Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

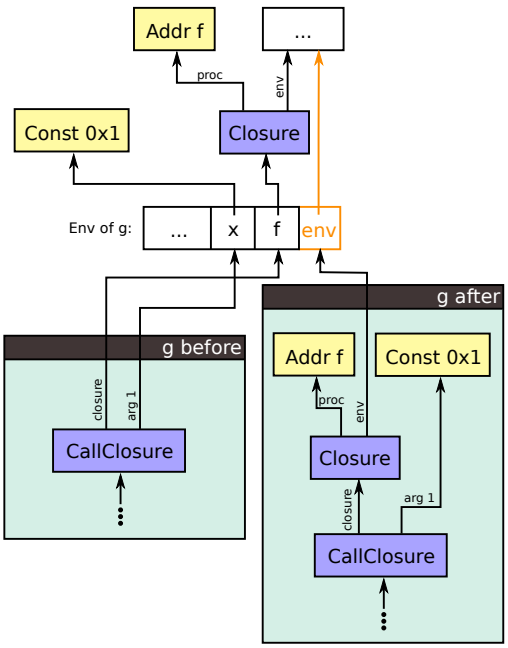
Lehrstuhl Prof. Dr.-Ing. Snelting

Optimierungen in einer funktionalen Sprache mit Firm

Bachelorarbeit von

Daniel Krüger

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M.Sc. Andreas Fried

Bearbeitungszeit: 12. Januar 2017 – 12. Mai 2017

Zusammenfassung

Funktionale Sprachen bieten durch Funktionen höherer Ordnung mehr Abstraktionsmöglichkeiten. Um diese Konstrukte effizient zu kompilieren, müssen in der Regel spezielle Optimierungen implementiert werden. Gegenstand dieser Arbeit ist die Entwicklung eines Compilers für Standard ML mit `libfirm`. Hierzu gehört die Erweiterung der Firm-Darstellung um spezielle Knoten zur Darstellung von Closures. Für diese Knoten erfolgte die Implementierung einer Optimierungsphase, nach der die Knoten zu primitiveren Knoten abgebaut werden.

Functional languages offer more possibilities for abstraction through higher-order functions. In order to compile these constructs efficiently, special optimizations generally have to be implemented. In this work we develop a compiler for Standard ML with `libfirm`. Additionally we extend Firm's intermediate representation by special nodes which represent operations on closures. We implement an optimization phase for these nodes, after wich they are lowered to more primitive nodes.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

In der funktionalen Programmierung werden Funktionen oft als Argumente genutzt. Ein Beispiel hierfür ist die `map`-Funktion, die eine Funktion auf jedes Element einer Eingabeliste anwendet und die Ergebnisse in einer neuen Liste zurückgibt (siehe Abbildung 1.1 `booked_seats` und `bookers`). So kann eine höhere Abstraktion erreicht werden, da in einer Funktion nicht nur über Werte, sondern über ganze Teile des Algorithmus abstrahiert werden kann.

Abbildung 1.1 stellt ein funktionales Programm in Standard ML dar. Es wird die Funktion `fold` definiert, die anschließend von `sum` und `join` benutzt wird. Dabei erhält `fold` als erstes Argument eine Funktion, die angibt, wie jedes Listenelement mit dem aktuellen Akkumulator zu einem neuen Akkumulator verknüpft werden soll. Im Falle von `join` wird dazu mit `fn` eine anonyme Funktion erzeugt, die außer ihren eigenen Argumenten (`l` und `r`) das Argument `sep` aus der umgebenden Funktion `join` verwendet.

In den Variablendefinitionen `booked_seats` und `bookers` werden diese Funktionen verwendet. Zudem sieht man die anfangs erwähnte Funktion `map`, die in diesen Fällen eine Funktion übergeben bekommt, die ein Feld aus einem Record extrahiert.

1.2 Problem

Einige Konstrukte aus funktionalen Programmiersprachen wurden in der Vergangenheit in imperative Programmiersprachen eingebaut. Sprachen wie zum Beispiel Java, Javascript und Python arbeiten mit einem Garbage Collector und besitzen anonyme Funktionen mit lexikalischem Kontext [1, 2, 3]. Da diese Konstrukte maschinenfern sind, ergibt sich das Problem diese effizient zu kompilieren.

```
(* fold f accum [1, 2, 3]
 *   = fold f accum (1 :: (2 :: (3 :: [])))
 *   = f 1 (f 2 (f 3 accum))
 *)
fun fold f accum ls =
  case ls of
    [] => accum
    | head :: tail => f head (fold f accum tail)

(* summiere alle Zahlen einer Liste auf *)
fun sum ls =
  fold (op +) 0 ls

(* verbinde alle Strings einer Liste mit dem
 * Seperator <sep> zwischen den Elementen
 *)
fun join sep ls =
  fold (fn l r => l ^ sep ^ r) "that's it." ls

val bookings =
  [ { name = "John", seats = 3 }
  , { name = "Mary", seats = 5 }
  , { name = "Hough", seats = 2 }
  ]

val booked_seats =
  sum (map (# seats) bookings)
  (* => 10 *)

val bookers =
  join ", " (map (# name) bookings)
  (* => "John, Mary, Hough, that's it." *)
```

Abbildung 1.1: Beispiel für funktionale Programmierung

2 Grundlagen und verwandte Arbeiten

2.1 Compiler Allgemein

Moderne Compiler lassen sich in der Regel in Frontend, Middleend und Backend einteilen. Das Frontend liest den Quelltext ein und wandelt ihn in eine strukturierte Zwischendarstellung um. In der darauf folgenden Optimierungsphase formt das Middleend diese Zwischendarstellung in mehreren Schritten in semantisch äquivalente Formen um. Dadurch soll eine Form erreicht werden, die möglichst effizient ausgeführt werden kann. Abschließend generiert das Backend aus der Zwischendarstellung den endgültigen Maschinencode.

2.2 SSA

Static Single Assignment ist eine Form von Zwischendarstellungen, die sich in den Entwicklungen neuer Compiler für imperative Sprachen durchgesetzt hat. Prominente Beispeile für Compiler, die eine SSA-Zwischendarstellung benutzen, sind llvm, gcc und hotspot. [4, 5, 6]

Das Programm wird in Grundblöcke aufgeteilt, um bedingte Ausführung zu ermöglichen. Grundblöcke stellen eine Ausführungseinheit dar, die immer komplett oder gar nicht ausgeführt wird. Der Kontrollfluss kommt durch bedingte oder unbedingte Sprünge zwischen den Grundblöcken zustande.

In SSA werden Variablen nach ihrer Definition nicht mehr verändert. Dadurch ergibt sich der Wertfluss ausschließlich aus den Abhängigkeiten in den Variablendefinitionen. Um dabei die unterschiedlichen Wertursprünge darstellen zu können, die sich durch Verzweigungen und Sprünge ergeben, wird eine Φ-Funktion genutzt. Sie vereint mehrere Werte aus unterschiedlichen Ursprüngen und wählt denjenigen aus, der im zuletzt ausgeführten Grundblock zugewiesen wurde. [7, Kapitel 19]

```c
void print(char *str);

int abs(int x) {
    int res;
    if (x < 0) {
        res = -x;
    } else {
        res = x;
    }
    print("abs");
    return res;
}
```

(a) Betragsfunktion in C

```
fun abs(x):
    if (x < 0) L_neg L_nat

  L_neg:
    res_1 := -x
    jmp L_end

  L_nat:
    res_2 := x
    jmp L_end

  L_end:
    res_3 := Φ(res_1, res_2)
    print("abs")
    return res_3
```

(b) Betragsfunktion in SSA

Abbildung 2.1: Vergleich von C mit SSA

Abbildung 2.1b zeigt die Betragsfunktion aus Abbildung 2.1a in SSA-Form. In den Grundblöcken `L_neg` und `L_nat` werden zwei unterschiedliche Werte für res erzeugt, die in `L_end` über eine Φ-Funktion zusammengeführt werden.

2.3 Firm

`libfirm` [8] ist ein Compilerframework, welches das Middleend und Backend eines Compilers umsetzt. Es arbeitet auf der Zwischendarstellung Firm, welche in [9] beschrieben wird. Damit wurden mehrere Compiler implementiert, wie bytecode2firm [10] (Java-Bytecode-Frontend) und cparser [11] (C-Frontend).

Firm beruht auf dem Konzept von SSA und zeichnet sich dadurch aus, dass es auf dem Programm in Graphenform arbeitet. Die Kannten stellen dabei die Werteabhängigkeiten dar, die von Wertnutzer zu Wertursprung zeigen. Die Knoten repräsentieren die Operationen. Im Graph wird die totale Ordnung der Operationen teilweise aufgelöst. Stattdessen stellen die Kanten die semantisch obligatorischen Abhängigkeiten dar. Durch die partielle Ordnung kann Firm beim Erzeugen des Maschinencodes eine optimale Anordnung wählen. In anderen Compilern, deren Zwischendarstellung eine totale Ordnung vorgibt, muss die Information über die Möglichkeiten zur Umordnung erst durch Analysen berechnet werden. Die Details dazu werden in [9] erläutert.

Ein Programm besteht in Firm aus einem oder mehreren Graphen, die jeweils für eine Prozedur stehen. Die Graphen bestehen aus mehreren Grundblöcken (siehe SSA), von denen immer ein Start-Block und ein End-Block vorhanden ist. Der Kontrollfluss beginnt im Start-Block und endet im End-Block. Im Start-Block liegt der Start-Knoten, aus dem der initiale Speicherzustand und die Funktionsargumente bezogen werden können. Der Return-Knoten ist immer die letzte Operation einer Prozedur. Um die gewünschte Ausführungsreihenfolge unter nebeneffektbehafteten Operationen zu beschreiben, gibt es zwischen diesen eine zusätzliche so genannte Speicher-Abhängigkeit.

Abbildung 2.5 zeigt den Firm-Graphen zur Betragsfunktion. Aus dem Start-Knoten werden der initiale Speicherzustand (Knoten 66) und das erste Argument (Knoten 68) extrahiert. Von den drei Verwendern des Arguments befinden sich zwei Verwender (Knoten 76 und Knoten 87) in anderen Grundblöcken, weshalb die Verbindung aus Übersichtlichkeitsgründen unterbrochen ist. Der dritte Verwender vergleicht das Argument (Knoten 70) mit Null. Der Cond-Knoten verwendet das Ergebnis, um zu entscheiden, ob zu Block 74 oder zu Block 75 gesprungen wird.

In Block 78 wird der negierte Argumentwert aus Knoten 76 und der unverarbeitete Argumentwert aus Knoten 68 in einem Φ-Knoten vereint und an den Return-Knoten weitergegeben. Zusätzlich wird die Funktion print in Knoten 85 aufgerufen, die keinen Rückgabewert produziert. Hier wird ersichtlich, wie über die Speicherabhängigkeiten die gewünschte Ausführungsreihenfolge von nebeneffektbehafteten Operationen sichergestellt wird. Der Aufruf hängt vom initialen Speicherzustand aus Knoten 66 ab und erzeugt einen neuen Speicherzustand, der an den Return-Knoten weitergegeben wird. Somit zeigt die Abhängigkeit des Return-Knotens vom Aufruf, dass vor Beenden der Funktion der Aufruf stattfinden muss.

2.4 Funktionaler Programmierstil

Funktionale Sprachen gehören zu den deklarativen Sprachen. In diesen beschreibt der Quelltext im Gegensatz zu den imperativen Sprachen nicht den Lösungsweg, sondern tendenziell das Problem. Programme sind Funktionen, deren Eingabe ausschließlich durch Funktionsargumente und deren Ausgabe durch den berechneten Wert dargestellt werden, streng genommen ohne dabei auf Nebeneffekten wie der Änderung eines Zustands zu beruhen. Dadurch ergibt sich keine in Blöcken organisierte Befehlsfolge, sondern eine hierarchische Struktur aus Funktionsaufrufen.

Integral für das funktionale Programmierparadigma ist, dass Funktionen ebenso behandelt werden wie andere Werte, sie sind Funktionen *erster Klasse*. Das bedeutet, Funktionen können als Argument eine Funktion annehmen oder als Ergebnis eine

```
fun pythagoras x y =
  sqrt (square x + square y)

fun addCPS x y cont =
  cont (x + y)

fun pythagorasCPS x y cont =
  squareCPS x (fn sqx =>
    squareCPS y (fn sqy =>
      addCPS sqx sqy (fn sum =>
        sqrtCPS sum (fn diag =>
          (cont diag)))))
```

Abbildung 2.2: Pythagorasfunktion in Continuation-Passing-Style

Funktion zurückliefern. Ein bekanntes Beispiel einer solchen *Funktion höherer Ordnung* ist die `map`-Funktion. Wichtig für die Funktionen erster Klasse ist auch, dass sie abhängig von ihrem lexikalischen Kontext sein können. Das heißt, dass innerhalb der Funktionen auf Variablen zugegriffen werden kann, die in den umgebenden Funktionen definiert werden. Um auf diese sogenannten *freien Variablen* zugreifen zu können — obwohl sie zum Aufrufzeitpunkt der Funktion nicht mehr Teil des Kontexts sind — werden Closures benutzt.

2.5 Funktionale Compiler

Die Unterschiede zwischen dem imperativen und dem funktionalen Paradigma zeigen sich auch in den Compilern. Teilweise verwenden funktionale Compiler eine zusätzliche höhere Zwischensprache. Darin werden die Informationen über Abhängigkeiten von Funktionen untereinander erhalten, welche Optimierungen ausnutzen können.

Im Backend einiger funktionaler Compiler wird das Programm im Continuation-Passing-Style (kurz CPS) ausgedrückt [12]. Hier besteht eine starke Ähnlichkeit zu SSA (siehe [13]). Im CPS bekommen alle Funktionen als zusätzliches Argument eine Continuation. Sie bestimmt, wie nach dem Funktionsaufruf weiter verfahren werden soll. Die Continuation ist selbst eine Funktion, die von der ursprünglichen Funktion mit dem Ergebnis aufgerufen wird. Dadurch müssen die Funktionen nicht mehr zu ihrem Aufrufer zurückspringen und Funktionsaufrufe können als einfacher Sprung mit Argumenten implementiert werden. Um in den Continuations auf die bisherigen Ergebnisse zugreifen zu können, brauchen sie dabei Zugriff auf ihren lexikalischen

Kontext.

Abbildung 2.2 zeigt die Pythagorasfunktion (`pythagorasCPS`) in Continuation-Passing-Style. Sie benutzt CPS-konvertierte Funktionen, von denen eine beispielhafte Implementation von `addCPS` zu sehen ist. Primitive Funktionen werden zu CPS konvertiert, indem sie die Continuation als zusätzliches Argument übergeben bekommen und diese mit dem Ergebnis der ursprünglichen Operation aufrufen. In `pythagorasCPS` wird innerhalb der Funktion `(fn sqy => ...)` sowohl auf das Funktionsargument `sqy`, als auch auf `sqx` zugegriffen. `sqx` ist dabei das Argument der umgebenden Funktion und somit aus dem lexikalischen Kontext der `squareCPS-y`-Continuation.

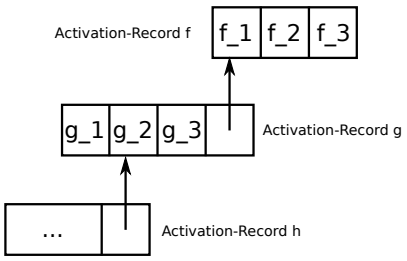
2.6 Closures

In der Funktionalen Programmierung braucht man innerhalb von Funktionen Zugriff auf ihren lexikalischen Kontext, um das volle Potenzial der Funktionen höherer Ordnung auszunutzen. Closures ermöglichen in einer Funktion den Zugriff auf ihre freien Variablen, nachdem die Activation-Records, aus denen sie stammen, nicht mehr aktiv sind. Das tritt beispielsweise auf, wenn eine Funktion von ihrer umgebenden Funktion zurückgegeben wurde und aufgerufen wird. Deshalb reicht es nicht aus, der inneren Funktion Zugriff auf den Aufrufstack zu geben. Die Variablen, die außerhalb der Funktion definiert worden sind und innerhalb der Funktion genutzt werden, heißen freie Variablen. Sie müssen zusammen mit der Funktion erhalten werden. Das wird mit einer Closure umgesetzt, welche die Werte der freien Variablen und die Funktion speichert.

Es gibt unterschiedliche Möglichkeiten, Closures umzusetzen:

Wenn heap-allokierte Activation-Records genutzt werden, können sie zusammen mit einem Zeiger auf den Funktionscode als Closure gespeichert werden [14]. Der Garbage Collector erkennt dann, ob die Activation-Records noch erhalten werden müssen. Der Vorteil liegt darin, dass der Aufwand eine Closure zu erzeugen sehr gering ist. Nachteilig ist, dass beim Zugriff auf die freien Variablen eine mehrfache Speicherindirektion durch mehrere Activation-Records auftreten kann. Zudem werden in den Activation-Records möglicherweise mehr Objekte am Leben gehalten als gebraucht werden.

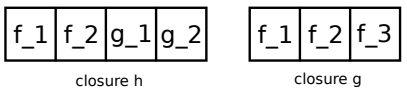
Eine andere Möglichkeit ist es, in der Closure explizit die freien Variablen zusammen mit dem Zeiger auf die Funktion zu speichern [15]. Das ist beim Anlegen der Closure mit höherem Speicheraufwand verbunden und kann dazu führen, dass Werte in mehreren Closures doppelt gespeichert werden. Andererseits tritt es für jede erzeugte Closure nur genau einmal auf und hält weniger ungenutzte Objekte am Le-



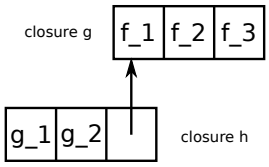
(b) Activation-Records als Closures

```
fun f f_1 f_2 f_3 =
  let
    fun g g_1 g_2 g_3 =
      let
        fun h ... =
          ... f_1 f_2 g_1 g_2 ...
          (* closure h *)
      in
        ... f_3 ...
        (* closure g *)
      end
  in
    ...
  end
```

(a) Beispielprogramm



(c) Flache Closures



(d) Hierarchische Closures

Abbildung 2.3: Darstellung unterschiedlicher Closureimplementierungen

ben. Zudem ist die Anzahl der Speicherindirektionen minimal. Die Closures können flach oder hierarchisch aufgebaut sein. Flache Closures speichern alle freien Variablen in einem Array. In hierarchischen Closures hingegen bestehen Referenzen auf vorhergehende Closures, damit keine Werte aus vorhergehenden Closures mehrfach gespeichert werden.

Weitere Implementationsmöglichkeiten für Closures wurden in [16] untersucht.

2.7 Optimierungen

Optimierungen in Compilern sind Transformationen auf einem Programm, welche das Laufzeitverhalten in Bezug auf Speicherverbrauch und Rechenzeit verbessern sol-

len, ohne dabei die Semantik des Programmes zu ändern. Sie tragen damit dazu bei, dass Programme in einer abstrakteren und möglicherweise einfacher verständlichen Form geschrieben werden können, ohne dabei an Effizienz zu verlieren.

Im Folgenden sind Beispiele für Optimierungen aufgelistet:

Inlining
: Ersetzung von Funktionsaufrufen durch den Funktionskörper.

Dead-Code-Elimination
: Enfernung ungenutzten Codes.

Common-Subexpression-Elimination
: Zusammenfassung von gleichen Unterausdrücken in einer Variable, die stattdessen referenziert wird.

Known-Case-Optimierung
: Ersetzung von Fallunterscheidungen durch einen ihrer Zweige, falls dieser statisch bekannt ist.

Tailrecursion-Optimization
: Umformung von endrekursiven Funktionsaufrufen, so dass der Aufruferstack nicht wächst, beispielsweise indem sie zu einer Schleife umgeschrieben werden.

Abbildung 2.4 stellt einen beispielhaften Ablauf für einige Optimierungen dar. Es ist zu erkennen, wie sie voneinander abhängen und durch eine vorhergehende Optimierung die Darauffolgenden erst möglich werden. In Abbildung 2.4b wurde beim Inlining `f` in `g` eingefügt und `a`, `b` und `c` entsprechend durch die Argumente `4`, `x` und `y` ersetzt. Das Constant-Folding ersetzt `4 > 3` durch `true` (Abbildung 2.4c), danach wird die Fallunterscheidung durch den Zweig für den wahren Fall ersetzt (Abbildung 2.4d). Abschließend wird die ungenutzte Variable `y` entfernt (Abbildung 2.4e).

2.8 Standard ML

Standard ML [17] ist eine funktionale Sprache, die ein strenges Typsystem mit polymorphen Typen, Typinferenz und algebraischen Datentypen hat.

Abbildung 1.1 zeigt ein Beispielprogramm in Standard ML. Auf oberster Ebene befinden sich nur Funktions- und Variablendefinitionen (`fun` und `val`). Die Funktion `fold` benutzt Pattern-Matching mithilfe von `case`, um zwischen der leeren Liste `[]` und einer Cons-Zelle (`::`) zu unterscheiden. In `sum` und `join` wird `fold` benutzt und

```
fun f a b c =
  if a > 3 then
    b
  else
    c

fun g a =
  let
    val x = 1

    val y = ...
  in
    f 4 x y
  end
```

(a) ohne Optimierungen

```
fun g a =
  let
    val x = 1

    val y = ...
  in
    if 4 > 3 then
      x
    else
      y
  end
```

(b) nach dem Inlining

```
fun g a =
  let
    val x = 1

    val y = ...
  in
    if true then
      x
    else
      y
  end
```

(c) nach dem Constant-Folding

```
fun g a =
  let
    val x = 1

    val y = ...
  in
    x
  end
```

(d) nach der Known-Case-Optimierung

```
fun g a =
  let
    val x = 1
  in
    x
  end
```

(e) nach der Dead-Code-Elimination

Abbildung 2.4: Beispiele für Unterschiedliche Optimierungen

mit einer Funktion als Argument aufgerufen. Im Falle von `sum` ist das Argument ein Operator, bei `join` wird eine anonyme Funktion durch `fn` erzeugt.

In den Variablendefinitionen werden die neu definierten Funktionen verwendet. `bookings` ist eine Liste an Records, welche die Felder `name` und `seats` enthalten. In `booked_seats` wird aus den Elementen von `bookings` jeweils das `seats`-Feld extrahiert und aufsummiert. In `bookers` wird jeweils das `name`-Feld extrahiert und mit `join` zu einem String verkettet.

2.9 Thorin

Thorin [18] ist eine Zwischendarstellung im Continuation-Passing-Style, welche wie Firm als Graph dargestellt wird. Da Werteabhängigkeiten durch Kanten im Graphen dargestellt werden, entstehen keine Mehrdeutigkeiten durch Variablennamen. Dadurch wird das Verschieben von Variablendefinitionen vereinfacht. Zusätzlich gibt es in Thorin keine explizite Verschachtelung von Funktionen, sie ergibt sich durch die Variablenabhängigkeiten zwischen Funktionen. Ändern sich die Abhängigkeiten, ergibt sich implizit eine andere Verschachtelung der Funktionen, die nicht explizit gepflegt werden muss.

In Thorin wird versucht alle Funktionen so umzuformen, dass keine Closures aufgebaut werden müssen. Das wird erreicht, indem beim lambda-mangling freie Variablen eliminiert werden. Dabei entstehen so genannte Control-Flow-Forms, welche entweder zu globalen Funktionen oder zu Grundblöcken in SSA übersetzt werden können.

2.10 Optimierungen in MLton

MLton ist ein Compiler für Standard ML, der verglichen mit anderen Standard-ML-Compilern sehr performante Programme erzeugt [19]. Er dient in dieser Arbeit zum Vergleich. MLton betrachtet das Programm im Ganzen, statt es in einzelne Übersetzungseinheiten aufzuteilen [20]. Mit den Informationen über das ganze Programm nutzt es mehr Optimierungsmöglichkeiten aus.

Ein Programm wird von MLton in mehreren Phasen übersetzt. Dabei werden unter anderem Sprachkonstrukte vereinfacht, polymorphe Funktionen und Datentypen instanziiert, Closures eingeführt und Maschinencode erzeugt. Gleichzeitig laufen zwischen den Übersetzungen Optimierungen auf der jeweiligen Zwischendarstellung [21].

Nach der Closure-Umwandlung liegt das Programm in einer SSA-Form vor. In dieser finden die Optimierungen hauptsächlich statt, darunter Dead-Code-Elimination, Inlining, Common-Subexpression-Elimination und die Tailrekursions-Optimierung [22]. Die Zwischendarstellung nutzt allerdings keine Phi-Funktionen, sondern übergibt wie beim Continuation-Passing-Style die jeweiligen Werte an den nächsten Grundblock. Es handelt sich dennoch nicht um Continuation-Passing-Style, da die Funktionen einen Wert zurückliefern und keine Continuation als Argument benötigen.

Die Closures sind in MLton flach und enthalten statt eines Prozedurzeigers einen Tag. Er unterscheidet, welche Prozedur auszuführen ist. Dazu wird an jeder Stelle bestimmt, welche Funktionen auftreten können. Ihnen wird eine neue Menge an Tags zugeordnet, welche in den entsprechenden Closures genutzt werden. An der Stelle der Aufrufer werden die Tags wieder aufgelöst und die zugeordnete Funktion mit den freien Variablen aus der Closure aufgerufen [23].

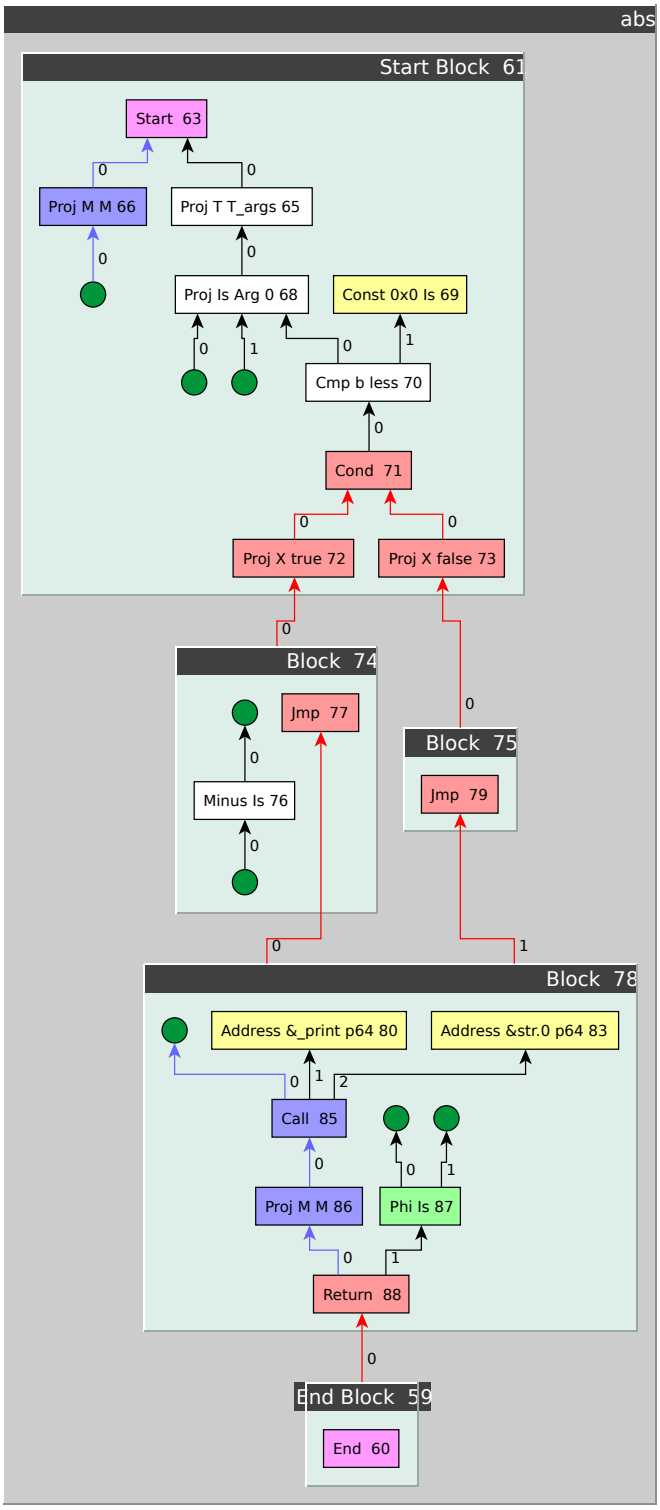


Abbildung 2.5: Betragsfunktion in Firm

3 Entwurf und Implementierung

Das Ergebnis dieser Arbeit ist der Compiler `sml-firm` für ein Subset von Standard ML. Der Compiler besteht aus zwei Teilen: Dem Frontend, welches Standard ML zu einem Firm-Graphen übersetzt, und dem Middle- und Backend, wofür `libfirm` dient. Es wurde das komplette Frontend entwickelt, welches ein Standard-ML-Programm zu einem erweiterten Firm-Graphen mit speziellen Knoten für Closure-Operationen übersetzt. `libfirm` wurde erweitert, den Graphen mit den neuen Knotentypen entgegenzunehmen, darauf spezielle Optimierungen durchzuführen und sie zu einfacheren Operationen abzubauen.

Das implementierte Subset von Standard ML enthält die folgenden Konstrukte: rekursive und anonyme Funktionen, Fallunterscheidung über if-then-else und arithmetische Operationen. Werte können entweder Ganzzahlen, boolesche Werte, Strings oder einfach verkettete Listen aus Ganzzahlen sein. Typen müssen im Gegensatz zu Standard ML explizit angegeben werden und homomorph sein.

Folgende externe Funktionen sind in C definiert (siehe `src/support/src/lib.c`) und werden immer eingebunden:

`inc` : `int -> int`	Erhöht eine intern gespeicherte Zahl um den angegebenen Wert und gibt ihren neuen Wert aus. Der interne Wert ist initial Null.
`printBool` : `bool -> int`	Gibt einen booleschen Wert aus. Rückgabewert kann beliebig sein.
`printInt` : `int -> int`	Gibt eine Zahl aus. Rückgabewert kann beliebig sein.
`print` : `string -> int`	Gibt den angegebenen String aus. Rückgabewert kann beliebig sein.

`null : intlist -> bool`	Prüft, ob es sich um die leere Liste handelt.
`hd : intlist -> int`	Extrahiert den Kopf aus einer Listenzelle.
`tl : intlist -> intlist`	Extrahiert den die nächste Listenzeille.

3.1 Frontend

Im Frontend laufen die ersten drei Compilerphasen:

- Der Parser, der den Quelltext in den abstrakten Syntaxbaum (AST) umwandelt,
- der Typprüfer, der das Programm auf Typfehler untersucht, und
- der Übersetzer, der den AST in die Firm Zwischendarstellung übersetzt.

Das Frontend ist in Haskell implementiert, um das Typsystem mit Typinferenz und algebraischen Datentypen, Konzepte der funktionalen Programmierung und Monaden für eine schnelle und fehlerarme Entwicklung zu nutzen. Es ergibt sich dadurch der Nachteil, dass Fehler beim Aufbau des Firm-Graphen erst spät erkannt und schwierig zur Fehlerursache zurückverfolgt werden können. So muss zum fehlerhaften Firm-Graphen erst der relevante C-Code ermittelt werden, der selbst wieder dem erzeugenden Haskell-Code zugeordnet werden muss. In der Praxis ist dieses Problem allerdings selten aufgetreten, was auch auf die in den nächsten Abschnitten erwähnten Typannotationen zurückzuführen ist.

Der Parser (`src/SML/Parser.hs`) wurde mithilfe der Bibliothek Parsec, welche mit monadischen Kombinatoren arbeitet (siehe Abbildung 3.1a: `choice`), in Haskell ohne einen Tokenizer umgesetzt. Abbildung 3.1 zeigt ein Beispiel einer Parser-Regel und die entsprechende Grammatik im Vergleich dazu. Im Parser ist auch der Tokenizer [24] (in diesem Beispiel `P.reserved standardml` und `P.identifier standardml`) umgesetzt. Der Parser baut einen abstrakten Syntaxbaum auf, der in `src/SML/AST.hs` definiert ist. In `src/SML/Types.hs` ist die Typprüfung als Teil der Typannotation in der Funktion `annotTypeE` umgesetzt. Danach wandelt `compile` aus `src/SML/Compiler.hs` den Typ-annotierten abstrakten Syntaxbaum in C-Code um, der mit `libfirm` den entsprechenden Firm-Graphen aufbaut.

Anschließend kann mit den Funktionen aus `src/Firm/Compile.hs` das Programm erzeugt werden. `compileWithConfig` kompiliert den generierte C-Code mit Steuercode, der die Optimierungen und das Lowering durchführt, und führt ihn aus.

```
decP :: Parser Dec
decP =
  choice
    [ valP
    , funP
    ]


valP :: Parser Dec
valP =
  do
    P.reserved standardml "val"
    name  <- P.identifier standardml
    P.reserved standardml "="
    expr  <- expP
    return (ValD name expr)
```

(a) Ausschnitt aus dem implementierten Standard-ML-Parser

```
decP ::= <valP>
       | <funP>

valP ::=
  val <id> = <expP>
```

(b) Entsprechende Grammatik

Abbildung 3.1: Vergleich des Parsec-Parsers und einer Grammatik

`compileAssemblyWithConfig` assembliert den entstandenen Assemblercode zu einer ausführbare Datei. `runCompiler` in `src/Main.hs` steuert den gesamten Kompilierungsprozess. `runCompiler` benutzt `compileFile` aus `src/SML.hs`, um Standard-ML-Quelltext zu C-Code zu kompilieren, und erzeugt danach mithilfe der Funktionen aus `src/Firm/Compile.hs` das ausführbare Programm.

Das Frontend generiert C-Code, da es einfacher ist, `libfirm` über generierten C-Code anzusteuern, als die Firm-API im Foreign-Function-Interface umzusetzen. Dieses Vorgehen wird dadurch ermöglicht, dass keine komplexe Interaktion zwischen dem Frontend und `libfirm` nötig ist. Es wird lediglich ein statischer Graph generiert und an `libfirm` übergeben, indem der generierte C-Code den Graphen in Firm aufbaut. Danach finden alle Operationen auf dem Graphen ausschließlich in `libfirm` statt.

Eine eigene Monade, FirmM (in `src/Firm/FirmM.hs`), vereinfacht das Generieren des C-Codes, indem sie den bisher generierten C-Code speichert und als Quelle für neue eindeutige Variablennamen dient. Der C-Code wird intern als String dargestellt. Spezielle Konstruktoren und Typen für Statements, Expressions und Identifier verhindern einfache syntaktische Fehler wie Statements an Stellen, an denen Expressions erwartet werden (umgesetzt in `src/Firm/CCode.hs`).

Um Typfehler im generierten C-Code zu verhindern, werden Ausdrücke durch ei-

```
cmp :: CId -> FNode a -> FNode a ->
         FirmM (FNode CF, FNode CF)
cmp relation l r = ...

newBlock :: [FNode CF] -> FirmM ()
newBlock inputs = ...
```

Abbildung 3.2: Beispiel aus `src/Firm/Simple.hs`

nen Phantom-Typen zusätzlich mit ihrem C-Typen in Firm annotiert[1]. Dieser Typ enthält im Falle von Firm-Knoten zusätzlich Informationen über den Typen des Knotenwerts, die in seinem äquivalenten C-Typen nicht vorhanden sind (siehe `src/Firm/Firm.hs`). So werden panics in `libfirm` durch invalide Graphen verhindert. Dargestellt werden die Annotationen in Haskells Typsystem durch neu eingeführte Typen (siehe `src/Firm/Typed.hs`).

In Abbildung 3.2 sind die Signaturen von `cmp` und `newBlock` dargestellt. `cmp` vergleicht `l` und `r` mit `relation` und erzeugt zwei Control-Flow Knoten für den wahren und den falschen Fall. `newBlock` wechselt vom aktuellen Block zu einem neuen Block, der von den angegebenen Knoten abhängt. Der Typ `FNode CF` zeigt bei `cmp`, dass dieser zwei Knoten (`FNode`) produziert, die den Kontrollfluss (`CF`) ändern. Entsprechend ist bei `newBlock` durch den Typen die Abhängigkeit von solchen Knoten zu sehen. Würde ein Äquivalent zu `newBlock` in C mit einem falschen Knotentypen aufgerufen werden, könnte der Fehler beim Kompilieren nicht durch das Typsystem von C erkannt werden.

Mithilfe dieser Werkzeuge zur C-Generierung wurden zwei Hilfsmodule (`src/Firm/Raw.hs` und `src/Firm/Simple.hs`) umgesetzt, deren Funktionen größtenteils Funktionen aus `libfirm` widerspiegeln. Da das restliche Frontend fast ausschließlich auf diesen Funktionen und der Monade FirmM aufbaut, kann zu einem späteren Zeitpunkt mit geringem Aufwand zum Foreign Function Interface gewechselt werden, indem die Implementationen in den Hilfsmodulen ausgetauscht werden.

[1] Dieser Ansatz wird in [25] beschrieben.

```
(* recursive version, limited by stack *)
fun range (from : int, to : int) : intlist =
  if from < to then
    from :: range (from + 1, to)
  else
    [];

(* tail-recursive version, limited only by heap *)
fun range_tr (from : int, to : int) : intlist =
  let
    fun loop (pos : int, ls : intlist) : intlist =
      if from <= pos then
        loop (pos - 1, pos :: ls)
      else
        ls
  in
    loop (to - 1, [])
  end;

fun fold (f : (int * int) -> int, start : int, ls : intlist) : int =
  if null ls then
    start
  else
    fold (f, f (start, hd ls), tl ls);

val sum =
  fold (fn (accum : int, elem : int) => accum + elem,
        0,
        range (0, 100))
```

Abbildung 3.3: Beispielprogramm im umgesetzten Subset von Standard ML

3.2 Backend

Als Backend dient das am Lehrstuhl entwickelte `libfirm`. Es optimiert und übersetzt die Firm-Zwischendarstellung in Assemblercode, welcher anschließend von einem Assembler zu Maschinencode übersetzt werden kann. Beim Übersetzen des Firm-Graphen werden durch das Lowering sukzessive abstraktere Operationen durch maschinennähere Operationen ersetzt.

Das Frontend baut einen um Closures erweiterten Graphen auf. Sie werden durch die in dieser Arbeit entwickelte Erweiterung von `libfirm` optimiert und danach abgebaut. Diese Knoten erhalten mehr Informationen über Closures, die durch maschinennähere Darstellungen verloren gehen würden. Dadurch sind die Optimierungen für Closures leichter umsetzbar.

Die Closures werden in einer neuen Phase optimiert und ihre spezifischen Knoten danach abgebaut. Die Optimierungen wandeln die durch Closures eingeführten Konstrukte in Formen um, die in späteren Phasen weiter optimiert werden können. Daraufhin arbeiten die restlichen Optimierungen auf dem Firm-Graphen ohne die neuen Knoten.

3.2.1 Closure- und CallClosure-Knoten

Für den Aufbau und den Aufruf von Closures wurden zwei neue Knotentypen eingeführt, Closure und CallClosure. Der Closure-Knoten baut aus einem Prozedurzeiger und einem Zeiger auf die gesicherten freien Variablen — hier Umgebung genannt — einen Closurewert auf. Dieser kann anschließend mit dem CallClosure-Knoten aufgerufen werden, wie eine normale Funktion mit dem Call-Knoten. Die Prozedur der Closure bekommt beim Aufruf der Closure als erstes Argument den Umgebungszeiger und ab dem zweiten Argument die ursprünglichen Argumente vom CallClosure-Knoten.

Closures werden in einem heap-allokierten Struct mit dem Prozedurzeiger und dem Umgebungszeiger gespeichert. Der Struct-Typ einer Closure wird einmalig generiert. Closure-Knoten werden zu den Speicheroperationen abgebaut, die das Struct erzeugen. CallClosure-Knoten werden zu einem Call-Knoten abgebaut, der den Prozedur- und Umgebungszeiger aus der Closure extrahiert und die Prozedur mit dem Umgebungszeiger als erstes Argument aufruft. Die ursprünglichen Argumente an die Funktion werden ab dem zweiten Argument übergeben (Vergleiche Knoten 432 in Abbildung 3.5a).

Die Optimierungen der Closures erkennen, wenn statisch bekannte Closures auf-

gerufen werden, und ersetzen diese Aufrufe durch normale Funktionsaufrufe mit den jeweiligen Umgebungszeigern und Argumenten (siehe Abbildung 3.4c). Dadurch können spätere Optimierungen, wie die Inlining- und die Tailrekursionsoptimierung, diese Aufrufe weiter optimieren.

3.2.2 Env- und EnvRef-Knoten

Für den Aufbau eines Vektors mit den freien Variablen — dem Umgebungsvektor — wurden zwei neue Knotentypen eingeführt: Der Env-Knoten, der einen Umgebungsvektor aufbaut, und der EnvRef-Knoten, der einen Wert aus einem Umgebungsvektor extrahiert. Der Env-Knoten bekommt eine dynamische Anzahl an Werten und erzeugt einen Zeiger auf den Vektor der Werte. Die Werte in diesem Vektor sind unveränderlich. Um einen veränderbaren Wert zu erhalten, müsste in dem Vektor ein Zeiger auf den Wert gespeichert werden.[2] Der EnvRef-Knoten extrahiert aus einer Umgebung den Wert am angegebenen Index.

Die Optimierungen der Umgebungen versuchen diese zu minimieren, so dass die Funktionen möglichst wenige Variablen aus der Umgebung referenzieren müssen. In Abbildung 3.4a hängt `g` von `f` und `x` ab. Diese müssen mit der Closure gespeichert und bei der Ausführung von `g` aus der Closure geladen werden. Die Optimierung zieht konstante Teile aus der Umgebung in die Funktion, so dass innerhalb der Funktion mehr über die Werte der Umgebung bekannt ist (siehe Abbildung 3.4b: `g after` enthält eine eigene Kopie der Konstante `0x1`).

Dies entspricht dem Vorgehen von Thorin beim lambda-mangling, mit dem Unterschied, dass Thorin dabei teilweise auch variable Werte aus der Umgebung zu den Funktionsargumenten verschiebt.

Zusätzlich werden referenzierte Closures so weit wie möglich in die Funktion gezogen. Dabei wird ein EnvRef-Knoten, der eine Referenz auf eine Closure in der Umgebung darstellt, durch den Konstruktor der Closure ersetzt (siehe Abbildung 3.4b, EnvRef-Knoten werden hier nicht direkt abgebildet). Angenommen `g` referenziert `f` wie in Abbildung 3.4a, dann existiert in `g` ein EnvRef-Knoten, der `f` aus der Umgebung extrahiert (in Abbildung 3.4b und 3.4c nicht dargestellt). Der Umgebungsvektor von `f` wird zum Umgebungsvektor von `g` hinzugefügt (`env` in Abbildung 3.4b) und von dort bezogen (siehe `g after`). In den meisten Fällen ist die Funktion konstant und kann danach innerhalb der Funktion für weitere Optimierungen genutzt werden. Nur der Umgebungsvektor zu dieser hineingezogenen Closure muss aus der Umgebung bezogen werden und dafür dem Umgebungsvektor der Funktion hinzugefügt werden,

[2] Dies entspricht der Semantik von Standard ML, in der Werte per se nicht veränderlich sind, außer sie sind in einem `ref` verpackt. Das wurde in diesem Frontend jedoch nicht umgesetzt.

in der die Optimierung gerade läuft.

3.2.3 Weiterer Optimierungsablauf

Durch die Optimierungen der Closures und der Closure-Umgebungen wird der Graph so umgeformt, dass neue Optimierungspotenziale eröffnet werden.

Ein Beispiel hierfür ist in Abbildung 3.5 dargestellt. Abbildung 3.5a zeigt, wie der Graph zu dem Programm aus Abbildung 3.4a nach den Closure-Optimierungen und dem Lowering aussieht. Durch die Vereinfachung des Closureaufrufs ist direkt zu sehen, dass die Prozedur `f` aufgerufen wird. Diese Information wird vom Inlining ausgenutzt (Abbildung 3.5b). Danach werden die sich ergebenden konstanten Ausdrücke durch das Constant-Folding vereinfacht (Abbildung 3.5c). Zudem braucht `f` die Umgebung nicht, weshalb die Load-Store-Optimierung die Lade-Operation entfernt, welche die Umgebung für `f` aus der Umgebung von `g` lädt. Übrig bleibt eine Funktion, die konstant `0x4` zurückgibt (Abbildung 3.5d).

Nicht alle Optimierungspotenziale wurden bisher durch die bestehenden Optimierungen genutzt. Deswegen war es nötig, einige Anpassungen an den bestehenden Optimierungen vorzunehmen, um das Potenzial der Closure-Optimierungen voll ausnutzen zu können.

Die Load-Store-Optimierung hatte teilweise Load-Operationen eines eigentlich bekannten Werts nicht optimiert. Das trat auf, wenn zwischen den entsprechenden Load- und Store-Operationen ein malloc-Aufruf lag. Die Load-Store-Optimierung nahm an, dass darin beliebiger Speicher verändert werden könnte. Allerdings sollten malloc-Funktionen in jedem Fall bestehenden Speicher unangetastet lassen [26]. Deshalb wurde die Load-Store-Optimierung angepasst, zu erkennen, dass malloc-Aufrufe bisher allokierten Speicher nicht verändern. Dafür wurde die bereits vorhandene malloc-Kennzeichnung von Funktionen genutzt.

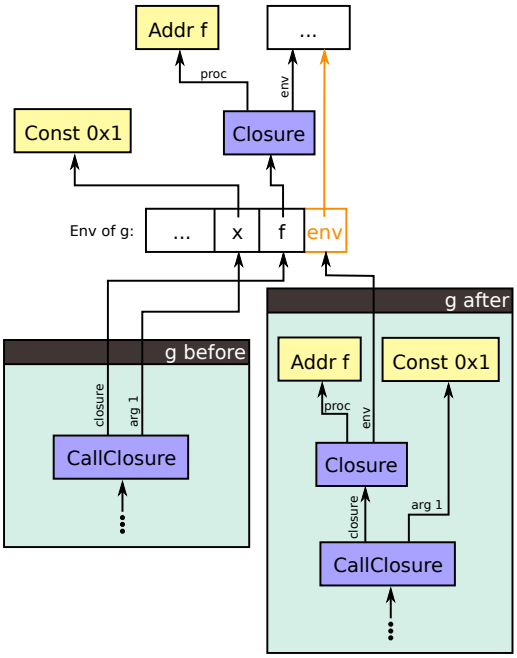
Die Fälle, die dadurch optimiert wurden, werden inzwischen durch die Closure-Optimierungen selbst abgedeckt. Allerdings ist die Anpassung weiterhin für funktionale Programme relevant, da sie die meisten Werte auf dem Heap allokieren. Ohne sie würden viele Optimierungsmöglichkeiten ungenutzt bleiben.

Die tailrec-Optimierung hat endrekursive Aufrufe nicht optimiert, wenn der Typ des Funktionsaufrufs nicht identisch zur Funktion ist. Dies wurde durch einen Zeigervergleich überprüft. Der Vergleich ist beim Optimieren von C nötig, wo Funktionen anders aufgerufen werden können, als sie definiert worden sind. `sml-firm` legt allerdings bei jedem Funktions-Aufruf den Funktionstyp neu an. Vor allem bei Argumenten, in denen eine Funktion übergeben wird, existiert im Allgemeinen keine

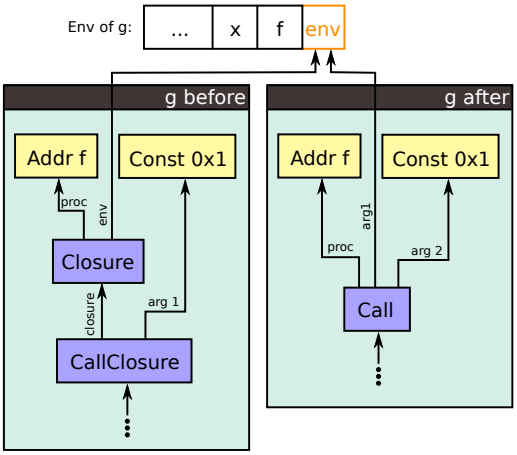
eindeutige Ursprungsfunktion, aus der der Funktionstyp bezogen werden kann. Dadurch reicht der Zeigervergleich nicht aus und müsste durch einen strukturellen Vergleich ersetzt werden. Strukturelle Vergleiche von Typen sind in `libfirm` allerdings noch nicht umgesetzt und müssten Rekursionen in den Typen beachten. Da in Standard ML der Funktionsaufruf jedoch immer zur Funktion passt und die Umsetzung des strukturellen Vergleichs kompliziert geworden wäre, wurde der Typvergleich für Standard ML abgeschaltet.

```
fun test () : int =
  let
    fun f (a : int) : int =
      a + 1;

    val x = 1;

    fun g () : int =
      f x * 2
  in
    ...
  end
```

(a) Beispielprogramm

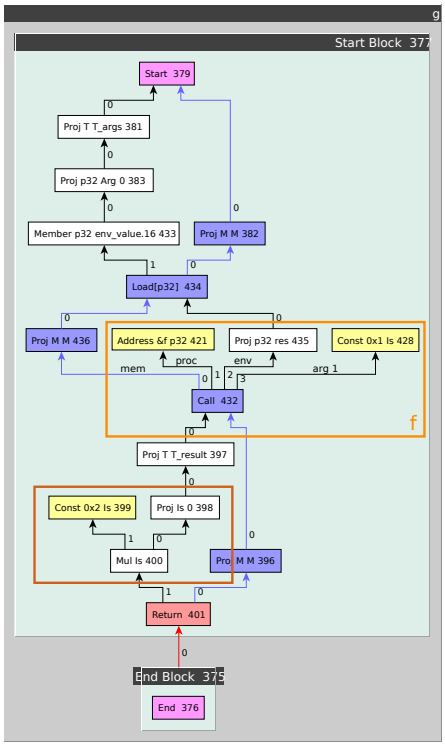


(b) Closure-Konstruktoren und Konstanten aus der Umgebung werden in die Funktion gezogen

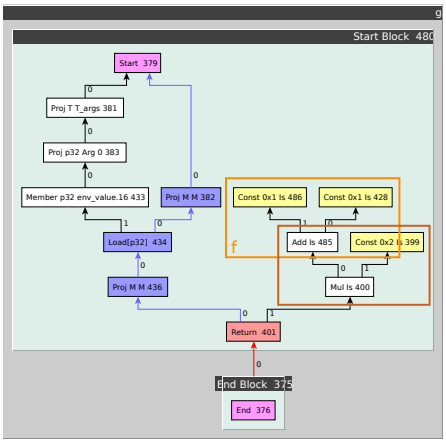


(c) Statische Closureaufrufe werden vereinfacht

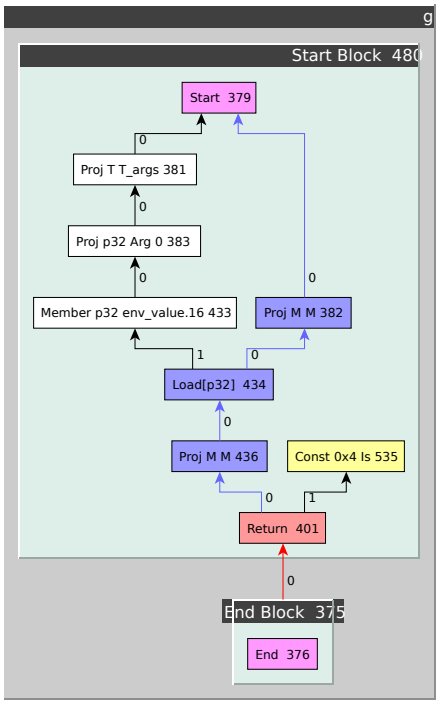
Abbildung 3.4: Optimierungen der Closures



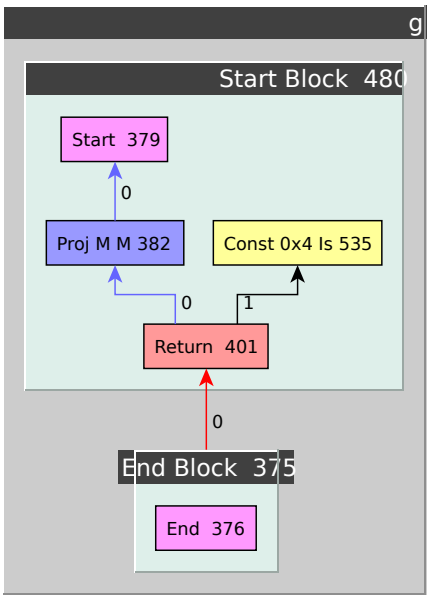
(a) Nach dem Closure-Abbau



(b) Nach dem Inlining



(c) Nach dem Constant-Folding



(d) Nach der Load-Store-Optimierung

Abbildung 3.5: Optimierungen nach dem Closure-Abbau

4 Evaluation

In dieser Arbeit wurden unterschiedlichen Konfigurationen von `sml-firm` und MLton verglichen. Die Benchmarks stammen zum Teil aus dem MLton-Projekt [19], wurden aber an das implementierte Subset von Standard ML angepasst.

Innerhalb der Hauptfunktionen waren die Anpassungen einfacher Natur. Zum einen mussten die Typen annotiert werden, da `sml-firm` keine Typinferenz unterstützt. Zum anderen wurden von `sml-firm` nicht unterstützte Konstrukte durch einfachere Konstrukte ersetzt. Das Pattern-Matching wurde durch `if-else` und im Falle von Listen durch das Prädikat `null` und die Zugriffsfunktionen `hd` und `tl` ersetzt. Verkettete Statements wurden umgesetzt, indem die ersten Statements neuen lokalen Variablen zugewiesen wurden und das letzte Statement im Bauch des entstandenen `let`s ausgeführt wurde. Ein Beispiel für eine Umformung ist in Abbildung 4.1 zu sehen. Es mussten teilweise Hilfsfunktionen ergänzt werden, die in `sml-firm` nicht vorhanden sind. Weiterhin wurden die Hilfsfunktionen, welche die Hauptfunktionen in einer Schleife ausführen, vereinfacht und direkt ausgeführt, statt sie in einer Struktur bereitzustellen.

Die Programme wurden jeweils in fünf Varianten kompiliert:

- Mit `sml-firm` und den vier Konfigurationsmöglichkeiten aus:
  - an- oder ausgeschalteten Optimierungen der Closures (`closure opts`) und
  - mit oder ohne die Anpassungen von Firm (`fixes`)
- Mit MLton

Die mit MLton kompilierten Programme wurden zur Laufzeit so konfiguriert, dass sie einen fixen 6 GB großen Heap verwenden, um ohne Unterbrechungen durch den Garbage Collector zu laufen. Zusätzlich enthalten diese Programme eine große Laufzeitumgebung, deren Overhead bei der Betrachtung der Ergebnisse nicht zu vernachlässigen ist. Auf genauere Messungen des Overheads wurde verzichtet, da der Vergleich mit MLton nur zur Einordnung dient und nicht primäres Ziel dieser Arbeit

```
fun len ls =
  case ls of
    [] => 0
    | h :: t =>
        (print h; 1 + len t)
```

(a) Volle Standard-ML-Syntax

```
fun len (ls : intlist) : int =
  if null ls then
    0
  else
    let
      val ignore = print (hd ls)
    in
      1 + len (tl ls)
    end
```

(b) Nur einfache Konstrukte

Abbildung 4.1: Umformung von Standard ML zu einfacheren Konstrukten

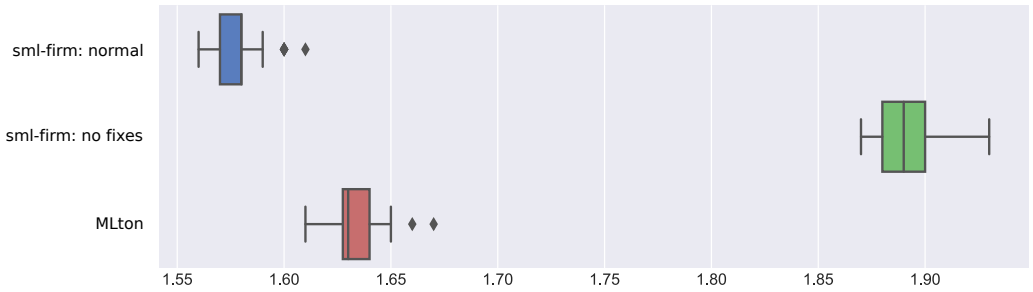
ist. Nicht alle Programme, die mit `sml-firm` kompiliert wurden, konnten fehlerfrei laufen. Die Varianten ohne Closure-Optimierungen oder Anpassungen von Firm haben teilweise entweder zu einem Stackoverflow geführt oder zu viel Speicher auf dem Heap allokiert. Die in den Ergebnissen nicht aufgeführten Varianten sind aus einem dieser Gründe abgestürzt.

Die Benchmarks wurden auf einem 2.6 GHz Intel Core i5 (I5-4278U) Prozessor mit 8 GB RAM unter macOS 10.12.4 (16E195) mit `temci`[27] ausgeführt. Da macOS von `temci` nicht vollständig unterstützt wird, wurden die Tests nicht mit höherer Priorität ausgeführt, was zu den größeren Abweichungen in den Laufzeiten geführt haben kann. Dadurch können keine sicheren Aussagen über kleine Differenzen zwischen den Programmvarianten gemacht werden.

Durch die Einschränkungen des implementierten Subsets von Standard ML sind die Benchmarks so gewählt worden, dass sie ohne Garbage Collector und ohne große Basisbibliothek laufen können. Für MLton wurden zusätzlich Funktionen definiert, die den eingebundenen Funktionen in `sml-firm` entsprechen. Es wurden rekursive und endrekursive Programme ausgewählt, sowie Programme, die Funktionen höherer Ordnung nutzen. Sie zeigen, wie sich die Optimierungen auf die Effizienz von typischen Szenarien in funktionalen Programmen auswirken, und ordnen sie im Vergleich mit MLton in bestehende Compiler ein.

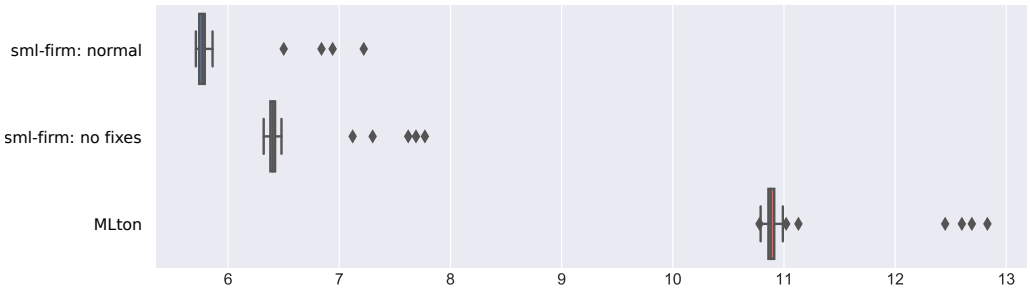
In vielen Fällen, vor allem Listenverarbeitung und höhere Funktionen betreffend (`mapList`, `sumList`, `merge` und `imp-for`), liegt `sml-firm` ein paar Größenordnungen hinter MLton. Sowohl in `merge` als auch in `imp-for` ist deutlich zu sehen, welche Auswirkung die Optimierungen der Closures im Vergleich mit der nicht optimierten Implementation haben. In `imp-for` wird dabei auch eine Funktion höherer Ordnung getestet, wobei die Anpassungen der bestehenden Firm-Optimierungen im Mittelwert eine zusätzliche Laufzeitverbesserung von circa 16% verursachen (`no-fixes` und `normal`).

Auch bei `tak` und `fib-rec` sind deutliche Laufzeitverbesserungen durch die Anpassungen von Firm in Höhe von 17% und 10% zu sehen. Die Varianten ohne Closure-Optimierungen sind durch den hohen Speicherverbrauch abgestürzt, den die primitive Implementation verursacht. Um alle Werte einheitlich darzustellen, wird in einer rekursiven Funktion eine neue Closure für die Funktion selbst angelegt, die später aufgerufen wird. Die Closure-Optimierungen würden diesen Aufbau wieder eliminieren. Dass `sml-firm` etwas schneller als MLton ist, ist wahrscheinlich auf den Overhead der Laufzeitumgebung von MLton zurückzuführen.



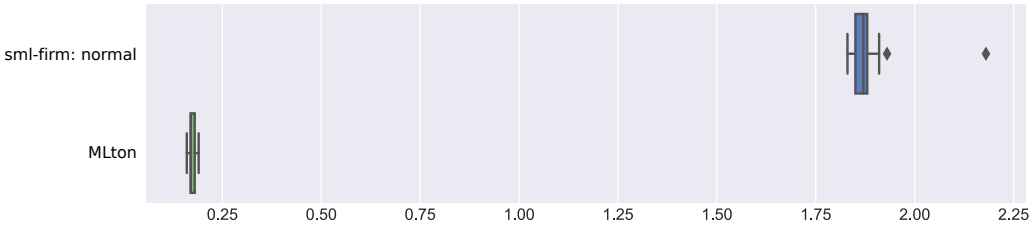
	mean	σ per mean	min	n
sml-firm: normal	1.57(7)	0.67(6)%	1.560	100
sml-firm: no fixes	1.89(1)	0.65(5)%	1.870	100
MLton	1.63(2)	0.65(8)%	1.610	100

Abbildung 4.2: Laufzeit von tak (in Sekunden)



	mean	σ per mean	min	n
sml-firm: normal	5.8(13)	3.8(77)%	5.710	100
sml-firm: no fixes	6.4(56)	3.8(38)%	6.320	100
MLton	10.9(55)	3.1(88)%	10.780	100

Abbildung 4.3: Laufzeit von fib-rec (in Sekunden)

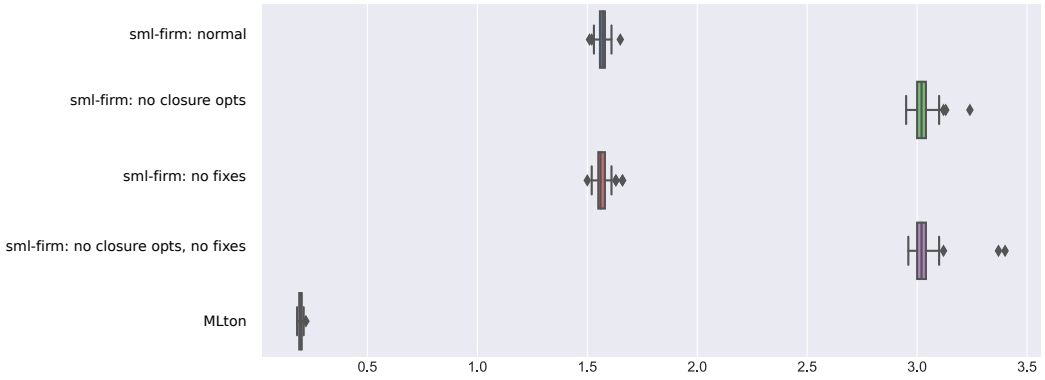


	mean	σ per mean	min	n
sml-firm: normal	1.87(0)	1.9(52)%	1.830	100
MLton	17(3).(000)m	(3).(148)%	160.000m	100

Abbildung 4.4: Laufzeit von mapList (in Sekunden)

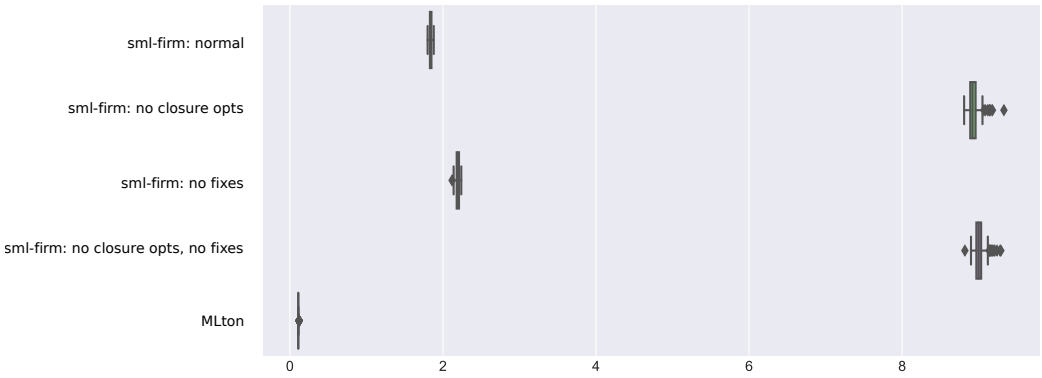
	mean	σ per mean	min	n
sml-firm: normal	6.9(87)	1.83(8)%	6.800	100
MLton	2.6(68)	4.(153)%	2.510	100

Abbildung 4.5: Laufzeit von sumList (in Sekunden)



	mean	σ per mean	min	n
sml-firm: normal	1.56(9)	1.3(79)%	1.510	100
sml-firm: no closure opts	3.02(1)	1.3(16)%	2.950	100
sml-firm: no fixes	1.56(6)	1.6(41)%	1.500	100
sml-firm: vanilla firm	3.0(28)	2.0(07)%	2.960	100
merge: mlton	19(7).(900)m	(3).(298)%	180.000m	100

Abbildung 4.6: Laufzeit von merge (in Sekunden)



	mean	σ per mean	min	n
sml-firm: normal	1.83(9)	0.94(2)%	1.800	100
sml-firm: no closure opts	8.9(43)	0.90(4)%	8.810	100
sml-firm: no fixes	2.19(2)	0.95(7)%	2.120	100
sml-firm: vanilla firm	9.0(17)	0.88(4)%	8.820	100
imp-for: mlton	111.(300)m	(3).(022)%	110.000m	100

Abbildung 4.7: Laufzeit von imp-for (in Sekunden)

5 Fazit und Ausblick

Mit `sml-firm` wurde ein einfacher Compiler für eine funktionale Sprache implementiert. Die Erweiterung von `libfirm` vereinfacht dabei das Arbeiten mit Closures und verbessert mit den neuen Optimierungen für Closures die Performanz signifikant. Im Vergleich mit MLton sind jedoch noch immer große Laufzeitunterschiede zu sehen.

Von diesem Stand aus können verschiedene Punkte weiter verfolgt werden:

Bisher sind noch keine Datentypen implementiert. Listen wurden über externe Funktionen aufgebaut und destrukturiert. Der Anteil des Overheads der Funktionsaufrufe und ausgelassenen Optimierungsmöglichkeiten an der Laufzeit der Listen-lastigen Benchmarks ist ein interessanter Punkt für weitere Anstrengungen.

Die aktuellen Entwicklungen deuten darauf hin, dass in Zukunft viele Sprachen Closures unterstützen könnten. Die hier entwickelte Erweiterung von Firm ist nicht auf funktionale Sprachen eingeschränkt, sie kann auch in einer imperativen Sprache als Startpunkt für weitere Entwicklungen verwendet werden. Ein möglicher erster Schritt ist es, die Erweiterung in einem C++-Compiler zu testen.

Die Lösung über spezielle Closure- und Umgebungs-Knoten ist zwar eine einfache Möglichkeit, `libfirm` um Optimierungen für Closures zu erweitern, jedoch keine optimale. Die Knoten erhalten Informationen über die Beziehungen zwischen unterschiedlichen Funktionen, die im Konzept von Firm jedoch voneinander unabhängig sind. Am Beispiel von Thorin kann man sehen, wie eine Zwischendarstellung, die solche Beziehungen im Grundkonzept enthält, die Optimierungen vereinfachen und verbessern kann. Deswegen ist es eine Überlegung wert, die Zwischendarstellung von `libfirm` — wenn `libfirm` in Zukunft Closures unterstützen soll — um Beziehungen zwischen Prozeduren zu erweitern.

Ein möglicher schrittweiser Übergang ist folgender:

Firm-Graphen werden um Beziehungen zwischen Prozeduren erweitert. Sie treten an die Stelle von Closure-Knoten. Es besteht weiterhin eine Phase, in der diese Abhängigkeiten — wie beim Closure-Lowering — für spätere Phasen eliminiert werden. Nach und nach können bestehende Optimierungen so erweitert werden, dass sie die Abhängigkeiten verstehen und ihre Informationen ausnutzen können. So kann

das Lowering der interprozeduralen Abhängigkeiten Schritt für Schritt nach hinten verschoben werden.

Literaturverzeichnis

[1] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st ed., 2014.

[2] Ecma International, *ECMAScript 2016 Language Specification*. 7th ed., June 2016.

[3] "The python language reference." `https://docs.python.org/3/reference/`.

[4] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pp. 75–, IEEE Computer Society, 2004.

[5] "The GCC internals documentation." `http://gcc.gnu.org/onlinedocs/gccint/SSA.html`. Retrieved: 8 May 2017.

[6] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot™ client compiler for Java 6," *ACM Transactions on Architecture and Code Optimization*, vol. 5, pp. 7:1–7:32, May 2008.

[7] A. W. Appel, *Modern Compiler Implementation in ML*. New York, NY, USA: Cambridge University Press, 2004.

[8] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[9] M. Braun, S. Buchwald, and A. Zwinkau, "Firm—a graph-based intermediate representation," Tech. Rep. 35, Karlsruhe Institute of Technology, 2011.

[10] "The bytecode2firm project." `https://github.com/libfirm/bytecode2firm`.

[11] "The cparser project." `https://github.com/libfirm/cparser/`.

[12] A. Appel, *Compiling with Continuations*. Cambridge University Press, 2006.

[13] R. A. Kelsey, "A correspondence between continuation passing style and static single assignment form," *ACM SIGPLAN Notices*, 1995.

[14] R. K. Dybvig, *Three Implementation Models for Scheme*. PhD thesis, Chapel Hill, NC, USA, 1987. UMI Order No. GAX87-22287.

[15] Z. Shao and A. W. Appel, "Space-efficient closure representations," *SIGPLAN Lisp Pointers*, vol. VII, pp. 150–161, July 1994.

[16] A. W. Appel and T. Jim, *Optimizing closure environment representations*. Princeton University, Department of Computer Science, 1988.

[17] R. Milner and M. Tofte, "The definition of standard ml," 1990.

[18] R. Leißa, M. Köster, and S. Hack, "A graph-based higher-order intermediate representation," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, (Washington, DC, USA), pp. 202–212, IEEE Computer Society, 2015.

[19] "Mlton performance." `http://mlton.org/Performance`.

[20] "Mlton whole-program optimization." `http://mlton.org/WholeProgramOptimization`.

[21] "Mlton compiler overview." `http://mlton.org/CompilerOverview`.

[22] "Mlton SSA simplify." `http://mlton.org/SSASimplify`.

[23] H. Cejtin, S. Jagannathan, and S. Weeks, "Flow-directed closure conversion for typed languages (extended summary)," in *In ESOP '00 [ESOP00*, pp. 56–71, Springer-Verlag.

[24] "Text.Parsec.Token documentation." `https://hackage.haskell.org/package/parsec-3.1.11/docs/Text-Parsec-Token.html`.

[25] D. Leijen and E. Meijer, "Domain specific embedded compilers," *SIGPLAN Not.*, vol. 35, pp. 109–122, Dec. 1999.

[26] "malloc specification." `http://pubs.opengroup.org/onlinepubs/9699919799/functions/malloc.html`.

[27] J. Bechberger, "Besser benchmarken." `http://pp.ipd.kit.edu/publication.php?id=bechberger16bachelorarbeit`, Apr. 2016.

Erklärung

Hiermit erkläre ich, Daniel Krüger, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift