschiedenen Basisblöcken zu speichern, von denen einer vorher durchlaufen wurde. Dies ist zum Beispiel wichtig,

um das Ergebnis eines if-then-else-Ausdrucks zu erhalten, wenn im then- und im else-Block jeweils ein Ergebnis in ein Register geschrieben wurde. Dazu gibt es bei SSA die sogenannten Φ-Knoten, bei LLVM dargestellt durch `phi`-Instruktionen, die es erlauben einem Register verschiedene Werte zuzuweisen, abhängig davon, aus welchem Basisblock der Kontrollfluss den aktuellen Block erreicht hat.

Dadurch enthält die Darstellung eines Programms in LLVM-IR (von Zugriffen auf den Speicher abgesehen) den Graphen der Datenabhängigkeiten des Programms. Dies macht viele Optimierungen einfach, für die ohne das explizite Vorhandensein dieser Informationen sehr viel aufwendigere Analysen nötig wären. Auch andere Compiler setzen deshalb SSA als eine der verwendeten Zwischendarstellungen ein, etwa der weit verbreitete GCC.[9]

Besonders zur Darstellung von funktionalen Programmen eignet sich die SSA-Form ausgesprochen gut, da hier Daten normalerweise einmal berechnet und danach nicht mehr verändert werden. Bei der Verwendung von LLVM treten dabei jedoch gewisse Probleme auf, die jedoch nicht SSA-inhärent sind, worauf ich in Kapitel 3.3.1 näher eingehen werde.

2.2.2 LLVMs Instruktionssatz

Im Folgenden möchte ich ein wenig auf die Instruktionen in LLVM-IR eingehen, die das im Rahmen dieser Arbeit entstandene Backend verwendet.

Kontrollfluss

Wie schon erwähnt sind Funktionen in LLVM aus Basisblöcken zusammengesetzt.[10] Diese beginnen mit einem Label (das am Anfang der Funktion optional ist) und enden mit einer der folgenden Kontrollfluss-Instruktionen.

`br` Bedingtes oder unbedingtes Springen zum Anfang eines Basisblocks

[9] vgl. GCC Dokumentation: `http://gcc.gnu.org/onlinedocs/gccint/SSA.html` (17. 3. 2012)

[10] Dieser Abschnitt basiert auf Terei (2009), S.21ff; vgl. auch `http://llvm.org/docs/LangRef.html` (20. 3. 2012)

`switch` Wie bedingtes `br` aber mit mehr als zwei möglichen Zielblöcken.

`ret` Aus Funktion zurückkehren und eventuell einen Wert zurückgeben.

Ein Basisblock darf nur dann mit einer anderen Instruktion enden, wenn das Ende des Blocks vom Kontrollfluss im Programm niemals erreicht werden kann, was man durch Verwendung der Instruktion `unreachable` ausdrücken kann. Dies ist beispielsweise sinnvoll, wenn eine Exception in diesem Block geworfen wird und die Ausführung dadurch das Ende des Blocks nicht erreichen kann.

Arithmetische Operationen

Als Arithmetische Operationen bietet LLVM die üblichen Operationen zur Addition, Subtraktion, Multiplikation, Division und Modulo, die ersten drei in einer ganzzahligen und einer Gleitkomma-Variante, die letzten beiden jeweils in einer vorzeichenbehafteten, einer vorzeichenlosen und einer Variante für Gleitkommazahlen. Sie erwarten zwei Operanden des gleichen Typs und geben einen Wert dieses Types zurück.

`add, sub, mul` implementieren Addition, Subtraktion und Multiplikation, sowohl von ganzen Zahlen, als auch von Gleitkomma-Zahlen.

`sdiv, fdiv` stellen vorzeichenbehaftete Division ganzer Zahlen, sowie die Division von Gleitkommazahlen dar.

`srem` gibt den Rest der vorzeichenbehafteten Division an.

Bitweise Operationen

Die folgenden bitweisen Operatoren operieren auf ganzen Zahlen. Auch hier müssen beide Argumente denselben Typ haben.

`shl, lshr, ashr` führen Bit-Schiebe-Operationen nach links, logisch und arithmetisch nach rechts (d.h. mit Nullen bzw. mit Vorzeichenbit auffüllen).

`and, or, xor` führen die UND-, ODER- und XOR-Operationen bitweise durch.

Speicherzugriff

Für den Zugriff auf Daten im Speicher bietet LLVM im Wesentlichen die folgenden Instruktionen:

`alloca` alloziert Speicher auf dem Stack.

`load, store` lädt bzw. speichert Daten an einer angegebenen Adresse.

`getelementptr` implementiert Zeigerarithmetik für den Zugriff auf Unterelemente einer zusammengesetzten Datenstruktur oder, wenn das erste Argument ein Zeiger ist, schlicht die Array-Index-Operation, wie man sie aus C kennt.

Konvertierungsoperationen

Durch LLVMs Typsystem ist es an vielen Stellen nötig, explizite Typecasts vorzunehmen, um beispielsweise bitweise Arithmetik auf Gleitkommazahlen zu erlauben, da LLVM keine impliziten Casts durchführt und die bitweisen Operationen nur für ganzzahlige Daten erlaubt sind. Im resultierenden Maschinencode sind diese Konvertierungen zwar nicht mehr vorhanden, sind aber für die Analyse wichtig.

Andere Operationen

`icmp, fcmp` stellen die üblichen Vergleichsoperationen für ganze und Gleitkommazahlen zur Verfügung. Als Ergebnis liefern sie einen Wert vom Typ `i1`, das heißt entweder `true`, oder `false`. Das Ergebnis kann anschließend für die Auswahl des Ziels bei einer bedingten Sprunginstruktion verwendet werden.

`phi` implementiert die Φ-Operation, wählt also abhängig vom Vorgänger-Block einen Wert als Ergebnis aus.

`call` stellt einen Funktionsaufruf mit einer bestimmten Calling Convention dar. Stimmt diese nicht mit der Calling Convention, mit der die Funktion deklariert wurde, überein, so ist das Ergebnis undefiniert. Mit geeigneter Calling Convention kann man LLVM durch Voranstellen des Schlüsselwortes `tail` dazu auffordern, eine tail-call-Optimierungen durchzuführen, sofern dies an dieser Stelle möglich ist.

2.2.3 Exception Handling

Grundsätzlich bietet LLVM zwei verschiedene Varianten des Exception Handlings an.[11] Zum einen gibt es eine gute Unterstützung für Zero Cost Exception Handling, wie es bei C++ verwendet wird, zum anderen eine Variante des setjmp/longjmp-Exception Handlings mit deutlich weniger Overhead, als bei Verwendung der setjmp/longjmp-Funktionen aus der (g)libc.

Dabei ist zu beachten, dass Zero Cost Exceptions keineswegs kostengünstige Operationen sind. Der Name bezieht sich nur darauf, dass der Code, sofern keine Exceptions geworfen werden, genauso schnell ist, als würde man vollständig auf Exception Handling-Konstrukte verzichten. Wird aber eine Exception geworfen, so bedeutet dies einen deutlich höheren Aufwand, als beispielsweise bei der Verwendung von setjmp/longjmp.

Bei setjmp/longjmp erkauft man höhere Geschwindigkeit beim Werfen und Fangen von Exceptions dadurch, dass man einen gewissen Overhead gegenüber einem Exception-freien Programm hat, auch dann, wenn nie eine Exception geworfen wird.

LLVMs eingebaute setjmp/longjmp-Implementierung ist verglichen mit der Implementierung aus der libc schneller und sparsamer beim Speicherverbrauch. Es werden nur die Callee-Save-Register und der Stackpointer gesichert, was auf AMD64 insgesamt fünf Maschinenwörter sind, während bei der Verwendung der (g)libc schon 25 Maschinenworte gesichert werden müssen. Dies erfordert sowohl mehr Speicher, als auch mehr Zeit.

Was aus der LLVM-Dokumentation nicht direkt hervorgeht ist, dass setjmp/ longjmp

[11] vgl. zu diesem Abschnitt `http://llvm.org/docs/ExceptionHandling.html` (19. 3. 2012)

nicht auf allen Plattformen, sondern nur auf arm/darwin implementiert ist.[12] Auf anderen Plattformen werden die setjmp- und longjmp-Aufrufe schlicht ignoriert, was zu fehlerhaftem Maschinencode führt.[13]

Man muss daher auf anderen Plattformen entweder C++-Exception Handling, setjmp/longjmp aus der libc oder ein komplett eigenes Verfahren für das Exception Handling verwenden.

2.3 Vergleich von Cmm und LLVM-IR

Während C-- und LLVM-IR große Ähnlichkeiten haben, nicht zuletzt dadurch, dass beide das gleiche Problem zu lösen versuchen, sind die Unterschiede von Cmm zu LLVM-IR deutlich größer.

Das beginnt schon damit, dass Cmm kaum lokale Variablen einsetzt. Zu übergebende Argumente werden nicht explizit in einer Variablen oder einem Register gespeichert, sondern der Ausdruck, der übergeben werden soll, direkt als Argument hingeschrieben. Durch die verwendete S-Expression-basierte Syntax, wie sie auch in Lisp zum Einsatz kommt, ist das Programm weniger eine Aneinanderreihung von separaten Instruktionen, als ein Baum von Teilausdrücken.

Möchte man also Cmm in LLVM-IR übersetzen, so ist es zunächst nötig, diese Baumstruktur in eine linearisierte Form, dass heißt eine Sequenz von Instruktionen umzuwandeln. Dabei müssen virtuelle Register für sämtliche Zwischenergebnisse benutzt werden.

Auch bietet Cmm ein explizites Schliefen-Konstrukt, das in LLVM-IR durch Labels und Sprünge ausgedrückt werden muss. Das Gleiche gilt für Cmms `if`- und `switch`-Ausdrücke.

Ein weiteres Hindernis besteht darin, dass Cmm nur an wenigen Stellen Typen explizit angibt, während in LLVM kaum eine Instruktion ohne eine explizite Typangabe auskommt.

[12] Bugreport: setjmp/longjmp-based exception handling, `http://llvm.org/bugs/show_bug.cgi?id=1642` (7. 3. 2012)

[13] Das Problem tritt auch mit anderen Compilern, etwa dem LLVM basierten C/C++/Objective-C-Compiler Clang bei Verwendung von `__builtin_setjmp` bzw. `__builtin_longjmp` auf.

Cmms Typsystem ist zwar sehr einfach, verursacht aber dennoch einen gewissen Aufwand bei der Typinferenz, da keine expliziten Casts vorhanden sind und dieselbe lokale Variable mal als Adresse und mal als Zahl benutzt werden kann. Insbesondere erhalten Funktionen nur Adressen als Argumente und geben immer eine Adresse zurück, auch dann, wenn es sich eigentlich bei den übergebenen Werten und dem Ergebnis um Zahlen handelt.

Dadurch ist es aber so, dass man auf die wirklichen Typen im Cmm-Code nur beim Laden, Speichern und der Speicherallokation achten muss, an allen anderen Stellen kann man bedenkenlos Register auf den erwarteten Typ casten.

Weiterhin verwendet LLVM Labels an vielen Stellen, während sie in Cmm nur mittels `catch` (und `try`) erzeugt werden können und dabei auch nur lokal zur Verfügung stehen. Insbesondere ist es mit diesen Konstrukten nicht möglich eine Schliefe zu konstruieren, hierzu muss man in Cmm `loop` heranziehen. Für die Übersetzung von Cmm in LLVM-IR ist dieser Unterschied natürlich kein Problem.

2.4 Ähnliche Projekte

In den letzten Jahren erfreut sich LLVM für die Code-Generierung großer Beliebtheit. Verwendet wird LLVM unter anderem von

- Clang, einem schnellen, gut optimierenden C/C++/Objective-C-Compiler, der komplett auf LLVM für die Optimierung und Codeerzeugung setzt und für seine ausgesprochen guten Fehlermeldungen bekannt ist,[14]
- DragonEgg, einem Plugin für GCC ab Version 4.5, das die gesamte Optimierung und Codeerzeugung von GCC durch LLVM ersetzt,[15]
- Unladen Swallow, einem LLVM-basierten JIT-Compiler für Python,[16]

[14] vgl. Clang-Homepage, `http://clang.llvm.org/`
[15] vgl. DragonEgg-Homepage, `http://dragonegg.llvm.org/`
[16] vgl. Projects built with LLVM, `http://llvm.org/ProjectsWithLLVM/`

- PyPy, einem Framework zur Generierung von JIT-Compilern für dynamische Sprachen, in erster Linie bekannt als experimentelle Python-Implementierung[16] und
- einem Backend von GHC, einem modernen, optimierenden Haskell-Compiler.[17]

[17] vgl. GHC-Homepage, `http://www.haskell.org/ghc/`

3 Das LLVM Backend

Das im Rahmen dieser Arbeit entwickelte Backend setzt zur Codeerzeugung nach der Erzeugung des Cmm-Codes an und erzeugt aus diesem mit Hilfe von LLVM (AMD64-)Assembler, statt, wie beim bisherigen Native-Code-Backend, über einige OCaml-interne Zwischenrepräsentationen selbst Assembler-Code auszugeben. Die vereinfachend als ein Schritt dargestellte Übersetzung von Cmm-Code in Assembler-Code aus Abbildung 2.2 wird dabei durch die in Abbildung 3.1 dargestellte Pipeline ersetzt. Dabei habe ich mich auf die Umsetzung für die AMD64-Architektur beschränkt, da es sich um eine der am weitesten verbreiteten Architekturen handelt und mir entsprechende Hardware zur Verfügung steht.

Aus verschiedenen, später erwähnten Gründen ist das von mir entwickelte Backend weder mit der Laufzeitumgebung, noch mit Binärdateien, die mit Hilfe von `ocamlopt` übersetzt wurden, kompatibel. So wird statt OCamls eigener Calling Convention (die ersten zehn ganzzahligen Parameter werden in `rax`, `rbx`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11` und das Ergebnis in `rax` übergeben; es gibt keine Callee-Save-Register) die in der System V ABI für AMD64 vorgesehene Calling Convention[18] (die ersten sechs Parameter in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, Ergebnis in `rax`, Callee-Save-Register `rbx`, `rbp`, `r12`–`r15`) verwendet.

[18] System V ABI, S. 21, `http://x86-64.org/documentation/abi.pdf`, (21. 3. 2012)

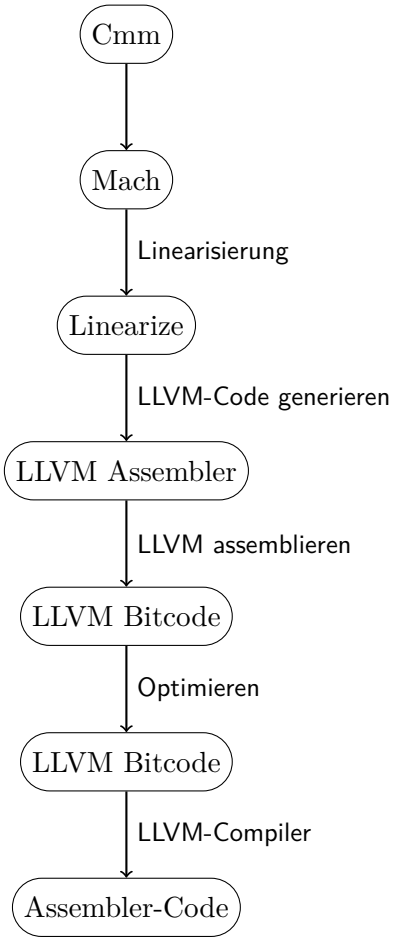

Abbildung 3.1: Erzeugung des Assembler-Codes

3.1 Verwendete Zwischenrepräsentationen

Im Laufe der Entwicklung haben sich die verwendeten Zwischenrepräsentationen drastisch verändert. Im Folgenden werde ich daher darauf eingehen, was die verschiedenen Stadien waren und wie sich die Veränderungen dazwischen ergeben haben.

3.1.1 Der erste Versuch: keine Zwischendarstellung

Anfangs wurde aus dem Cmm-Code für eine Funktion direkt LLVM-IR generiert. Dazu wurde die gesamte Baumstruktur des Cmm-Codes von einer Funktion rekursiv durchlaufen,

die ein Tripel als Ergebnis hatte. Die erste Komponente enthielt den auszugebenden LLVM-IR-Code, sofern ein solcher vorhanden war, ansonsten einen leeren String. Die zweite Komponente vom Type `string option` enthielt, falls vorhanden, den Namen des Registers, in dem das Ergebnis der auszugebenden Instruktionen zu finden war. Die dritte Komponente mit demselben Typ enthielt, wenn es ein solches Register gab, dessen Typ, dargestellt durch die Zeichenkette, die diesen Typen im LLVM-IR repräsentiert.

Dieses Register wurde, sofern vorhanden, als Argument für die Cmm-Anweisung in der nächstniedrigeren Rekursionstiefe verwendet. Da die zweite Komponente jedoch auch `None` sein konnte, war dazu viel Code zur Überprüfung dieser Komponente nötig, der in leichten Variationen vielfach im Programm auftauchte.

Am Ende wurden so alle Instruktionen zu einem einzigen String konkateniert, der die gerade übersetzte Cmm-Funktion in LLVM-IR darstellte. Dieser wurde dann in eine Datei geschrieben.

Diese Vorgehensweise ist zwar bei den einfachen Cmm-Konstrukten, wie beispielsweise den arithmetischen Operationen, durchaus sinnvoll einsetzbar, doch spätestens bei komplexeren Konstrukten, wie if-then-else- oder catch-Ausdrücken ausgesprochen unübersichtlich und damit fehleranfällig. Zudem wurde OCamls Fähigkeit, viele Fehler durch das Typsystem zu verhindern, durch die konsequente Nutzung von Strings weitestgehend umgangen.

Auch war diese Art der Übersetzung absolut nicht dazu geeignet, nicht-triviale Transformationen am Code durchzuführen, die sich später für die Garbage Collection als nötig erwiesen.

Es war daher sehr schnell klar, dass in irgendeiner Form eine Zwischenrepräsentation für den Code nötig sein würde. Allein die vielen Konkatenationen machten das Programm, trotz des zu diesem Zeitpunkt geringen Umfangs von nur wenigen hundert Zeilen, extrem unübersichtlich.

3.1.2 Erste Zwischenrepräsentation

Deshalb wurde diese Variante der Übersetzung durch eine neue ersetzt, die eine Cmm-ähnliche Baumstruktur zur Darstellung des auszugebenden LLVM-Codes nutzte. Damit wurde die Ausgabe des LLVM-IR-Codes von der Übersetzung des Cmm-Codes getrennt, was schon für besser lesbaren Code sorgte.

Die verwendete Baumstruktur nutzte, wie Cmm, keine expliziten Register, was in Fällen, in denen ein Wert an zwei Stellen nötig war, dazu führte, dass Code dupliziert wurde oder explizit eine Variable angelegt werden musste, in der der berechnete Wert gespeichert wurde und, wenn nötig, wieder geladen werden konnte. Die Register wurden erst bei der Ausgabe des LLVM-IR-Codes erzeugt und verwendet.

Insbesondere behindert die Baumstruktur die Ausnutzung der natürlichen Struktur eines freien Monoids der Konkatenation von Strings, oder hier der Aneinanderreihung von Instruktionen. Die Baumstruktur erzwingt hier eine Klammerung von aufeinander folgenden Instruktionen, obwohl diese Klammerung eigentlich vollkommen irrelevant ist.

Diese Darstellung war gewissermaßen die schlechtestmögliche Kombination von Eigenschaften des Cmm- und LLVM-IR-Codes. Einerseits waren die dargestellten Instruktionen direkt in LLVM-Instruktionen übersetzbar, was die Übersetzung von Cmm in diese Darstellung nicht gerade erleichterte, andererseits hatte der Code eine für LLVM-Code, der im Wesentlichen eine Aneinanderreihung einzelner Instruktionen ist, völlig unpassende Baumstruktur, die besser zu einer Cmm-nahen Darstellung gepasst hätte.

Um dieses Problem zu beheben, habe ich anschließend diese Zwischenrepräsentation durch zwei neue ersetzt, die beide aufeinander folgende Instruktionen mit Hilfe einer einfach verknüpften Liste darstellen, wobei die eine noch ein high-level-if-then-else-, switch- und loop- Statement enthält, also noch recht Cmm-nah ist (bis auf die Übersetzung von Teilbäumen zur Parameterübergabe in explizite Parameterübergabe mit Registern), während die andere diese Konstrukte durch Labels und bedingte und unbedingte Sprünge ersetzt, und in ihrer Struktur sehr nah an LLVM-IR ist.

3.1.3 Mach

Bei *Mach* handelt es sich um eine Darstellung, die stark an `ocamlopt`s „Mach“ angelehnt ist. Es bleiben gewisse high-level-Konstrukte aus dem Cmm-Code erhalten, wie etwa if-, switch- oder loop-Statements, während der Code von einer allgemeinen Baumform in Teilstücke mit explizit verwalteten Registern und verknüpften Listen für die Aneinanderreihung der Instruktionen transformiert wird. Diese Zwischendarstellung wird eher dazu verwendet, die einzelnen Übersetzungsschritte der Übersichtlichkeit halber klein zu halten, könnte aber auch für eventuelle zukünftige OCaml-spezifische Optimierungen, die LLVM nicht durchführen kann, nützlich sein.

Eine einzelne Instruktion wird dargestellt durch einen Record aus

- einer Beschreibung der Instruktion, etwa um angeben zu können, dass es sich um einen if-then-else-Ausdruck oder den Aufruf einer Funktion handelt. Diese Beschreibung kann recht viele Informationen enthalten. Bei Funktionsaufrufen wird hier ein Register, das einen Zeiger auf die Funktion enthält, gespeichert, beim if-then-else-Statement wird das Paar aus then- und else-Block (als Mach-Instruktionen) hier gespeichert.
- einem Array von Parametern, die diese Instruktion entgegennimmt, bei if-then-else beispielsweise die Bedingung, bei Funktionsaufrufen die Parameter der Funktion.
- einem optionalen Ergebnisregister. Es ist vorhanden, wenn die Instruktion ein Ergebnis zurückgibt oder am Ende eines Blocks steht, der ein Ergebnis zurückgibt, beispielsweise in einem Zweig eines if-then-else-Statements. So kann man leicht herausfinden, was oder ob ein Block etwas zurückgibt.
- Debugging-Informationen, die einfach nur weitergereicht werden.
- einer Referenz auf die nächste Instruktion. Hiermit wird die Listenstruktur von Anweisungsfolgen modelliert.
- einem Typ, der je nach Instruktion verschiedene Bedeutungen hat. Bei den arith-

metischen Operationen handelt es sich um den Typ, den die Argumente und damit das Ergebnis haben sollen, bei Vergleichsoperationen nur um den Argumenttyp, bei store-Instruktionen um den Typ der zu speichernden Daten.

Die explizite Speicherung eines Typen, der weder Argument- noch Rückgabetyp sein muss, ist nötig, weil in Mach noch keine Typecasts dargestellt werden können. Mit der Komponente für den Typ einer Instruktion kann man aber den späteren Pipelinestufen die nötigen Informationen für die Casts mitteilen.

3.1.4 Linearize

In der *Linearize* Darstellung werden, wie in `ocamlopts` „`Linearize`“, die Instruktionen einer Funktion in einer verknüpfen Liste dargestellt, ohne dass es Instruktionen mit untergeordneten Listen von Instruktionen wie bei Mach gibt. Die höheren Konstrukte aus Mach werden linearisiert, das heißt Instruktionen wie if-then-else, switch oder loop werden durch Labels und Sprünge in einer einzigen verlinkten Instruktionsliste dargestellt.

Die einzelnen Instruktionen werden wie bei Mach beschrieben durch einen Record aus

- einer Beschreibung, die angibt, um welche LLVM-Instruktion es sich handelt, die für Sprünge, switch-, Label-Instruktionen und Funktionsaufrufe zusätzliche Informationen enthält. Bei Sprüngen werden hier die Ziellabel(s) gespeichert, bei switch-Instruktionen wird gespeichert, von welchem Wert zu welchem Fall gesprungen wird, bei Labels wird der Name und bei Funktionsaufrufen – wie gehabt – die Funktion die aufgerufen werden soll, gespeichert.
- einem Array der Parameter für diese Instruktion,
- einem Register, in das das Ergebnis geschrieben werden soll, sofern ein solches benötigt wird,
- Debugging-Informationen für zukünftige Verwendung, bisher werden sie einfach ignoriert und

- einer Referenz auf die nächste Instruktion.

Im Gegensatz zu Mach sind hier die Beschreibung und der Zeiger auf das nächste Element veränderlich, um Transformationen zu erlauben. Dies ist insbesondere für zukünftige Optimierungen gedacht, wird aber auch jetzt schon verwendet, um alle `alloca`s an den Anfang einer Funktion zu verschieben. Ein Typ muss bei einer Instruktion nicht mehr explizit gespeichert werden, da schon alle Register und Konstanten mit Typinformationen ausgestattet sind und der Code beim Erstellen der Linearize-Repräsentation mit allen nötigen Casts versehen wird.

Programme in der Linearize-Darstellung entsprechen dann so genau der LLVM-IR-Darstellung, dass das abschließende Übersetzen in LLVM-Assembler nicht einmal 250 Zeilen Code erfordert, der größtenteils aus einfachem Pattern Matching besteht.

Durch die Verwendung dieser zwei Repräsentationen ist die Lesbarkeit des Codes im Backend erheblich gestiegen, da so die einzelnen Übersetzungsschritte wesentlich einfacher ausfallen als bei den vorherigen Varianten.

Dadurch, dass die Linearize-Darstellung sehr nah am generierten LLVM-IR-Code ist, dürfte es auch recht leicht sein, die Codeerzeugung auf die Verwendung der LLVM-API umzustellen, um sich so den Aufwand des Schreibens temporärer Dateien sowie des anschließenden Parsens von deren Inhalt zu ersparen.

3.2 Exception Handling

Während die meisten Transformationen leicht realisierbar waren, erwies sich das Exception Handling von OCaml als problematisch.

Bei der bisherigen Implementierung von OCaml wird das Exception Handling durch einige Assembler-Instruktionen erledigt. Zum Fangen einer Exception wird (in Intel-Syntax)

```
push    r14
mov     r14, rsp
```

ausgeführt. Anschließend erfolgt ein `call` des Blocks, in dem eine Exception auftreten kann, sodass mit einem `ret` zum Exception Handler zurückgekehrt werden kann, der nach dem `call` folgt. Hier wird also der aktuelle Stackpointer gesichert, nachdem der vom letzten Exception Handler auf dem Stack gesichert wurde. Um zum Handler zurückzukehren, muss man nur diesen Stackpointer wiederherstellen, den alten Exception Handling-Zeiger vom Stack lesen und gemäß der auf dem Stack befindlichen Rücksprung-Adresse zum eigentlichen Exception Handler springen:

```
mov     rsp , r14
pop     r14
ret
```

Der Exception Handler erwartet einen Zeiger auf das Exception-Objekt im Register `rax`.

Diese Konstruktion bietet eine ausgesprochen gute Performance. Zum einen implementiert sie das Exception Handling so, dass das Werfen und Fangen einer Exception kaum schneller realisiert werden kann, zum anderen ist auch der Overhead, wenn die Exception nicht geworfen wird, ausgesprochen gering. In diesem Fall wird nur unnötigerweise ein Register im Speicher gesichert und ein Funktionsaufruf ausgeführt.

Dies lässt sich mit LLVM leider nicht ohne weiteres implementieren. Zunächst ist es nötig, dass immer der richtige Zeiger in `r14` gespeichert ist. Dies lässt sich mit einer eigenen Calling und Return Convention mit LLVM recht leicht realisieren. Dazu gibt jede Funktion zusätzlich zu ihrem eigentlichen Ergebnis einen zweiten Wert zurück, der im Register `r14` zurückgegeben wird. Entsprechend wird `r14` an jede Funktion als Parameter übergeben, sodass jederzeit der korrekte Wert zur Verfügung steht. Wenn er gerade nicht benötigt wird, hat LLVM zudem die Freiheit, den Wert auf dem Stack zu sichern und das Register in dieser Zeit für etwas anderes zu verwenden.

Doch das ist nicht das einzige Problem. Das wesentlich größere Problem ist, dass LLVM keine expliziten Manipulationen des Stacks zulässt. Die einzigen Möglichkeiten, um mit LLVM den Stack zu verändern, sind das Aufrufen von und das Zurückkehren aus

Funktionen. Ansonsten schreibt LLVM eventuell noch Werte auf den Stack, wenn Spilling nötig ist, oder um Register bei Funktionsaufrufen zu sichern. Diese Einschränkung macht auch durchaus Sinn, da zum einen explizite Stackmanipulationen für die Übersetzung vieler Programmiersprachen nicht nötig sind und zum anderen das explizite Manipulieren des Stacks den Abstraktionsgrad von LLVM-IR soweit senken würde, das vieles, was mit der jetzigen Variante leicht umsetzbar ist, bei expliziten Änderungen des Stack nicht oder deutlich schwieriger zu realisieren wären.

Es gibt beim LLVM zwar gewisse intrinsische Funktionen zum Lesen und Schreiben des Stackzeigers (`llvm.stacksave` und `llvm.stackrestore`), die aber nur für die Implementierung von dynamischen Arrays gedacht sind.[19] Aufgrund der vorgesehenen Verwendung werden die Aufrufe dieser Funktionen beim Optimieren entfernt, wenn sie nicht sehr nah beieinander stehen, was beim Exception Handling bekanntlich normalerweise nicht der Fall ist.

Da die Änderungen, die nötig wären, um LLVM explizite Stackmanipulationen beizubringen zum einen aus vielen Gründen nicht sinnvoll wären und zum anderen mit Sicherheit den Rahmen dieser Bachelorarbeit gesprengt hätten, bleib mir nichts anderes übrig, als von diesem Exception Handling-Verfahren Abstand zu nehmen und damit auf die Binärkompatibilität mit bestehenden OCaml-Binärdateien endgültig zu verzichten.

3.2.1 Alternative Exception Handling-Verfahren

Für das Exception Handling stehen also die beiden offensichtlichen Möglichkeiten zur Verfügung: Exception Handling mit setjmp/longjmp oder C++-/Zero Cost-Exceptions, die beide von LLVM grundsätzlich unterstützt werden.

Um die relativ hohe Performance bei der Behandlung von Exceptions erhalten zu können, fiel die Wahl auf setjmp/longjmp. Da dies jedoch von LLVM nur auf arm/darwin unterstützt wird, blieb eigentlich nur die Möglichkeit, die setjmp/longjmp-Implementierung

[19] vgl. LLVM Assembly Language Reference, `http://llvm.org/docs/LangRef.html#int_stacksave` (20. 3. 2012)

aus der libc zu verwenden. Diese verbrauchen, wie oben schon erwähnt, auf AMD64 zwar deutlich mehr Speicher (und Zeit) als GCCs eingebaute setjmp/longjmp-Implementierung, die auch von LLVM zur Verfügung gestellt wird, sind aber beim Werfen und Fangen von Exceptions noch deutlich schneller, als C++-Exceptions.

Da etwa in der OCaml-Standardbibliothek an vielen Stellen Exceptions in Situationen geworfen werden, die alles andere als außergewöhnlich sind, wie der Name „Exception“ eigentlich erwarten ließe, ist die Geschwindigkeit beim Werfen und Fangen von Exceptions relativ wichtig. Daher ist dies, auch wenn es nicht gerade ideal ist, besser als die Verwendung von Zero Cost Exceptions.

Das Exception Handling wird nun auf Basis von setjmp/longjmp umgesetzt, was auch die Interoperabilität mit C-Code verbessern dürfte (bisher war es zwar möglich, im C-Code Exceptions zu werfen und diese in OCaml zu fangen, Exceptions in OCaml-Code zu werfen und in C- oder OCaml-Code zu fangen, aber nicht Exceptions in C-Code zu werfen und zu fangen). Dadurch, dass jetzt nur noch setjmp/longjmp benutzt werden, besteht diese Einschränkung nicht mehr.

3.3 Garbage Collection

Für die Garbage Collection braucht OCaml zusätzlich zu der Unterscheidung von Zahlen und Zeigern, die durch die tagged-integer-Darstellung gewährleistet ist, noch eine wesentliche Komponente: Die möglichen Wurzeln, von denen aus Speicher auf dem Heap erreicht werden kann. Da LLVM dies nur mit Speicherplätzen auf dem Stack erlaubt, müssen vor jedem Funktionsaufruf alle Zeiger, die vorher geschrieben und nachher gelesen werden, auf dem Stack gesichert werden. Die Offsets der dazu verwendeten Speicherplätze relativ zum Stackpointer werden in der Frametabelle („frametable“) gespeichert.

LLVM bietet Hilfe bei der Verwendung verschiedener Garbage-Collection-Verfahren, ohne dabei selbst einen Garbage Collector mitzuliefern. Seit LLVM 2.4 ist darauf zwar kein Hinweis mehr in der Dokumentation zu finden, LLVM unterstützt aber dennoch OCamls

Garbage Collector.

Man kann LLVM mitteilen, dass eine lokale (mit `alloca` angelegte) Variable eine Wurzel ist, indem man die intrinsische Funktion `llvm.gcroot` aufruft und die Adresse der lokalen Variable als erstes Argument übergibt. Der zweite Parameter kann Metadaten an den Garbage Collector übergeben, was hier aber nicht nötig ist, es wird daher einfach `null` übergeben. Bei der Codeerzeugung erstellt dann LLVM mit Hilfe der Aufrufe von `llvm.gcroot` die Frametabelle und verwirft die Aufrufe.

3.3.1 Probleme

Aufgrund eines Bugs in LLVM lassen sich Programme, in denen gewisse mit `alloca` angelegte Speicherbereiche als Wurzeln markiert sind, nicht kompilieren. Um dieses Problem zu umgehen, werden vom Backend alle `alloca`s an den Anfang der jeweiligen Funktion verschoben.

Da OCamls Garbage Collector Daten im Speicher verschieben kann, ist es nötig, an jeder Stelle, an der der Garbage Collector aufgerufen werden kann (das heißt überall dort, wo eine Funktion aufgerufen wird), alle Register, die einen Zeiger auf ein Element auf dem Heap enthalten können, auf dem Stack zu sichern und nach der Rückkehr aus der aufgerufenen Funktion wieder von dort zu laden. Das heißt, bei jeder Allokation und jedem Funktionsaufruf müssen alle Argumente berechnet werden. Die, die einen Zeiger enthalten können, müssen auf dem Stack gesichert werden und nachdem alle Argumente berechnet wurden, müssen alle, die gesichert wurden, wieder vom Stack geladen werden. Ansonsten könnten es passieren, dass das erste Argument ein Zeiger auf den Minor Heap ist, dessen Ziel bei einer Allokation während der Berechnung des zweiten Arguments auf den Major Heap verschoben wird.

Das funktioniert in der momentan vorliegenden Version des Backends leider noch nicht in allen Fällen. Bei gewissen Programmen gibt es nach wie vor Stellen, an denen ein Zeiger nicht als Wurzel markiert ist, der dennoch vor und nach einem Funktionsaufruf verwendet

wird. Zudem ist es nicht möglich, LLVMs Standardoptimierungen zu verwenden, da diese bei manchen Programmen markierte Wurzeln behandeln, als seien sie keine, das heißt sie werden nach einem Funktionsaufruf nicht aus dem Speicher neu geladen.

4 Bewertung

Ein LLVM-basiertes Backend für OCaml lässt sich durchaus mit verhältnismäßig geringem Aufwand realisieren. Durch die verschiedenen Besonderheiten von OCaml ist dies jedoch mit Binärkompatibilität zum bisherigen Backend kaum realisierbar. Dazu wären deutlich größere Änderungen an LLVM nötig, als etwa für das GHC-Backend, für das nur eine eigene Calling Convention integriert werden musste. Hier zeigt sich die gegenüber Assembler höhere Abstraktionsebene von LLVM. Dinge, die mit wenigen Assembler-Instruktionen beim bisherigen Backend implementiert werden können, lassen sich mit LLVM nicht in kompatibler Form realisieren. Insbesondere die expliziten Stackoperationen, die LLVM nicht zulässt, stellen ein erhebliches Problem dar.

Im Gegensatz zu GHCs LLVM-Backend gibt es beim hier entwickelten Backend für OCaml einige erschwerende Faktoren. Zum einen sind alle von GHC erzeugten Funktionsaufrufe endrekursiv, so dass Funktionsaufrufe nie zurückkehren. Eine Return Convention ist also grundsätzlich unnötig. Es ist daher auch nicht nötig, irgendwelche Register vor einem Funktionsaufruf zu sichern. GHC verwendet dabei einen selbst verwalteten Stack, sodass auch LLVMs Einschränkungen bei Stackmanipulationen sowie bei der Garbage Collection irrelevant sind. Der von LLVM verwaltete Stack des Computers wird ausschließlich für das Spilling von Registern innerhalb einer Funktion verwendet. Abgesehen davon ist die Länge des Stacks konstant (keine Caller oder Callee Save-Register und keine Rücksprungadressen).

Der Garbage Collector aus der Laufzeitbibliothek, die auch für die anderen (älteren) Backends eingesetzt wurde, konnte daher ohne Änderungen auch mit dem neuen GHC-

Backend verwendet werden.

Im Gegensatz dazu benötigt OCamls Garbage Collector explizite Unterstützung durch LLVM um sicherzustellen, dass alle Zeiger korrekt geändert werden. So wird LLVM dazu verwendet, OCamls Frametabelle zu schreiben, die beschreibt, an welcher Stelle im Code Zeiger an welchen Stack-Offsets im Speicher liegen.

4.1 Aufwand

Es lässt sich feststellen, dass mit gewissen Einschränkungen ein LLVM-Backend für den OCaml-Compiler in prototypenhafter Form im Rahmen einer Bachelorarbeit (also mit relativ geringem Aufwand) realisiert werden kann. Durch die aufgetretenen Probleme (Umsetzung von Exception Handling, Änderungen am Laufzeitsystem, Garbage Collection) dauerte dies aber länger als ursprünglich erhofft und die Garbage Collection funktioniert leider bislang nicht vollständig.

Der Code für das neue Backend ist dabei deutlich kürzer, als der alte. Das gesamte asmcomp-Verzeichnis umfasst 4647 Zeilen[20] architekturunabhängigen Codes, für AMD64 kommen 67 Zeilen hinzu, bei den anderen Architekturen dürfte die Länge ähnlich ausfallen. Das alte Backend umfasst 6436 Zeilen Code, der für alle Architekturen gleich ist, sowie für AMD64 1149 Zeilen architekturabhängigen Codes, für Architekturen, auf denen Windows nicht verfügbar ist, ist der architekturabhängige Teil etwa halb so lang.

Bei dem architekturabhängigen Code handelt es sich zudem um recht einfachen Code. Es geht hierbei um die Länge von ganzen Zahlen, Zeigern und Gleitkommazahlen im Speicher, die Endianness und Assembler-Direktiven für die Deklaration von Daten.

Die Verwendung von LLVM für die Code-Erzeugung sorgt also für eine deutliche Vereinfachung des Backends, insbesondere beim plattformspezifischen Teil.

[20] gemessen mit cloc (`http://cloc.sf.net`)

4.2 Übersetzungsgeschwindigkeit

Gegenüber dem OCamls bisherigen Native-Code-Backend ist die Übersetzungszeit deutlich gestiegen. Dies wird sicherlich unter anderem dadurch verursacht, dass bei der Übersetzung zunächst in Form von Strings einzelne LLVM-IR-Instruktionen „zusammengebastelt“, in eine Datei geschrieben (Festplattenzugriff!) von `opt` aus dieser Datei gelesen, geparst und nach der Optimierung wieder in eine Datei geschrieben werden, aus der dann `llc` wieder LLVM-IR parsen muss, um anschließend Assembler zu erzeugen, der dann von GCC wiederum aus einer Datei gelesen wird, bevor der Maschinencode mit den Bibliotheken gelinkt in eine ausführbare Datei geschrieben werden kann. Es wird hier also, anders als bisher, statt nur einmal den OCaml-Code und einmal Assembler zu parsen auch noch zweimal LLVM-IR geparst und jeweils zwischenzeitlich in eine Datei geschrieben.

Die Übersetzungsgeschwindigkeit ließe sich sicher verringern durch Verwendung der LLVM-API zur Erzeugung von LLVM-IR sowie durch Verzicht auf das Speichern der temporären Dateien. Wie mit der GCC-Option „`-pipe`“ könnten die Daten von `opt` direkt, das heißt ohne in eine Datei geschrieben werden, an `llc` weitergereicht und der resultierende Assembler-Code dann an GCC weitergeleitet werden.

	`ocamlopt`	`ocamllvm`
ocamlllvm	10.78	172.04
stdlib	10.93	43.02

Tabelle 4.1: Übersetzungsdauer

Wie man leicht sehen kann ist die Übersetzungsdauer bei Verwendung vom neuen Backend (Spalte `ocamllvm`) erheblich länger als beim bisherigen Backend (Spalte `ocamlopt`), sowohl beim Übersetzen von ocamlllvm, als auch beim Übersetzen der Standardbibliothek (stdlib).

Der Aufwand für die Übersetzung des LLVM-IR-Codes in Assembler-Code ist also um ein Vielfaches höher, als der gesamte Übersetzungsvorgang beim bisherigen Compiler. Dabei muss man natürlich beachten, dass LLVM aggressive Optimierungen durchführt.

4.3 Ausführungsgeschwindigkeit

Verglichen mit dem von `ocamlopt` erzeugten Code ist der vom neuen Backend erzeugte Code mit LLVMs Standardoptimierungen meist langsamer. Ohne diese Optimierungen, die durch Bugs in LLVM zu Fehlern in manchen Programmen führen, laufen die meisten Programme deutlich langsamer. Aufgrund von Fehlern bei der Gleitkommaarithmetik und den schon erwähnten Problemen bei der Garbage Collection konnte ich leider nur wenige der von ocamlnat[21] übernommenen Benchmarks testen.

Benchmark	ocamlc	ocamlopt	ocamlllvm
fib	11.09	0.88	1.26
takc	5.46	0.38	1.64
taku	8.54	0.38	2.09

Tabelle 4.2: Laufzeit

Laufzeit bei verschiedenen einfachen Benchmarks. Die meisten komplizierteren funktionieren aufgrund von Fehlern bei der Gleitkommaarithmetik und dem schon erwähnten Problem mit der Garbage Collection nicht und konnten deshalb nicht getestet werden.

Zu einem erheblichen Teil ist diese ziemlich schlechte Performance darauf zurückzuführen, dass bei den vorliegenden Benchmarks guter OCaml-Code vorliegt, den `ocamlopt` in guten Maschinencode übersetzen kann. Zudem ist es aber so, dass das neue Backend nahezu nie Callee-Save-Register ausnutzt, sehr viele Zwischenergebnisse auf dem Stack zwischenspeichert und ohne LLVMs Optimierungen läuft, da diese bei gewissen Programmen zu Fehlern bei der Garbage Collection führen. Gegenüber der mit Optimierungen übersetzten Variante gibt es aber bei den hier getesteten Programmen nicht mehr als zehn Prozent Geschwindigkeitseinbußen.

[21] ocamlnat Homepage `http://benediktmeurer.de/ocamlnat/` (21. 3. 2012)

5 Schluss

Insgesamt ist festzustellen, dass LLVM für ein Backend für OCaml durchaus geeignet ist, auch wenn es einiges gibt, was besser funktionieren könnte. Verglichen mit dem LLVM-Backend für GHC ist der Aufwand jedoch zum Teil deutlich größer. Teilweise weil GHC C++-Exception Handling verwendet, teilweise weil GHC einen eigenen Stack einsetzt und somit nicht auf LLVMs Unterstützung bei der Garbage Collection angewiesen ist und teilweise weil GHC bereits so gut optimiert, dass LLVMs Optimierungen – zumindest bei der ersten Version des neuen Backends – nichts Sinnvolles mehr zu tun hatten.[22]

5.1 Zukünftige Ziele

Mit Verwendung der LLVM-API wäre es wohl auch recht leicht möglich, auf Basis des im Rahmen dieser Arbeit entwickelten Ahead-of-time-Compilers einen JIT-Compiler für OCaml zu implementieren. Das vielleicht größte Problem dabei wäre, dass die momentane Darstellung von Daten auf dem Heap sich in dieser Form bei den globalen Variablen nicht nativ mit LLVM-IR darstellen lässt. Dazu kommt architekturspezifischer Inline-Assembler zum Einsatz, für den einfach die Ausgabe von globalen Variablen des bisherigen Native-Code-Backends in einer für LLVMs Inline-Assembler geeigneten Form verwendet wird.

Es wäre interessant zu untersuchen, wie sich die Performance mit besserem Exception Handling (etwa mit GCCs `__builtin_setjmp`/`__builtin_longjmp`) und vor allem

[22] vgl. Terei 2009

besserer Unterstützung für OCamls Garbage Collector durch LLVM verändern würde. Eine Erweiterung von LLVM, die es zuließe, SSA-Register und nicht nur Stack-Slots als Wurzeln zu markieren, würde zum einen den generierten LLVM-IR-Code erheblich übersichtlicher machen, zum anderen dürfte dies bei entsprechender Unterstützung von LLVM dazu führen, dass die Register nur dann auf dem Stack gesichert würden, wenn dies nötig ist. Momentan ist es nämlich so, dass auch Stack-Slots, in denen sowieso kein sinnvoller Zeiger mehr steht, dem Garbage Collector als mögliche Wurzeln übergeben werden. Mit einer einfachen Liveness-Analyse für SSA-Register könnte dies deutlich verbessert werden.

Hinzu kommt, dass die Stack-Slots für die Garbage Collection am Anfang jeder Funktion mit Null belegt werden, auch wenn sie vor dem ersten Funktionsaufruf oder vor dem ersten Lesen noch überschrieben würden.

Literaturverzeichnis

[1] Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D. (2007): Compilers: Principles, Techniques, & Tools, 2. Auflage, Boston 2007.

[2] Leroy, Xavier. „Re: [Caml-list] Google summer of Code proposal“ 23. März 2009, `caml-list@inria.fr` (21. 2. 2012).

[3] Terei, David A. (2009): Low Level Virtual Machine for Glasgow Haskell Compiler, `http://www.llvm.org/pubs/2009-10-TereiThesis.pdf` (14. 3. 2012).

[4] Terei, David A.; Chakravarty, Manuel M. T. (2010): An LLVM Backend For GHC, in „Proceedings ACM SIGPLAN Haskell Symposium 2010“, Baltimore MD, United States, September 2010.

[5] o.V.: System V ABI `http://x86-64.org/documentation/abi.pdf`.

[6] o.V.: Caml Homepage `http://caml.inria.fr/`.

[7] o.V.: Clang Homepage `http://clang.llvm.org/`.

[8] o.V.: DragonEgg Hompage `http://dragonegg.llvm.org/`.

[9] o.V.: GHC Homepage `http://www.haskell.org/ghc/`.

[10] o.V.: LLVM Hompage `http://llvm.org/`.

[11] o.V.: LLVM Homepage: Projects built with LLVM `http://llvm.org/ProjectsWithLLVM/`.

[12] OCaml Homepage `http://caml.inria.fr/ocaml/index.en.html`.

[13] ocamlnat Homepage: http://benediktmeurer.de/ocamlnat/.

[14] van Schie, John: Compiling Haskell to LLVM (Thesis defense) http://www.cs.uu.nl/wiki/bin/view/Stc/CompilingHaskellToLLVM.